NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

DESIGN OF A LOAD-BALANCING ARCHITECTURE FOR PARALLEL FIREWALLS

by

William L. Joyner

March 1999

Thesis Advisor:	Cynthia Irvine
Second Reader:	Geoffrey Xie

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Design of a Load-Balancing Architecture For Parallel Firewalls		5. FUNDING NUMBERS
6. AUTHOR(S) William L. Joyner		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Because firewalls can become a potential choke point as network speeds and loads increase, the Navy needs a cost-effective means of increasing data rate through firewalls by placing several machines in parallel and balancing the traffic load among them. Current firewall architectures consisting of multiple machines do not balance load among machines and require that each type of traffic be allocated to a machine dedicated to processing specific protocols. This situation creates a performance bottleneck.

This thesis proposes a load-balancing firewall architecture to meet the Navy's needs. It first conducts an architectural analysis of the problem and then presents a high-level system design as a solution. Finally, the thesis provides a detailed system design, targeted for the BSD/OS operating system. The detailed design describes the state transitions, data types and databases, functional interfaces, and threads of execution for a modular layered software architecture.

The result of this thesis is a procedural blueprint for implementation of a firewall architecture, from both software and hardware perspectives, that should mitigate the performance bottleneck. The software architecture is easily verifiable due to its modular, layered design; does not affect either the commercial routers or firewall products; and provides an administrative interface for performance tuning.

14. SUBJECT TERMS Load balancing, Parallel/distributed processing, Firewalls, Routing			15. NUMBER OF PAGES 138
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

DESIGN OF A LOAD-BALANCING ARCHITECTURE FOR PARALLEL FIREWALLS

William L. Joyner
Lieutenant, United States Navy
B.S., Virginia Tech, 1989

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Author: [illegible]
William L. Joyner

Approved by: Cynthia E. Irvine
Cynthia Irvine, Thesis Advisor

[illegible]
Geoffrey Xie, Second Reader

Dan C Boger
Dan Boger, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Because firewalls can become a potential choke point as network speeds and loads increase, the Navy needs a cost-effective means of increasing data rate through firewalls by placing several machines in parallel and balancing the traffic load among them. Current firewall architectures consisting of multiple machines do not balance load among machines and require that each type of traffic be allocated to a machine dedicated to processing specific protocols. This situation creates a performance bottleneck.

This thesis proposes a load-balancing firewall architecture to meet the Navy's needs. It first conducts an architectural analysis of the problem and then presents a high-level system design as a solution. Finally, the thesis provides a detailed system design, targeted for the BSD/OS operating system. The detailed design describes the state transitions, data types and databases, functional interfaces, and threads of execution for a modular layered software architecture.

The result of this thesis is a procedural blueprint for implementation of a firewall architecture, from both software and hardware perspectives, that should mitigate the performance bottleneck. The software architecture is easily verifiable due to its modular, layered design; does not affect either the commercial routers or firewall products; and provides an administrative interface for performance tuning.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my thanks to the following people for their part in helping me to complete this thesis. While some people's roles were bigger than others were, they all were necessary for me to reach a successful conclusion in my work.

- Dr. Cynthia Irvine, my advisor, whose guidance and patience was invaluable. She continued introducing me to new ideas through the very end.
- Dr. Geoffrey Xie, for his input on matters of networking.
- Valerie Brooks, for sharing her knowledge of UNIX system administration and for tolerating the problems that I caused her as a student.
- Paul Clarke, Cindy Holden, and Anatacia Cruz-Tokar, for putting up with me in the lab and for getting me the networking equipment that I needed for the project.
- Ken Lam, for his technical assistance on current firewall configurations being used by SPAWAR.

I. INTRODUCTION

With the advent of the IT-21 [Ref. 1] and Navy Virtual Internet (NVI) [Ref. 2] initiatives, the United States Navy is moving toward becoming fully connected by computer networks. Navy policy [Ref. 24] requires that protection mechanisms such as firewalls be used to prevent unauthorized access to its networks by users on open networks such as the Internet. The Navy's Space and Naval Warfare Center (SPAWAR) installs and maintains firewalls at regional network operating centers (NOC) through which most Navy network traffic flows. Already, the volume of network traffic passing through these NOCs requires multiple firewall machines to be placed in parallel to keep pace. The current parallel firewall architecture uses its machines inefficiently, resulting in potential denial of service for one or more network protocols (e.g. all email or HTTP traffic) even though the volume of network traffic may be far less than the total capability of the firewall machines. SPAWAR has found no viable commercial solution to this problem. The Navy needs a cost effective, load-balancing firewall system for the network operating centers.

A. BACKGROUND

The Navy has NOCs located at its Naval Computer and Telecommunications Area Master Stations (NCTAMS) to process communications between fleet ships and shore based networks such as the Defense Message System, the non-classified information packet routing network (NIPRNET), and the Internet. NOCs are also located at the Pentagon and major shore commands such as Commander, Naval Air Forces (COMNAVAIR). Future plans entail NOCs being strategically placed around the world with each NOC being responsible for routing Navy network traffic to and from a defined geographic area.

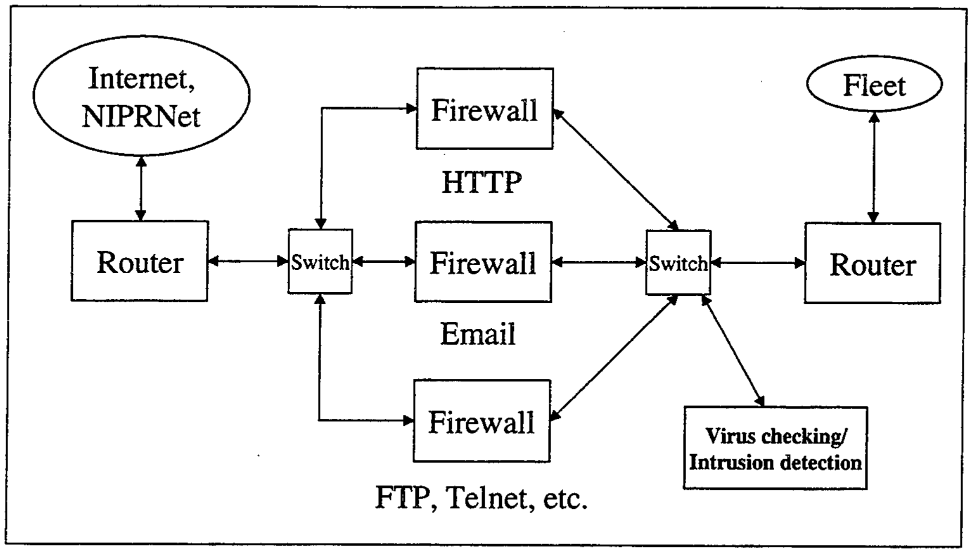

Figure 1 Current firewall architecture

Following Navy policy, each NOC has a firewall between it and the Internet or other open networks to which the NOC may be attached. The SPAWAR Systems Center Charleston, South Carolina builds and maintains the NOC firewalls using commercial off-the-shelf (COTS) products. The volume of network traffic passing through the NOC firewall necessitates placing multiple machines in parallel to cope with the traffic without affecting service. The current parallel firewall architecture consists of an internal router, several parallel firewall machines, an external router, and servers for such functions as virus checking and intrusion detection as shown in Figure 1. Responsibility for connection-oriented (e.g. TCP) traffic is allocated to the firewall machines by traffic type. For example, one machine may process only HTTP traffic, another machine may process only SMTP traffic, and a third machine may process other miscellaneous services such as FTP and Telnet. Non-connection-oriented traffic is routed using the open shortest path first (OSPF) protocol.

OSPF is a link-state protocol. With link-state protocols, decisions are made based on the status of chosen characteristics for each link (i.e. path from one node to its immediate neighbor in a network). OSPF has some inherent load balancing capability so that non-TCP traffic would either be distributed in a round robin fashion or go to the firewall machine currently determined by the protocol to have the lightest load. RFC 2328 [Ref. 3] describes OSPF in the following way:

> OSPF routes IP packets based solely on the destination IP address found in the IP packet header. ... In a link-state routing protocol, each router maintains a database describing the [system's] topology. This database is referred to as the link-state database. Each participating router has an identical database. Each individual piece of this database is a particular router's local state (e.g. the router's usable interfaces and reachable neighbors). The router distributes its local state throughout the [system] by flooding. All routers run the exact same algorithm, in parallel. From the link-state database, each router constructs a tree of shortest paths with itself as root. This shortest-path tree gives the route to each destination in the [system]. ... When several equal-cost routes to a destination exist, traffic is distributed equally among them. The cost of a route is described by a single dimensionless metric. ... This cost is configurable by the system administrator. The lower the cost, the more likely the interface is to be used to forward data traffic.

The current configuration uses identical firewall machines so the administrator can input an arbitrary cost as long as the cost is the same for each path. If a heterogeneous mix of firewall machines is used, then the system administrator must input costs indicating the performance capability of each machine relative to the other machines. Since the OSPF protocol routes packets based solely on IP address, it is inappropriate for connection-oriented traffic. The reason is that once a connection is established through the firewall, all subsequent packets for that connection must go through the same machine. OSPF may send packets from the same connection to any of the firewall machines.

Under the current architecture, a flood of traffic of one protocol could cause the firewall machine processing that protocol to become overloaded. Since that machine is the only one designated to process that protocol, the other machines would provide no help even though they may be very lightly loaded. The result is essentially a denial of service for one protocol even though the total capacity of the firewall far exceeds that of the current traffic load. An example scenario in which one traffic type composes most network traffic is a high volume of email traffic being sent during crisis situations or major exercises involving several ships. A more common example would be surges in email traffic seen during certain parts of the typical workday or workweek for shore commands.

SPAWAR's expressed needs are for a firewall architecture with the following characteristics:

- Balances network load (both connection-oriented and non-connection-oriented) across all machines
- COTS-based solution
- total capacity of 100 Mbps now with capability for greater throughput in the future

Network traffic, both connection-oriented and non-connection-oriented, should be balanced among the firewall machines according to each machine's processing capability and current status. This capability will allow each machine to process all types of traffic. While OSPF provides a solution for connectionless traffic, a method for balancing connection-oriented traffic must be devised.

The architecture should be based on COTS equipment to minimize cost and allow for rapid, easy equipment upgrades as technology advances. The Navy is moving away from specialized development wherever practical because of the high cost, effort involved in managing development programs, difficulty keeping up with technology advances, and cost/logistics of creating a training program to accompany the specialized equipment.

The architecture must currently support 100Mbps throughput to keep pace with Fast Ethernet networks. As ATM or Gigabit Ethernet networks are deployed in the Navy, the throughput requirements for the firewall will obviously increase. So, the architecture should allow for upgrade to the higher speed networks.

B. PROBLEM STATEMENT

SPAWAR has found no viable commercial solution to meet its needs for a load-balancing firewall. Commercial products are known to meet some requirements but all arguably have shortfalls. Problems seen with commercial solutions include 1) not balancing connection-oriented traffic, 2) balancing connection-oriented traffic using a weak balancing method (e.g. using the number of connections to a given machine as the metric of load), and 3) having a high cost per component.

C. GOALS OF THIS THESIS

This thesis proposes an architecture that meets SPAWAR's needs for a cost-effective, load-balancing firewall. It also takes into account good security practices, modular design for scalability, and robustness of design. The architecture has the following characteristics:

1. support for a set of heterogeneous machines running firewall software
2. firewall product independence
3. load balancing among the parallel machines that accounts for each machine's current load and its processing capability
4. load balancing for both connection-oriented (i.e. TCP) and connectionless network traffic
5. robustness to account for a firewall machine malfunctioning
6. scalability to allow greater firewall throughput by adding more machines in parallel
7. commonly used COTS hardware and software products for all machines involved in the architecture
8. minimal cost addition to the current firewall architecture
9. minimal effect on configuration and management of the system

II. CHOOSING THE ARCHITECTURE

A. BASIC CONCEPTS

1. Scheduling

Much work has been done in the areas of scheduling and load balancing in distributed and parallel systems. Distributed/parallel systems are sets of autonomous computers connected by a communications network. The development of scheduling algorithms has provided a strong means of improving performance in these systems [Refs. 4, 5]. The object of scheduling is to meet most efficiently the task processing requirements set forth by the owner of a particular system. Some example criteria are that the maximum number of tasks be completed in the least amount of time, that no task is delayed in processing more than a set amount of time, and that higher priority tasks are completed before lower priority tasks.

Optimal scheduling in the general case is known to be a NP complete problem [Ref. 6]. As a result, research focuses on finding optimal or near optimal solutions for subsets of the general case. That is, given particular types of tasks about which some information is known, researchers develop scheduling methods for those task types. The research covers numerous approaches to scheduling. Many of these approaches involve matching the best scheduling algorithm with the appropriate system architecture given a listing of the system requirements and the types of tasks the system is intended to perform. Architectures of interest range from a single-processor machine on which all tasks run to a widely dispersed network of heterogeneous computers connected in a complex configuration.

Scheduling research covers static versus dynamic, preemptive versus non-preemptive, and adaptive versus non-adaptive scheduling [Ref. 6]. Static scheduling requires having, at compile time, the complete set of tasks to be processed so the required runtime parameters (e.g. execution time and communication delays) and computer resources (e.g. processors, memory, and input/output devices) for each task are known before any scheduling occurs. Having this information allows an optimal schedule to be found for the tasks, because the solution is for a specific case vice the general case. However, static scheduling offers no flexibility for changing or adding tasks without recompiling the entire algorithm.

Dynamic scheduling assumes no compile-time knowledge about runtime parameters of tasks. So, tasks may be added or changed after processing has begun. An advantage of dynamic scheduling is that it offers the flexibility needed in an environment with changing tasks. A disadvantage is that the scheduling is more difficult and brings processing overhead not required in static scheduling. Each time the list of tasks changes, the scheduler must revise its schedule to provide a "best fit" arrangement for processing tasks with the available resources. A load-balancing firewall requires dynamic scheduling since each new connection through the firewall is a new task to be completed and the firewall has no knowledge of the amount of data accompanying each connection.

Non-preemptive scheduling does not allow a task to be interrupted once it has begun execution. Once a task has a processor and other resources it needs to execute, it runs to completion using those resources. An advantage is that the scheduler can forget about a task once the task has begun execution, making the scheduling problem less complex compared to preemptive scheduling. It schedules the remaining tasks in its list based on their known parameters and the expected completion time of the currently running tasks. Because of the time-out features of connection-oriented transport layer protocols (e.g. TCP), a parallel firewall will generally use non-preemptive scheduling.

Preemptive scheduling allows the processor and other resources to be taken away from a task with the understanding that the task will regain its resources and proceed to completion later. An advantage of preemptive scheduling is that it provides more flexibility in pursuing an optimal schedule according to the goals of the scheduler. A disadvantage is that it is more complex, and carries more processing overhead, than non-preemptive scheduling because the scheduler must consider whether running tasks should be stopped temporarily or moved to other processors. A parallel firewall may use preemptive scheduling if it is designed to move connections between machines as part of a load-balancing scheme.

In adaptive scheduling, the scheduler changes its scheduling behavior based on current behavior of the system, possibly in response to previous decisions made by the scheduler. For example, a scheduler may use a scheduling algorithm suited for heavy loads when it recognizes a heavy load on a system and it may use a different algorithm under lighter load. Non-adaptive scheduling may also use feedback from the system, but the scheduler does not change its behavior in response to the feedback. An advantage of adaptive scheduling is that it may provide more efficient scheduling over a wider range of system conditions. A disadvantage is greater complexity in the scheduling algorithm. Non-adaptive scheduling may provide less efficiency considering the whole of system conditions, but is generally less complex. A parallel firewall may use either adaptive or non-adaptive scheduling depending on the requirements of the design.

Some scheduling algorithms, such as a round-robin algorithm, work with no knowledge of the machines or processors for which they schedule tasks. Other algorithms, regardless of whether they are adaptive or not, use various types of system information to schedule tasks. The algorithms may use such information as the pre-determined maximum processing capability of machines or processors in the system, the progress of tasks being executed, historical information about runtimes of tasks on various machines, and network traffic load. In general, algorithms that use system information in their scheduling provide more robust and responsive service than those that use no system information. These algorithms are said to be smarter algorithms.

2. Load Balancing

Load balancing attempts to maximize throughput of a distributed system processes tasks by trying to equalize load among all computers in the system according to some pre-defined set of rules. It uses the concepts developed for scheduling algorithms as its basis. The term load balancing as used in this paper would more accurately be called load distribution. Load-distributing algorithms can be further classified as being load-sharing or load-balancing [Ref. 7]. Load-sharing algorithms attempt to maximize the rate at which a distributed system processes jobs by trying to avoid situations in which some computer is idle while another has multiple jobs. Load-balancing algorithms share the same goal as load-sharing algorithms but they go a step further; they attempt to equalize the loads on all computers vice looking only for computers with no task while others have multiple tasks.

Load balancing is typically dynamic in that it uses system state information to schedule new tasks or move tasks between processors during runtime. Tasks are initially scheduled using some dynamic scheduling algorithm. The load-balancing algorithm monitors system information and may move tasks to different machines as the system state changes.

A dynamic load-balancing algorithm typically has four components: a transfer policy, a selection policy, a location policy, and an information policy [Ref. 7]. The transfer policy determines when a node is loaded to the point that a task should be moved elsewhere, usually by defining some threshold load value. A node exceeding the threshold is called a sender node. The selection policy determines which task should be moved from a sender node to another node. The location policy determines a suitable receiver location to accept the task transfer. If no suitable receiver is found then the task remains at the sender node. The information policy determines what system information will be used in making load-balancing decisions.

As an example, consider an information policy that uses a pre-defined measure of the processing capability of each machine, the expected runtime of tasks on some "standard" machine, and a measure of the expected time to transfer a task from one machine to another. The selection policy may be that the algorithm selects for transfer the task with the shortest expected runtime that will drop the sender machine below its transfer threshold. The location policy may be that the expected runtime on a receiver machine plus the expected time to transfer the task between machines must show an expected speedup in task completion without causing the receiver machine to exceed its transfer threshold. If a machine becomes loaded beyond a threshold determined by the transfer policy, the algorithm chooses a task for transfer and looks for a machine to which a task can be transferred. If it finds a satisfactory receiver machine then it will transfer the task.

One question that arises with dynamic, adaptive load sharing schemes is how much system information to use. Research has shown that using very small amounts of system information in simple ways yields dramatic performance improvement relative to the no load balancing case [Ref. 8]. The performance is close to that expected from much more complex schemes but with much less overhead due to data collection and communication delay.

Much research has been done in dynamic, adaptive scheduling for heterogeneous distributed systems. Some of these research projects are the V-System [Ref. 9], Sprite [Ref. 10], Condor [Ref. 11], Stealth [Ref. 12], Gammon [Ref. 13], MSHN [Ref. 14], and Globus [Ref. 15]. These types of adaptive systems typically allocate resources to tasks based on the information known about the tasks. For example, the particular task or a similar task may have run on the system before, so past performance data may be collected in a task history file. Another possibility is that a task new to the system may have had analysis performed on it before scheduling so it includes prediction information to be used for scheduling. The dynamic scheduling systems also monitor the progress of scheduled tasks and perform checks to determine whether tasks should be dynamically rescheduled (e.g. moved to another machine) to finish more quickly or efficiently as various resources in the system become available. Some of these systems also monitor resources and dynamically adapt to changing conditions. So, these types of projects blur any line that may exist between research in scheduling and research in load balancing.

3. Clustering

Computers in a distributed system must have some way of acting together as a single virtual machine. This is sometimes called clustering. Three approaches, all areas of active research, enable a set of computers to be clustered: distributed operating systems, resource management systems (RMS), and distributed computing environments (DCE). Distributed operating systems make the set of computers act as if they are actually hardware in one machine. That is, distributed operating systems micro-manage each machine's resources (memory, I/O devices, etc.) rather than each machine running an independent operating system. They accomplish this job by having each machine run a copy of the distributed operating system software. The machines' operating systems work together as the overall distributed operating system. So, one operating system controls all of the machines just as in a single computer. Distributed operating systems are rarely used in the United States but are used to some extent in Europe.

RMSs allow each machine to be managed by its own, independent operating system. A RMS has some central elements such as a scheduler, a task monitor, and a database of historical information, and a set of machines for performing tasks. The scheduler allocates individual or groups of machines to tasks. The machines then work on the tasking, reporting their progress to the monitor. The monitor or the scheduler makes decisions about reallocating resources if necessary. RMS examples are Condor, MSHN, and Globus.

DCEs accomplish the creation of virtual machines by use of libraries or programming language support. In DCEs, each computer has the libraries or language support modules necessary to allow machines to work together seamlessly as a virtual machine. These libraries and modules provide a coordinated means of communication between computers in the DCE while the computers execute programs written using them. DCE examples are Linda [Ref. 16], PVM [Ref. 17], and MPI [Ref. 18].

Another area of research is the use of distributed shared memory (DSM) schemes to create clusters of computers. In DSM schemes, each computer in a cluster is controlled by its native operating system but it runs an application that maps the machine's memory into a larger, shared memory address space. This mapping means that the virtual machine's contiguous memory spans the memory of all the machines in the cluster. As a simple example, a three-machine cluster with 32MB of memory in each machine would enable the creation of a virtual machine with 96MB of memory. Each machine's memory address space goes from 0 to 32M but from the perspective of the virtual machine, machine one has memory locations 0-32M, machine two has 32M-64M, and machine three has 64M-96M.

All applications run on top of the DSM mapping. So, an application running on one machine may have its data located in the memory of any of the machines.

Any of these clustering techniques could be used for a load-balancing firewall though some may be far more difficult to implement than others. Another option was to use concepts from more than one technique to create a solution to the problem at hand.

B. SCHEMA CONSIDERED

Several options were available for developing a load-balancing system architecture for parallel firewall machines. These options included the following ideas:

- Modify the source code in the routers to make them balance connection-oriented load and let OSPF balance connectionless traffic load. Since all network traffic between the protected enclave and the outside world passes through the routers and a router's normal tasking is to make routing decisions for packets, the routers are a logical location for load balancing to occur. The routers could distribute among the various firewall machines all new TCP connections destined for the firewall. They could then route subsequent packets associated with those connections to the appropriate machine. They could record the mediating firewall machine's address for new connections from the firewall so they could route return traffic to the same machine. The routers could use the existing capability of OSPF to distribute connectionless traffic or they could use the decision process used for new TCP connections. By modifying the router code, the physical network could look just as it does in Figure 1.
- Create a firewall cluster using known clustering techniques or distributed shared memory across the machines so all firewalls know the state of every connection through the cluster. Using this method, the firewall machines would use a separate network to communicate among themselves as shown in Figure 2. The firewall machines could use this network either to exchange information on the state of every connection through the firewall or to facilitate quick access to

memory in a shared memory scheme that was physically located on another firewall machine, depending on the clustering technique chosen.

- Use load-balancing Domain Name Servers (DNS) to distribute new connections and connectionless traffic to the various firewall machines. Figure 3 shows both internal and external servers in the firewall architecture. Whenever a request is made from the outside world for the address of the protected enclave's domain, the external DNS answers with the address of the firewall machine that should receive the next portion of network traffic. Clients in the protected enclave must ask the internal DNS for the address of the gateway to the outside world. The DNS responds with the address of the firewall machine that should receive the next portion of network traffic.
- Use intermediate load-balancing machines between the routers and the firewalls, as shown in Figure 4, to distribute traffic among the firewall machines. The load-balancing stations balance network traffic load destined for the firewall; they simply pass traffic transiting from the firewall with no action on their part.

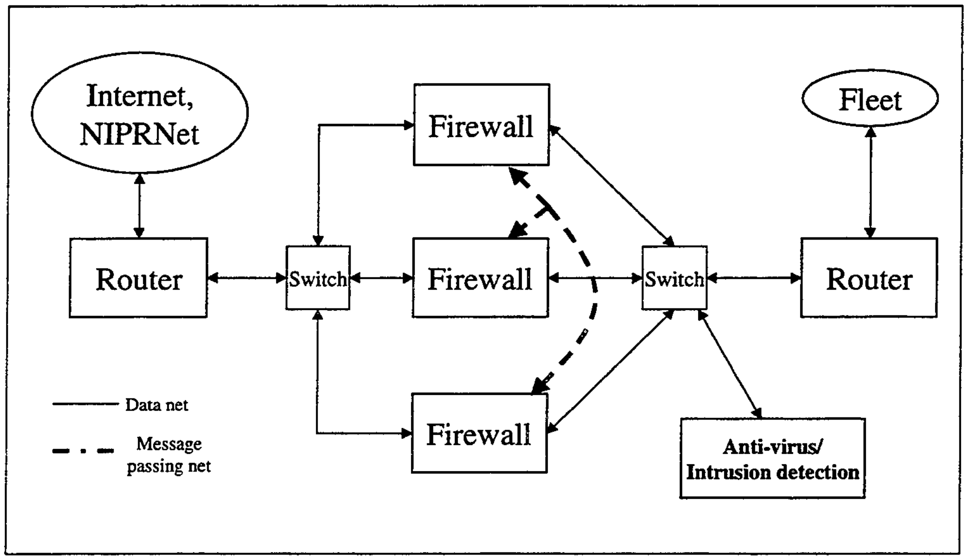

Figure 2 Clustering or DSM scheme

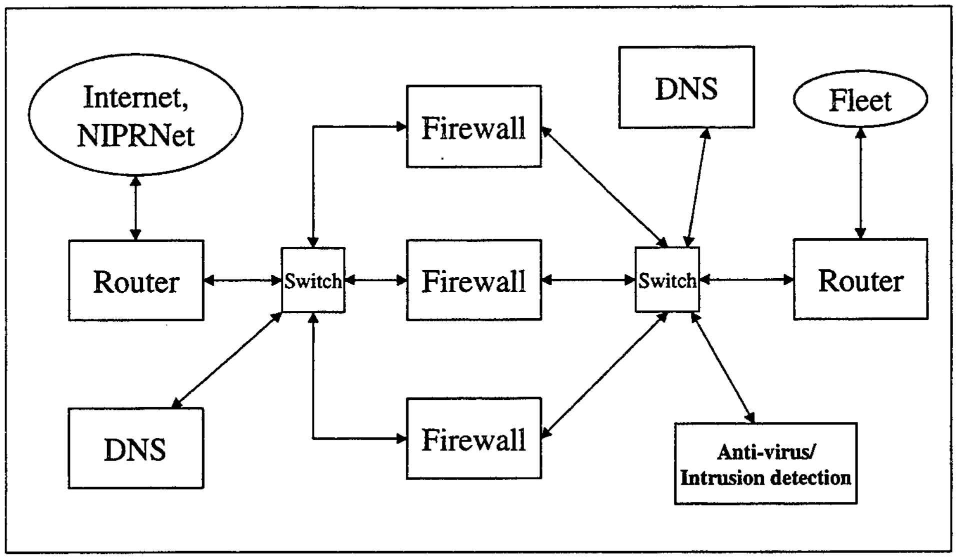

Figure 3 DNS scheme

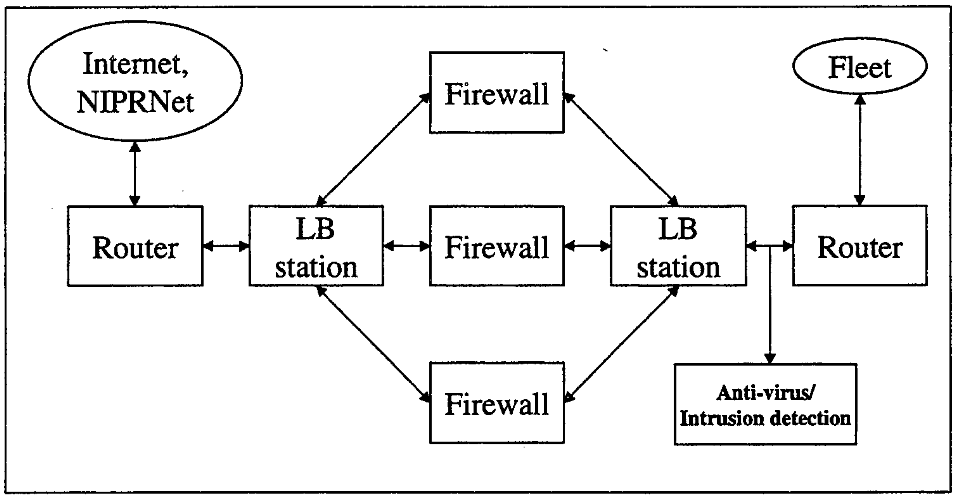

Figure 4 Intermediate routing station scheme

Given that the routers used in the firewall network are proprietary, modifying the source code was eliminated as an option. While the most convenient method of balancing firewall load would be via the routers, modifying the source code would nullify any software warrantee or maintenance contract unless the specialized load-balancing software was developed by the router manufacturer, a time consuming and costly venture.

Creating a firewall cluster using a DSM scheme would be an elegant solution from the computer science viewpoint. Using this scheme, the routers would need to use only

OSPF to balance load between individual firewall machines. Another advantage is that OSPF could provide fine grain load balancing in that it could balance load almost on a per packet basis (as opposed to a per connection basis), although this task might require modifiying the OSPF implementation used.

However, the DSM option was eliminated because of potential serialization problems and lack of compatibility between DSM schemes and TIS Gauntlet, SPAWAR's current firewall product of choice. The concern with serialization stems from the fact that Gauntlet uses proxy servers. Using proxies, connections through the firewall are in two parts: a connection between the protected user and the firewall proxy, and a connection between the proxy and the outside user. The proxy acts as a middle-man ensuring isolation between the two connections. The potential serialization problem in the DSM scheme is that the inner router might send consecutive packets of an outbound TCP connection to different firewall machines. The state of the proxy TCP connection between the firewall and the outside destination would be kept in shared memory. If a firewall machine receiving a later TCP packet in a connection processed the packet before a machine receiving an earlier packet, then the packets between the firewall and destination would be numbered incorrectly, causing problems at the destination.

It may be possible to avoid the serialization problem by recording additional correlation data in the shared memory, but the concern regarding compatibility between Gauntlet and DSM schemes becomes an issue. In general, the compatibility concern is that Gauntlet might not be able to be run on top of a DSM scheme because Gauntlet provides no programming interfaces that could be used to access it.

Using the researched RMSs for a firewall cluster was eliminated as an option. RMSs are generally designed for use with relatively long tasks that are controlled wholly within the cluster. However, a firewall cluster provides proxy services or stateful inspection (if using a product such as Checkpoint's Firewall-1) for network traffic that has both origin and destination outside the cluster. The cluster has no control over the length or status of each network connection between endpoints. Thus, the tasks are not controlled within the cluster. Also, network traffic is characteristically frequent and short term, not the longer term tasks for which RMSs are designed. So, not only would the advantages of the powerful load balancing techniques developed for use in an RMS not be realized in the firewall cluster, but these very techniques carry processing overhead that may hurt performance of the firewall cluster.

Using an existing DCE was eliminated as an option. The individual firewall machines can work independently; their only requirement is that the traffic be routed to them correctly. A DCE typically has a master station that partitions a task, hands it out to constituent machines, and collects results to construct the full solution to a computational task. So, the DCE did not seem to fit the problem. Furthermore, it is expected that a streamlined algorithm written specifically for the firewall cluster would not carry as much overhead as a DCE.

Using a load-balancing DNS seemed to be a viable option for the solution. Use of the DNS allows each firewall machine to act independently. The DNS is involved only in providing an IP address for the appropriate firewall machine. The idea is that any new connection through the parallel firewall would need an address for a firewall machine. Load balancing can be accomplished by having the DNS provide the firewall machines' addresses to domain name requests such that the load is balanced among the machines. Once a connection is established, all traffic for that connection goes through the same machine without intervention by the DNS.

A DNS scheme would be appropriate for traffic inbound to the protected enclave but not for outbound traffic. The problem for outbound traffic is that the parallel firewall should be transparent to internal clients. However, having multiple firewalls requires both a load-balancing DNS to provide firewall addresses and a normal internal DNS to translate domain names to IP addresses.

DNS uses a resolver on each client machine and one to three servers to process domain name requests [Ref. 25]. Resolvers act on behalf of client programs, communicating with servers to translate domain names to IP addresses. Where multiple servers are used there is a master server and backup servers. If the master server does respond then resolvers try the backup servers. Regardless of the number of servers, resolvers ask one question (i.e. what is the address for this domain name?) and get an answer to that question. If the destination IP address is outside of the protected enclave, then the client sends the packets to a gateway machine to the outside world.

Clients are normally configured so that they always know the identity of their gateway. Requiring clients to ask a load-balancing DNS for the gateway address necessitates reconfiguring every internal client to use the two DNS scheme vice having the parallel firewall solution solve the problem, an undue burden considering the vast number of clients on ships in each theatre. Another weakness with the DNS scheme is that it provides relatively coarse grain load balancing since it balances load by use of connections rather than on a per packet basis as with a DSM scheme.

Using intermediate routing stations between the routers and the parallel firewall also seemed to provide a viable solution. This scheme allows the needed load-balancing functionality that cannot be added to the commercial router to be added to the network. These routing stations do not replace the routers and all of the functionality that they provide, but rather they simply provide the specific functions needed for load balancing. Each intermediate station performs the same type of load-balancing function as the DNS described in the previous option. Under this scheme the intermediate station decides where to send a new connection. It then routes all traffic for that connection through the same machine with no further decision required on its part. The weaknesses in using an intermediate station are the coarse granularity discussed for the DNS and the fact that the scheme requires all traffic to pass through another machine. Another machine in series means another potential failure or choke point.

C. CHOICE OF ARCHITECTURE

Given the previously stated constraints and discussion of options, the architecture shown in Figure 5 seemed a best fit for the solution. Figure 5 shows a load-balancing DNS providing load-balancing services for inbound traffic, a set of independent firewall machines (i.e. each controlled by its native operating system and running its own firewall software), and an intermediate controller providing load-balancing services for outbound traffic. The controller is discussed in the previous section as an intermediate routing station.

The DNS is the least intrusive method of load balancing because it has minimal impact on traffic throughput for the parallel firewall. Setting a short time in the time-to-live (TTL) field for domain name responses should ensure that machines outside the protected enclave almost always have to query the DNS before talking to one of the firewall machines. The intermediate controller providing load-balancing services for outbound traffic allows us to encapsulate the parallel firewall within the internal and external routers. That is, the load-balancing firewall scheme is transparent to all machines that are not between the two routers.

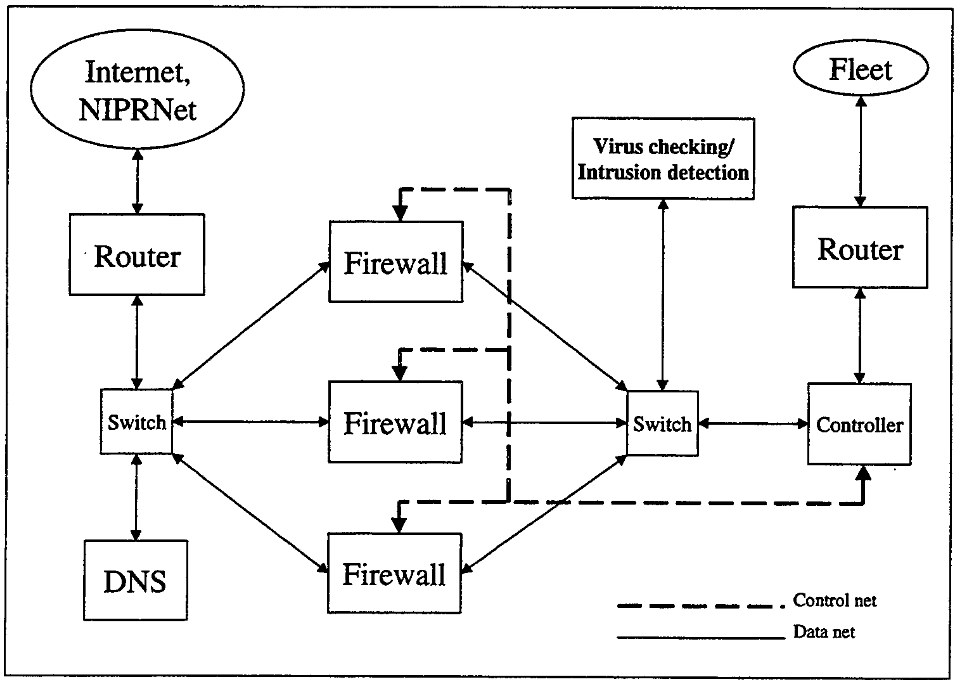

Figure 5 Proposed firewall architecture

Security concerns govern the way the hardware components of the firewall scheme are connected and the way the components should be configured. Integrity, authenticity, and availability are the computer security concepts with which we are concerned here. The policies for these concepts are as follows:

1. The specific implementation and operation of the architecture should be transparent to users.
2. Unauthorized users should not be able to spoof control signals to affect the operation of the architecture.
3. Data congestion should not prevent control signals from reaching the firewall components.
4. A firewall machine must mediate all access from one side of the firewall architecture to the other.

The first policy deals with system integrity. The structure, operation, and internal communications of the architecture should not affect the way outsiders (i.e. machines outside the routers) communicate through the firewall. Likewise, outsiders have no legitimate reason to communicate with specific components of the architecture. The second policy also pertains to integrity, authenticity, and availability. Outsiders should not be able to insert fake control signals by impersonating a component of the architecture. So, fake control signals should not be able to compromise the integrity of the architecture or availability of the architecture's services. The third policy addresses availability. Control signals should be able to reach the components regardless of the state of the data path. This arrangement allows problems to be corrected via the control path. The last policy addresses completeness so that all communication between the protected enclave and the outside world must pass through a firewall machine.

Figure 5 shows separate data and control networks. Data and control signals travel on physically separate media accessed via separate interfaces so that the signals do not compete for time on a common network. From an availability standpoint, the following advantages arise:

- Congestion or other problems on the data path do not prevent control signals from reaching the various components.
- Control signals between components do not add to the amount of traffic on a common network.

- Remote administration is supported because the state of the data path does not affect the ability of an administrator to reach the components.
- Control signals should not be able to be spoofed via the data path.

From a system integrity standpoint, physically separate networks and properly configured components prevent cross-communication between data and control networks. They also hide both control signals and the existence of some components in the architecture from the outside world. Only the controller is hidden initially; other components, such as an administrator station or remote access station, added to the control net later would also be hidden. Proper configuration of the routers prevents unintended signals from passing outside the routers, hiding the specific internal implementation and operation and thus contributing to the integrity of the architecture.

From an authenticity standpoint, only legitimate firewall components should be able to communicate on the control network. Additional authenticity safeguards would need to come from the software controlling the architecture.

This line of reasoning does not imply any measure of assurance for integrity or authenticity policy enforcement should the underlying platforms be compromised. However, proper configuration of the firewall components provides some amount of logical separation between the data and control networks. Since control signals use physically separate media from data signals, the operating system and software on each component determine the level of assurance for integrity and authenticity.

The DNS is not connected to the control network. This design decision avoids a physical path around the firewall machines, enforcing the completeness policy. So, any control signals to the DNS must reach it via the data network. Proper configuration of the routers prevents control signals to the DNS from reaching the outside world. Another option is to use a one-way fiber arrangement so that the DNS receives but cannot transmit signals on the control network. However, the one-way fiber would provide an outbound path around the firewall machines, a risk that must be weighed against potential benefits in firewall architecture performance.

D. CONFIGURATION ISSUES

Several configuration issues arise when considering the security concerns mentioned in the preceding section. Issues are discussed here by hardware component, but no specific implementation details are provided because they may differ by manufacturer.

The routers should be configured to block all traffic addressed specifically to or from the controller. The controller may communicate with the DNS but spoofed communications from the controller will be prevented. One result is that the internal router must be the gateway addressed by internal clients. The internal router should block all communication with the DNS and the external router should block all traffic addressed to the DNS other than domain name requests.

The operating systems for all machines in the architecture should be in a stripped down configuration. Protocols such as Telnet and FTP should not be installed. Commands such as rlogin and rsh should be disabled. The idea is to create a machine upon which is installed only those components of the operating system required for the machine to do its job. The machine should be difficult to log into without access to its control terminal. Lastly, the machines should run only the firewall and control programs necessary for proper operation of the architecture.

The DNS should have only one network card unless a one-way fiber is used to connect it to the control net. As with the controller, the DNS should have minimum operating system components installed and should be difficult to log into without access to its control terminal. The DNS should accept control signals on the data net only from the controller. Should one-way fiber be used for control net connection then no control signals would be accepted from the data net; control signals would come from the control net.

E. CHOICE OF LOAD BALANCING ALGORITHM

Dynamic, non-preemptive, non-adaptive scheduling seemed appropriate for the load-balancing firewall scheme. The lack of a predetermined, finite task set dictates the use of dynamic scheduling. Once a firewall machine mediates a TCP connection through the firewall, then either all subsequent TCP traffic associated with that connection must go through that machine or all machines in the firewall must know the state of the connection at any given time. The latter method requires a tremendous amount of message passing between firewall machines. Also, the lack of interfaces to Gauntlet makes this option undesirable.

Balancing load by directing new connections to the appropriate machine and routing all subsequent traffic for that connection to the same machine solves the problem of processing connection-oriented network traffic. The method is non-preemptive because a task (i.e. connection) cannot be moved between machines once it has been begun. Connectionless traffic may be distributed by sending it to the firewall machine that is scheduled to receive the next connection. These methods work well for both the DNS and the controller.

Adaptive scheduling does not appear to be needed in the prototype design so non-adaptive scheduling is used in this thesis for its simplicity. Adaptive scheduling may be useful should bench testing of firewall machines show a significant difference in processing times for various protocols. In that case, an additional machine, attached to a hub placed in the data network between the interior switch and the interior router, would be warranted. The machine would analyze network traffic for its composition, by protocol, and would provide feedback to the controller. The controller would use this information to adapt its scheduling behavior accordingly.

A statistical distribution algorithm was chosen for scheduling in the controller. The controller can distribute new connections based on the current load and predetermined processing capabilities of each firewall machine. A statistical distribution algorithm was also chosen for scheduling in the DNS. However, control signals are not available to the DNS without using the data network for control communications. So, new connections are distributed to each firewall machine according only to its predetermined processing capability. Using a one-way fiber arrangement, load updates would be available to the DNS via the control network so the distribution algorithm used in the controller could also be used in the DNS.

III. SOFTWARE MODULE DESIGN

This chapter presents the software design for the proposed load-balancing architecture. It first discusses general decisions about the architecture's software and then discusses the software on a per-component basis. Appendix A describes the requirements for the software. Appendix B provides a detailed explanation of the data types, modules, and functions involved in the design. The software is designed for the BSD/OS operating system produced by BSDI. It supports IPv4 protocol and Ethernet hardware format addresses.

The general design philosophy for the software is that of modularity and layering as prescribed by Parnas [Ref. 19] and Dijkstra [Ref. 20]. Figure 6 illustrates a modular, layered model. Functional responsibility determines the formation of modules. The modules are arranged in hierarchical levels, each level representing a different level of dependency. Figure 6 shows arrows from higher to lower modules in the overall structure. The arrows represent function calls from higher levels of abstraction to more specific, lower abstraction modules to perform tasks. Having the arrows going always downward creates a partial order among the modules (i.e. there is no potential for loops in the design). A partially ordered, layered, modular design aids verifiability and ease of understanding of the design.

Stevens's *UNIX Network Programming, Vol. 1* [Ref. 21] was used extensively during the design and programming phases of this software. A review of UNIX system data structures and sockets is recommended for those readers not familiar with the subject.

When designing a load-balancing system, some basic decisions must be made about the definition of load on the system, how that load is to be measured, and where it is to be measured. The question of load seems straightforward; it is the amount of network traffic a machine is processing, measured in the unit of choice (e.g. packets per second or bytes per second).

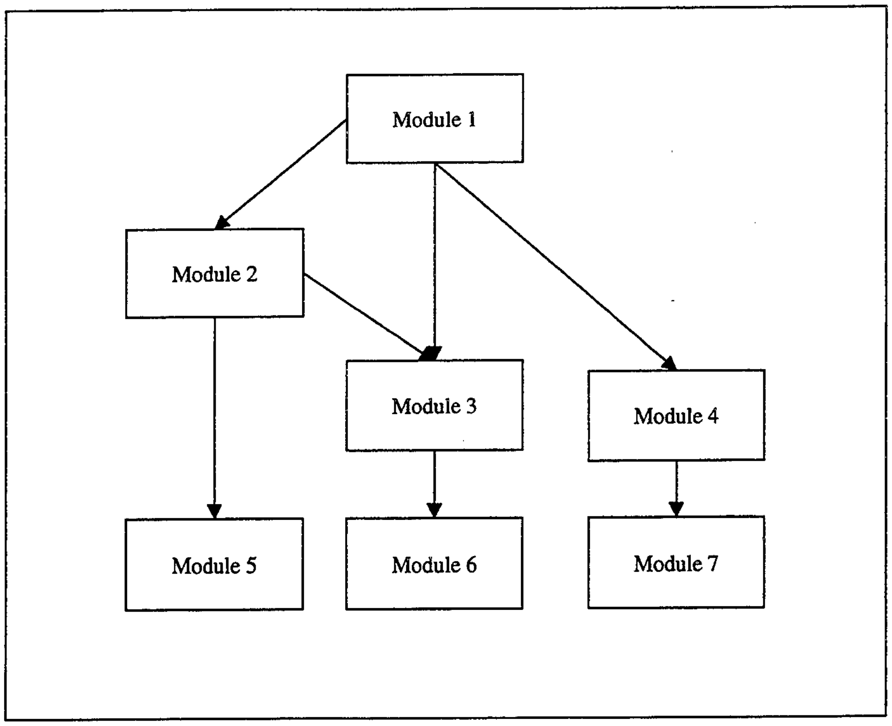

Figure 6 A modular, layered design model

Determining load is a more difficult problem. Any number of metrics can be used for determining load on a firewall machine. For example, the load could be determined by measuring the number of packets or bytes of data coming to a machine on each of its network connections. Other possibilities include using the number of TCP connections through a firewall to estimate load and using the percent usage of the firewall's CPU to determine its load.

Once the metric is decided, the question of how best to estimate load using that metric arises. One has the option of monitoring the metric continuously, sampling it periodically, or determining it by a single snapshot. The options are listed in decreasing order of both processing overhead and accuracy.

Deciding where in the system to monitor the load metric is the last question. Options include having each machine perform its own monitoring, having one component do the monitoring for all components in addition to its own tasking, and adding a new component to the system to perform load monitoring. Factors such as cost, processing overhead, and the chosen metric itself affect which method of monitoring is appropriate for a given situation.

The metric and monitoring technique chosen for a load-balancing system should 1) provide an acceptable level of accuracy in measuring load and 2) have an acceptable cost in processing overhead and communication latency associated with measuring the metric and sending it to the appropriate place for use in balancing load. These requirements are somewhat subjective in nature. Judgement calls must be made as to what is acceptable. In the case of a firewall, the metric and its measurement should not significantly affect the data throughput of the firewall architecture.

The following metrics for measuring how much network traffic a firewall machine is processing were considered for the proposed architecture:

- Measure the number of connections sent to each firewall machine. The measurement could be done by each firewall machine or by the controller and DNS.
- Measure bytes of data sent to each firewall machine. This measurement could be done by the controller and load-balancing DNS machines, by each firewall machine, or by a new component added to the system for this specific purpose.
- Measure the receive queue for each firewall machine. This measurement would need to be taken by the firewall machines.

Measuring load by monitoring the number of connections to each firewall machine was eliminated as an option. This metric would seem to cost relatively little processing overhead. However, it was eliminated because it falls short in providing an accurate measure of the amount of data being processed by a firewall machine. The reason is that each TCP connection is simply an open channel for passing data between machines. As stated in the discussion of the chosen load-balancing algorithm, once a firewall machine mediates a TCP connection, all traffic associated with that connection must go through the same machine. But, no knowledge about the amount of data associated with any given TCP connection is gained simply by knowing that the connection exists. So, knowing only how many connections a particular firewall machine is mediating provides little information about the actual amount of data that the machine must process. For this metric to be useful, some way of mapping associated data volume to connections must be created. The transport layer protocols in use today (TCP and UDP) provide no such mapping.

Measuring the amount of data, in bytes, sent to each machine was eliminated as a complete solution for three reasons. First, the measurement is one that should be done continuously. Having the controller or firewall machines perform the measurement would consume processor cycles and would degrade firewall throughput. Second, the DNS may be able to perform the measurements but the information would benefit only the DNS since it could not transmit on the control network. The DNS could send load updates to the controller via the data network but sending control signals on the data network was discussed previously as undesirable. Third, adding a new component to the architecture solely to measure network traffic sent to each firewall machine would add unnecessary cost and complexity to the architecture.

Note, however, that having the DNS machine monitor inbound network traffic in promiscuous mode may offer a partial solution. The DNS would determine, for its own use in its load-balancing algorithm, the amount of data going to each firewall machine. This scenario would allow the DNS to base its load balancing on both known capabilities and current load of the firewall machines without the need for a one-way fiber arrangement. Testing this option to determine whether the monitoring degrades the DNS's performance is left to future work.

The metric chosen for the proposed architecture is the length of the receive queue. The receive queue contains traffic awaiting processing and located in a firewall's receive buffer. Queue length indicates the amount of work the machine has ahead of it. Knowing this information, together with the known processing capability of the machine (determined by bench testing – see Appendix C), provides an indication of how busy the machine is. It also accounts for both inbound and outbound traffic while other metrics may account for traffic in only one direction. Having each firewall machine determine its own load increases the total work performed at each machine. However, queue length measurement may be performed periodically rather than continuously. This choice mitigates the processing overhead imposed by the sampling system and provides an indication of load that should approximate that of continuous monitoring. How close the sampled estimate is to the results of continuous monitoring is a function of sampling rate and network traffic patterns. The higher the sampling rate, the better the estimate but the more processing overhead that is introduced. Wildly fluctuating network data volumes could degrade the sampling method's accuracy.

As noted earlier, some means of mapping load to connections is needed before the existence of a connection has significant meaning. Recording queue length per TCP connection using the UNIX netstat utility provides a means of mapping load to TCP connections. Knowing the historical data volume associated with a particular connection allows a load-balancing algorithm to use an expected load for that connection in its calculations, even if there is no data passing through the connection at the time of the calculations. Using the greater of current receive queue length and a historical, weighted average of queue length associated with each connection accounts for two special cases. One case is where a connection has a history of high data rate but is quiet at the time of the calculations. By including the weighted average, the algorithm takes into account the previous, and thus expected, connection data rate. The algorithm assumes that the connection will become busy again. The other case is where a connection that was quiet has just been supplied a burst of traffic that must be processed by the firewall. The algorithm must account for data actually in the firewall's receive buffer.

One may argue that simply monitoring total data volume destined for each firewall machine, regardless of the associated TCP connection, provides similar information and saves processing overhead. However, using total data volume to estimate expected load for a particular firewall machine has inherent error in two areas. First, past connectionless traffic volume is a poor indication of its future volume. Also, since connectionless traffic may be sent to any firewall, it is irrelevant to expected load calculations for a particular machine. Second, past data volume associated with connections that have since closed is also irrelevant to expected load. Since the connections are closed, no data associated with those connections will contribute to future load on the firewall machine. The concern is only for load history of connections that exist at the time of the load-balancing algorithm's calculations. So, the predicted load using total traffic volume to a machine as an indicator would contain error for both closed TCP connections and connectionless traffic.

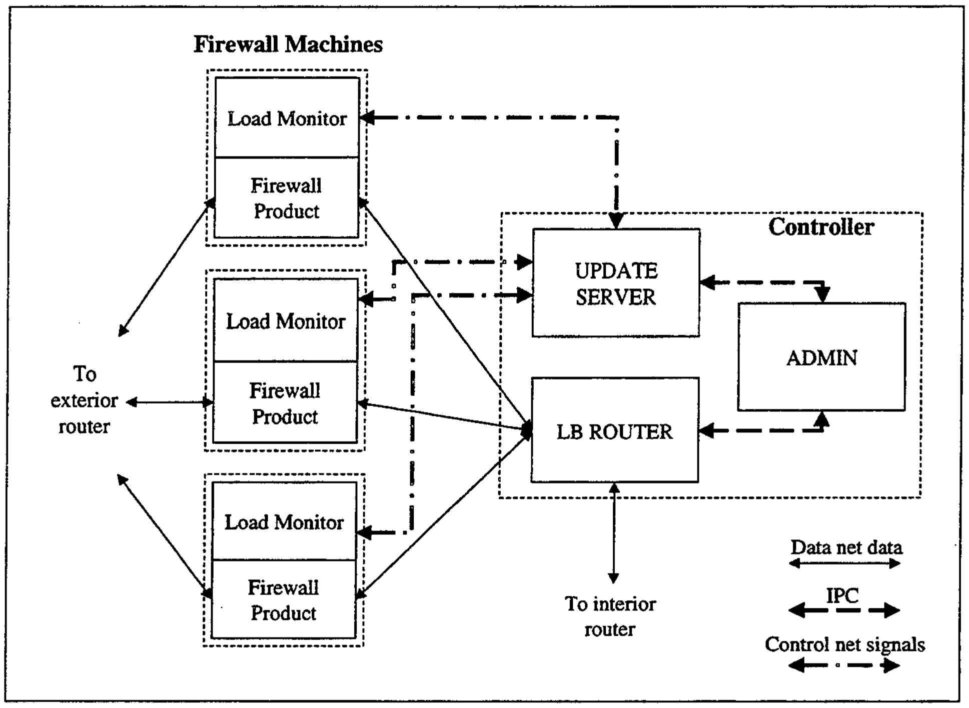

Figure 7 Inter-process communications – global view

If system testing shows that using queue length per TCP connection degrades firewall performance to an unacceptable level, then an alternative method for measuring load will be to construct a profile of the average load associated with a TCP connection via statistical sampling. In this case, multiplying the average load per TCP connection by the number of TCP connections for a given firewall would yield the estimated load for that firewall. Statistical sampling of data associated with TCP connections could be done by the firewall machines or the controller. The statistical sampling would have lower processing overhead, at the cost of a lower level of accuracy, and should yield an acceptable estimate of load for each firewall machine.

Figures 7 and 8 show the communications that occur between the various processes in the software design. Figure 7 shows communications paths between the processes on each machine. Figure 8 shows the specific signals and messages being passed on those communications paths. State diagrams for each process in the software design are provided in subsequent sections.

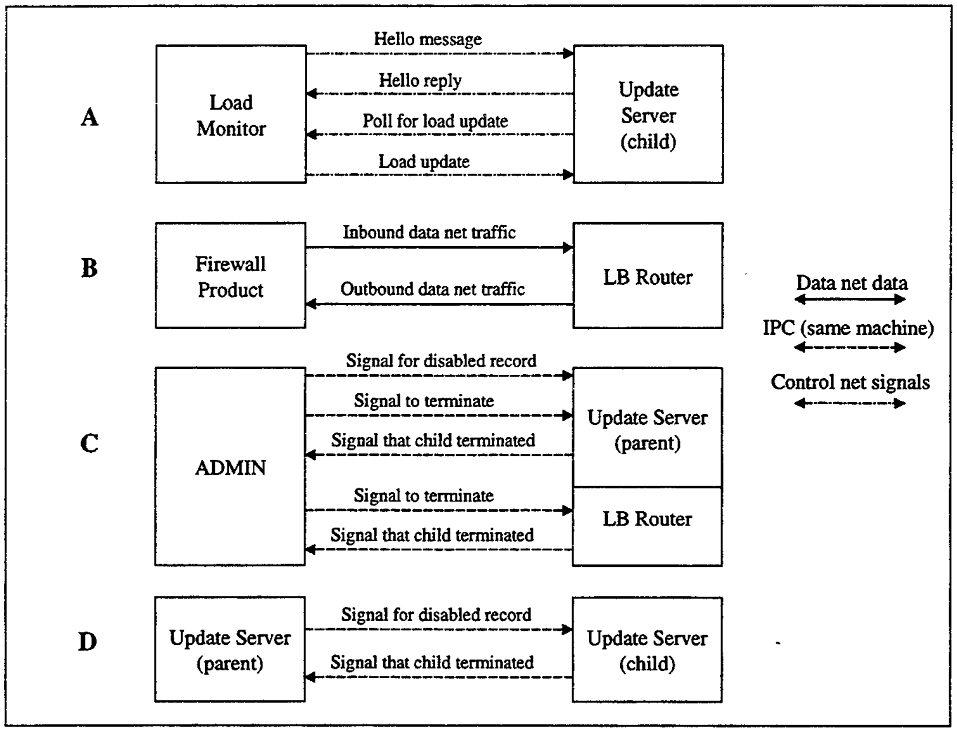

Figure 8 Inter-process communications – detailed view

A. FIREWALL

The requirements for each firewall machine are 1) to run firewall software, 2) to monitor load on the machine, and 3) to report the load to the controller. Each firewall machine's primary task is to run the commercial firewall software chosen for that machine by the security administrator. This thesis does not discuss firewall software in any detail because details are vendor specific. Also, the proposed architecture is designed to work regardless of the firewall product chosen. The firewall software mediates all network traffic between a protected network and the outside world. The proposed architecture also has each machine run a load monitor that calculates load on the machine and reports it to the controller periodically.

1. Load Monitor

The load monitor provides the following functions:

- Announces to the controller that the firewall machine is ready to operate
- Periodically samples receive queue length for each open connection and for connectionless network traffic
- Performs a health check of the firewall process
- Sends load updates to the controller

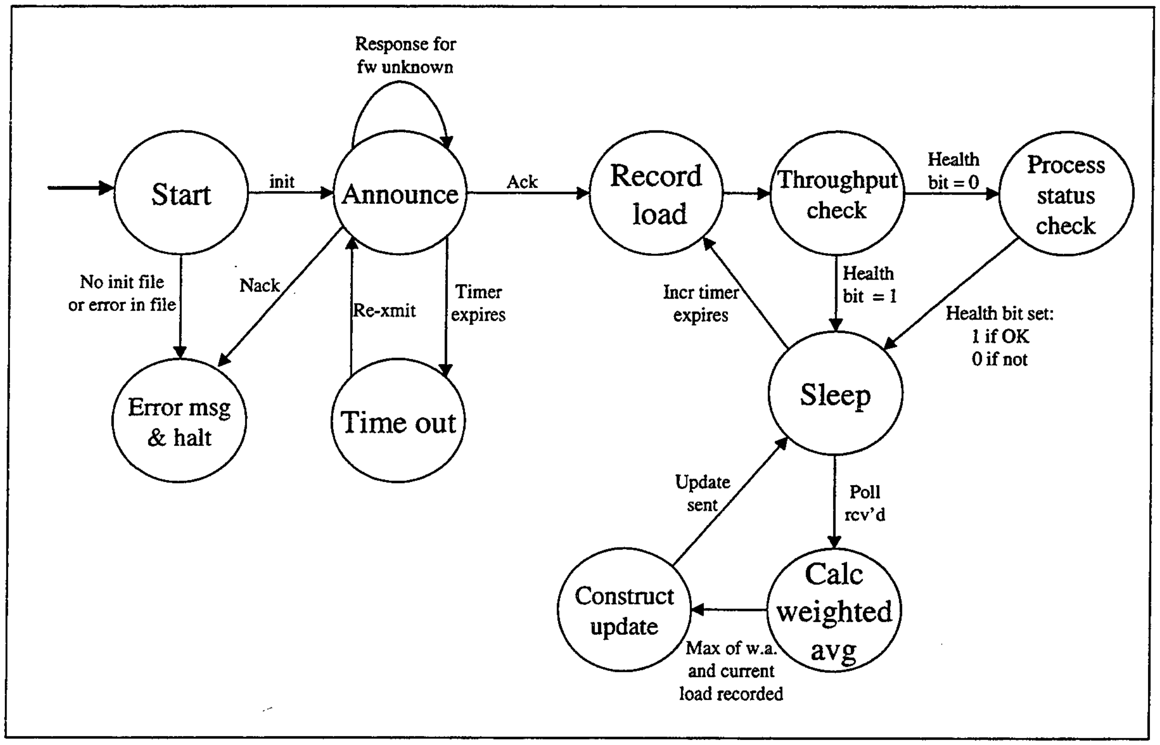

Figure 9 Load Monitor state diagram

Figure 9 shows the state diagram for the load monitor. The load monitor starts upon execution of file *loadmon.* It initializes by reading information such as the control network socket address of the controller and the load monitoring increment from its initialization file. It then opens a TCP socket to the controller on the control network, announces itself as being ready for data network traffic, and awaits an acknowledgement. Once it receives an acknowledgement, it begins monitoring load on the firewall machine. The socket is left open for further communications. If the load monitor receives a message saying that the firewall machine is unknown to the controller, then it displays an alert to the screen and continues announcing. The thought here is that the system administrator has started up the firewall machine prior to entering its record into the controller's firewall and they will add the record shortly. If it receives a negative acknowledgement, then it displays an alert and terminates.

The load monitor maintains a database of information on open TCP connections. Each record in the database consists of source and destination socket addresses as a connection identifier and a running weighted average of queue length associated with that connection. Load monitoring involves periodically recording receive queue length for each TCP connection and checking that the firewall process is processing data (i.e. checks for a healthy firewall process). Each time the load monitor samples a connection's queue length, it adds the result to the appropriate entry in the database. As it iterates through the open connections, it adds new connections to and deletes closed connections from the database. The load monitor calculates the weighted average using only samples taken since the last time it sent a load update to the controller. All history is purged between sending a load update and taking the next load sample.

The monitor determines total queue length by summing the greater of current queue length and weighted average queue length, since the last update, for each current TCP connection. In this way, the monitor errs conservatively when reporting load, the benefits of which were discussed earlier. It then adds the current total queue length for connectionless traffic to the previous sum. The load monitor's load equations are as follows:

$$Q_{wa} = \frac{\sum_{i=1}^{n} i * q_i}{n * (n+1) / 2}$$

where q_i = queue length of i^{th} sample since the last load update to the controller
n = number of samples
$i = i^{th}$ sample in chronological order and $i = 1$ is the oldest sample
Q_{wa} = weighted average queue length

$$L = Q_{non\text{-}conn} + \sum_{k=1}^{m} \max(q_k, Q_{k\text{-}wa})$$

where m = number of open connections,
$k = k^{th}$ open connection,
q_k = current queue length of k^{th} open connection
$Q_{k\text{-}wa}$ = weighted average queue length of k^{th} open connection
$Q_{non\text{-}conn}$ = current queue length of connectionless traffic

The load monitor checks the health of the firewall process using BSD Packet Filter (BPF), a service provided by the BSD/OS kernel. It uses BPF to verify that the firewall process is sending out processed traffic, provided there is traffic in the receive buffer to be processed. If this check fails, then the load monitor checks the process status to determine whether the firewall process exists or has died. If the firewall's health is satisfactory in the process status check, then the load monitor records a healthy status. If it is unsatisfactory, then the load monitor records an unhealthy status. These checks look for the case where the firewall machine is working but the firewall process has crashed or stalled. If the load monitor reports that the firewall is malfunctioning and the controller has stopped routing traffic to the machine, then the load monitor must be halted and restarted when the firewall process is restarted. This action causes the load monitor to re-announce itself as a functioning firewall.

When polled for a load update by the controller via the open control network socket, the load monitor packages the machine's load and firewall process's health information into a reporting format. It then sends the load update to the controller via the open socket.

B. CONTROLLER

The requirements for the controller are as follows:

- Balance network traffic load among the parallel firewall machines
- Provide a user interface for the system administrator

The controller runs on a single processor and provides the following functions:

- Balance outbound network traffic load among the firewall machines by routing new TCP connections and connectionless traffic to the appropriate machine
- Route network traffic for existing connections to the appropriate machine
- Collect load updates from the firewall load monitors for use in its load-balancing algorithm
- Remove a firewall machine from the distribution scheme upon indications that the machine is malfunctioning
- Provide a user interface to the system administrator

The controller software deals only with outbound network traffic. Inbound network traffic passes through the controller machine via the normal forwarding capability of BSD/OS. The controller software has several processes running to accomplish its functions: an administrative process, a load update process, and a load-balancing router process. The administrative process starts first and spawns the other processes.

1. Admin

The administrative process (ADMIN) starts upon execution of file *control*. Figure 10 shows the state diagram. ADMIN first initializes by reading an initialization file containing information such as the socket addresses of the firewall machines and the DNS, the processing capability of each firewall machine, and the time interval between load updates. Information about each potential firewall machine is placed in a firewall record in a firewall database. Operational parameters such as the polling increment are maintained separately.

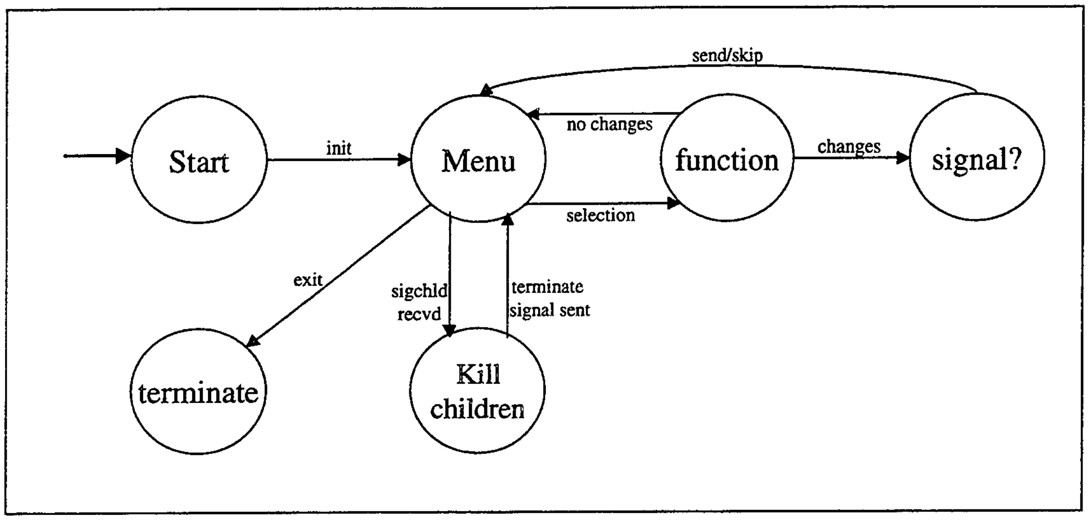

Figure 10 ADMIN process state diagram

Each firewall record in the firewall database consists of the following information:

- Handle – integer used for referencing the record from outside the database
- Firewall availability: ENABLED = the firewall may be placed in service,
  DISABLED = may not be placed in service
- Firewall status: ACTIVE = the firewall is included in load-balancing scheme,
  INACTIVE = not included
- Control net socket address - used to obtain load updates
- Processing capability - pre-determined by bench testing
- Latest load update - used by the load-balancing algorithm
- Data net IPv4 address - used to obtain Ethernet address
- Data net Ethernet hardware address - used by LB ROUTER to forward traffic
- Number of consecutive poor health reports and non-responses to polling – used to determine whether a firewall is malfunctioning and should be removed from the load-balancing scheme
- Distribution parameter - used by LB ROUTER to distribute new connections

The controller software design is such that data fields are controlled by only one process. That is, multiple processes may read them, but they are written by only one process. For example, ADMIN writes the handle, availability, control net socket address, processing capability, and data net IPv4 address fields.

After initialization, ADMIN displays a menu containing the following items:

- start controller – spawns the operational processes (i.e. update server and load-balancing router)
- stop controller – halts operation of the controller; kills the operational processes
- enable/disable a firewall record – allows firewall machines to be designated as ready for service or not from the controller's point of view
- add/delete/modify firewall records – edit information in disabled firewall records.
- view firewall record list
- view operational parameters
- change parameters – changes the parameters on the fly
- exit – kills the operational processes and terminates the administrator

The menu remains available to the user running the controller program until the exit option is selected or the ADMIN process is manually killed. ADMIN waits for a selection to be made from the menu. It also reacts to problems with its children. A SIGCHLD signal from the operating system tells ADMIN that a problem has occurred causing one of its children to terminate. It reacts by displaying an alert to the screen and halting controller operation by killing its other child process.

2. Update Server

The update server is an operational process spawned by ADMIN. UPDATE SERVER listens on the control net for firewall machines to announce their readiness for operation. When the server receives a new TCP connection on the designated port to which it is listening, it spawns an instance of itself: a child process to deal with the particular firewall. The parent process, whose state diagram is shown in Figure 11, continues to listen for other firewall announcements.

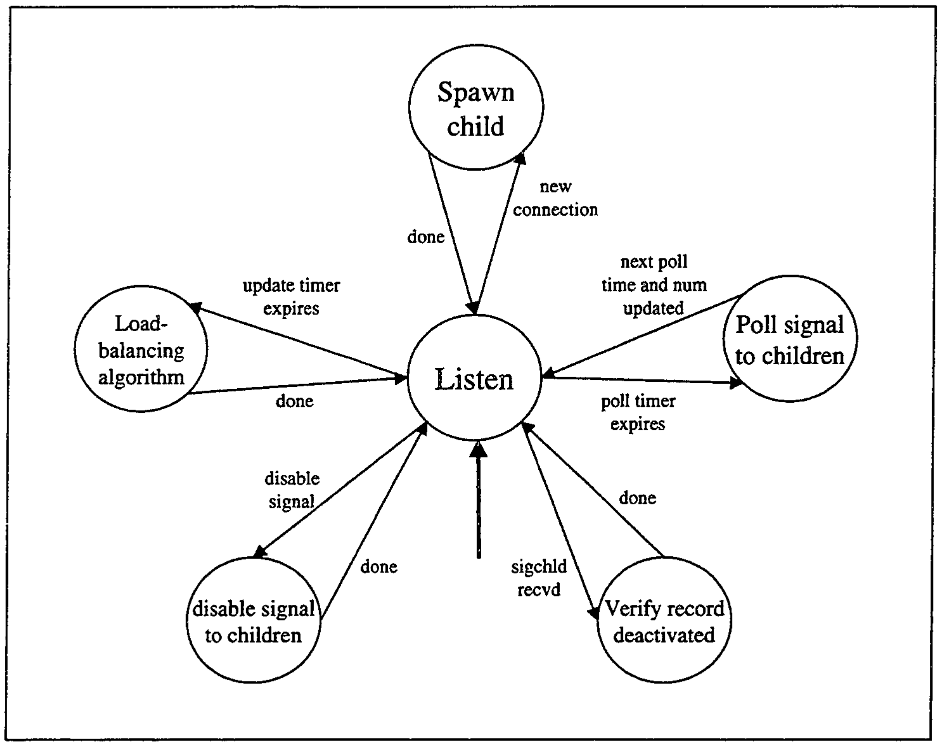

Figure 11 Parent Update Server state diagram

The parent also performs other tasks as follows:

1) Coordinates polling of the active firewalls
2) Relays signals to its children when ADMIN disables a record and signals the parent update server
3) Calls the load-balancing algorithm after load updates and after firewall activation/deactivation by its children
4) Verifies that a child update server's associated record is deactivated when the child terminates.

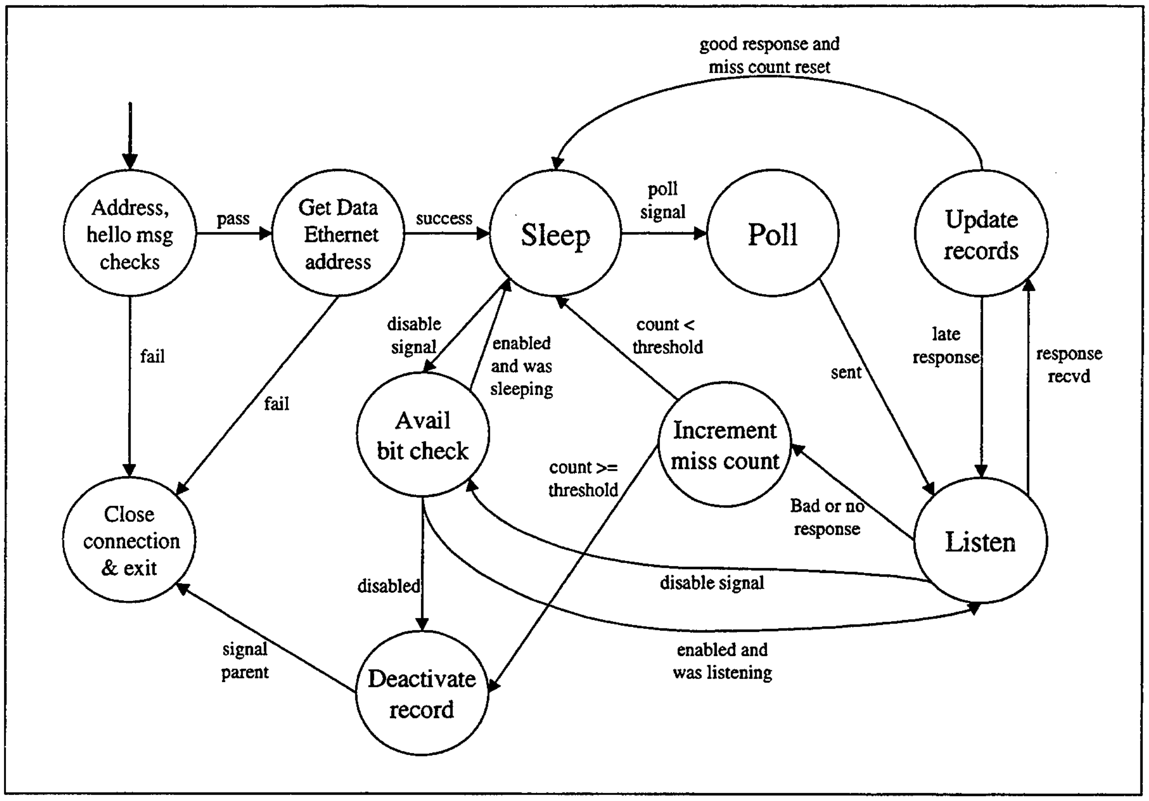

Figure 12 Child Update Server state diagram

The parent process coordinates polling for load updates so that all of the children poll at about the same time. The goal is for load updates to arrive at the controller within a reasonably tight time span of each other, allowing the parent to call the load-balancing algorithm shortly thereafter. The parent process relays signals from the ADMIN process because it knows who its children are. When the ADMIN process disables a record, the UPDATE SERVER children must check the availability field of their respective firewall records to determine whether they should deactivate the record and terminate the connection. ADMIN is able to signal the parent UPDATE SERVER. The parent relays the signal to its children.

The child UPDATE SERVER, whose state diagram is shown in Figure 12, verifies the announcing firewall machine by checking for an enabled entry in the firewall table. If the firewall is recognized by an enable entry, then the child UPDATE SERVER checks the "hello" message against its expected message. If the check passes, then the server sends an

Address Resolution Protocol (ARP) request on the data network to obtain the Ethernet address of the firewall's data net interface to the controller. It uses the IP address listed in the data net address field of the firewall record for the ARP request. If either the address check or the hello message check fails, or if the server is unable to obtain an Ethernet address for the firewall, then the child server terminates the connection and exits. It also notifies the user that the firewall announced itself and the reason for not accepting it as an operational firewall. A successful validation causes the server to acknowledge the firewall, activate the firewall record, and begin polling the firewall machine for load updates using the open TCP connection. The result of spawning child update servers is that the controller has an UPDATE SERVER for each active firewall.

Each child UPDATE SERVER records updates received from its respective firewall in the load field of the associated record of the firewall table. It also resets the associated misses field to zero with each answered poll. The misses field indicates the number of consecutive poor health reports and poll non-responses. If a firewall machine does not answer a load update poll or reports poor health, then the UPDATE SERVER increments the misses field of its associated firewall record. If a firewall's misses field exceeds a predetermined threshold value, then the UPDATE SERVER deactivates the associated firewall record and signals the parent UPDATE SERVER process.

The UPDATE SERVER's load-balancing algorithm determines the parameters to be used for statistical distribution of new connections. The load-balancing algorithm determines the proportion of new connections that each firewall should receive. It determines the distribution factor for each firewall as follows [Ref. 22]:

Load-balancing algorithm

- For each active firewall x

  $m_x = cap_x / cap_{tot}$

  $LF_x(t) = L_x(t) / L_{tot}$

  $k_x(t) = LF_x(t) - m_x$

  $df_x(t+1) = df_x(t) - r * k_x(t)$

where m_x = fraction of total firewall processing capability that x represents

cap_x = processing capability of firewall x

cap_{tot} = total processing capability of the firewall architecture

$LF_x(t)$ = load factor for firewall x

$L_x(t)$ = current load on firewall x

L_{tot} = total load on all active firewalls

$k_x(t)$ = correction factor for x

$df_x(t)$ = current distribution for x

$df_x(t+1)$ = corrected distribution factor for x

r = a relaxation factor, typically 0.7

- Set the distr field of the first active record to $df_1(t+1)$
- For each active record x, beginning with the second one, set the distr field to distr of the previous active record + $df_x(t+1)$

3. LB Router

The load-balancing router (LB ROUTER) is an operational process spawned by ADMIN. It routes outbound packets to the appropriate firewall machine based on the results of its scheduling algorithm and the contents of its routing table. Figure 13 shows the state diagram for LB ROUTER.

LB ROUTER runs in user, vice kernel, memory space. A result of this decision not to modify the BSD/OS kernel is that certain implementation methods and system configuration must be used. LB ROUTER should not have to process inbound network traffic for two reasons: 1) it does not need to balance load for incoming traffic and 2) the BSD/OS operating system has IP forwarding capability. So, the operating system on the controller machine should be configured to forward inbound traffic. However, since LB ROUTER is responsible for routing outbound traffic, the operating system should not forward outbound traffic. Outbound network traffic will not be addressed to the controller machine since the operating system will not pass it up through the protocol stack in the kernel to a user space program. However, LB ROUTER must receive the traffic so it can examine each packet in user space and route it to the appropriate firewall machine.

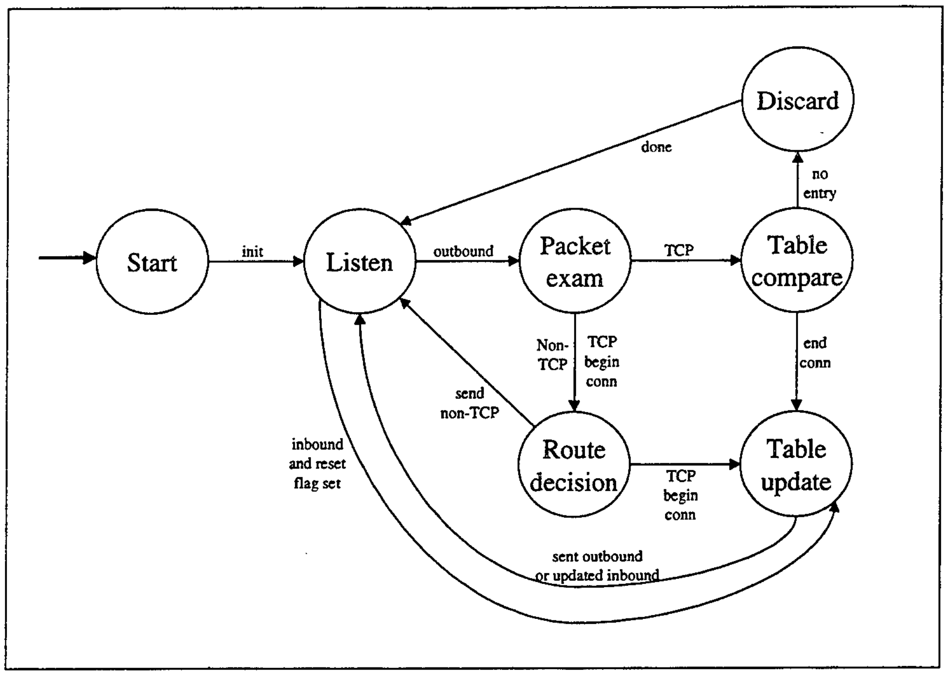

Figure 13 LB ROUTER state diagram

LB ROUTER uses BPF to intercept outbound Ethernet frames at the data link layer of the protocol stack (i.e. the network interface). BPF is a service in the BSD/OS kernel. The kernel sends a copy of each Ethernet frame received by the network interface to any open BPF devices before sending the frame up the protocol stack. LB ROUTER 1) examines each frame, and thus the IP packet, 2) makes additions or deletions needed to its routing table, 3) constructs a new Ethernet frame with the address of the appropriate firewall machine, and 4) transmits the frame on the Ethernet interface to the firewalls using BPF.

LB ROUTER's routing table contains records for all open TCP connections. Each record consists of the following information:

- source socket address
- destination socket address
- data net Ethernet address of the firewall mediating the connection

LB ROUTER examines each intercepted packet to determine whether it establishes a new connection. If the packet establishes a new connection, LB ROUTER calls its scheduling routine to determine which firewall should get the new connection. It then records the connection in its routing table and generates a new data link layer frame for routing. If the packet is connectionless traffic, LB ROUTER calls its scheduling routine and generates a new Ethernet frame. If the packet is connection-oriented and not for a new connection, LB ROUTER sends the packet to the firewall machine recorded in its routing table. If the packet is TCP but belongs to no existing connection, then it is discarded.

IV. TEST NETWORK CONFIGURATION

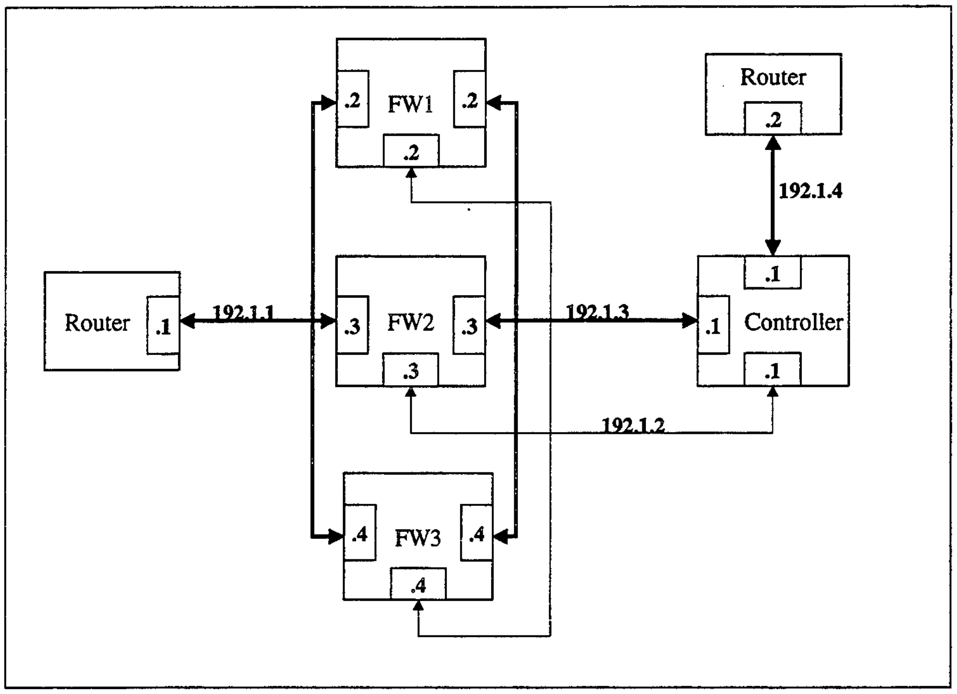

Figure 14 Test network addressing scheme

This chapter discusses issues involved in building a test network for the software design described in Chapter 3. The intent is to aid system testers and to provide some guidance for creating reproducible tests. Figure 14 shows an example test network. The network consists of three firewall machines, a controller machine, and two machines representing the interior and exterior routers in the physical architecture. For testing purposes, these two endpoint machines should not be routers, but rather should be machines that will generate and receive test traffic passing through the firewalls and controller. For simplicity and to reduce cost, the test network uses three 4-port Ethernet hubs and a direct connect cable rather than switches as shown in Figures 1 and 5. The direct connect cable connects the interior router to the controller and one hub connects each of the other three domains.

The firewall machines and controller run the BSD/OS operating system and have three network interface cards (NIC) represented in Figure 14 by rectangles within each machine. BSD/OS 3.1 is fairly restrictive in the selection of NICs that it supports so checking the hardware support list is essential before building the test network. SPAWAR typically uses the 3COM 3C905B Fast EtherLink and the Intel Pro/100+ Ethernet cards. The 3COM card requires two patches (numbers 21 and 28) for BSD/OS 3.1 to be installed before the operating system will properly support it.

The addressing scheme of the network uses four Class C domains. The 192.x.x.x addresses are not assigned to any particular Internet user and so are typically seen in test networks [Ref. 21]. Figure 14 shows the IPv4 address associated with each network domain (192.1.x) and the unit address within each domain (.x) for the network interfaces.

This thesis does not cover BSD/OS system administration. However, most UNIX administration texts explain how to configure network interfaces and to manually enter routes into a machine's routing table. Manual route entry is required to enable the controller's operating system to forward inbound traffic. Once the network interfaces are configured and manual routes entered, TIS Gauntlet firewall software is installed on the firewall machines.

Initialization files tailored to the test network interfaces must be written for the firewall machines and for the controller. The firewall initialization files tell each firewall the control net address of the controller and what port number it should use to communicate with the controller (192.1.2.1 in Figure 14). They also provide operational parameters such as load sampling rate. The controller's initialization file tells it the network interface designated as the control net and the port number on which it should listen for firewalls to announce themselves on the control net as ready to receive data network traffic. The file also contains operational parameters such as load polling increment.

Test programs should generate traffic originating at the interior endpoint of the network (192.1.4.2) and destined for the exterior endpoint (192.1.1.1). These programs should generate varying volumes of traffic and numbers of simultaneous connections to stress the logical operation and capacity of the architecture.

V. DISCUSSION AND CONCLUSIONS

This section discusses observations made during the software design, follow-on work, and future research topics. It ends with conclusions on the work presented here.

A. LESSONS LEARNED FROM SOFTWARE DESIGN

The software design underwent a series of changes during development. This section describes some of the lessons learned and changes made. Some changes were simply refinements to aid ease of understanding the problem and proposed solution while others were needed to avoid potentially serious problems, especially when considering a multiple processor controller platform.

The first decision that must be made in software design is the architectural style to be used. The merits of a modular, layered design have long been known. However, this approach to design once again showed itself to be a superior one. Other architectural styles were initially considered for the design. However, modularity and layering eventually emerged as the appropriate methodology. While one cannot be provably certain that no flaws exist in a system, modularity and layering provide a means of eliminating many potential flaws during the design phase of a project rather than the implementation and testing phases.

The design process proved to be an iterative one. Modularity and layering lend themselves to iterative design, with refinements being made at each stage of the process. One problem discovered as a result of the iterative design review was a situation in which a child update server process could terminate due to an error without deactivating the firewall record for which it was responsible. This situation would cause the controller to route network traffic to the firewall with no knowledge of its load or whether it was even operating.

Initially, a linked list was considered for the firewall database. However, concurrency problems arose unless mutual exclusion techniques were used to restrict rather severely the various processes' access to the shared data structure. Additional analysis indicated that a completely general solution was not needed. Linked lists were unnecessary given the realistic limitation on the number of firewall machines that would be used. So, the database was designed as a static array, thus avoiding the concurrency problems encountered with linked lists.

Another refinement of the system was encapsulation of data structures within source files rather than providing global access to the shared structures. Handles to records in databases were used, where practical, vice passing pointers around, a riskier prospect in terms of breaking encapsulation.

The choice of inter-process communication (IPC) method was also a concern. The appropriate IPC must be an integral part of a multi-process design scheme. The selected method should be synergistic with the control flow of a design. Several IPC options were considered before deciding upon signals. The need for asynchronous, near real-time communications between the controller's processes made signals an appealing choice.

As a final note, the level of knowledge and skill required for proper system design is formidable. Much research and many experiments with UNIX networking and IPC were conducted to gain a sufficient knowledge base to be able to begin design work. Designing multi-threaded or multi-process systems is non-trivial. Given that implementation and testing are not complete, the design presented in this thesis is sure to have future refinements or changes.

B. FOLLOW-ON WORK AND FUTURE RESEARCH

The following list discusses follow-on work and future research areas related to the load-balancing firewall architecture presented in this thesis:

1. Continue implementation of the proposed physical architecture and the software architecture for outbound network traffic. The physical architecture, for testing purposes, is relatively easy to setup in a lab environment. The software libraries for the various data structures discussed earlier have been implemented with the exception of the lt_update_table() function, which involves the netstat program. The netstat source

code and programming interface are being reviewed for use. Control logic for the various processes is in progress. Remaining work includes writing signal handlers for IPC in the controller, writing code involving BPF for both the controller and the firewall load monitor, and writing the LBrouter() function.

2. Design and implement a standard bench testing procedure for obtaining values for the baseline processing capability of firewall machines. This procedure should take into account the extension of system capabilities to include adaptive scheduling according to network load and traffic composition. With a system that adapts to network traffic composition, the bench testing must provide a baseline throughput value for each protocol admitted through the firewall.
3. Thoroughly test the performance of the system. The testing should be done in stages: first testing each layer of the design, then testing groups of adjacent layers, and finally testing the system as a whole. System testing should be conducted using various network loads, with various traffic compositions, and with various settings of the tunable system variables (i.e. adjusting relaxation factor, polling interval, and sampling interval).
4. Design and implement the load-balancing DNS software for balancing inbound network traffic. A load-balancing DNS program exists in the Perl scripting language [Ref. 23]. This program could be converted to C and used as a building block for the DNS. The design of the DNS program should take into account both connected and disconnected modes of operation. Connected mode involves a one-way fiber system connecting the DNS to the control net, allowing the server to receive control signals. Disconnected mode has the DNS operating essentially independently. The controller may need to send basic firewall machine status reports (i.e. a new machine is up or a machine is down) through the firewall to the DNS. This decision would allow the DNS to obtain the information it needs on operational firewall machines with a minimum of control signals being passed over the data net. Another option is to have the DNS detect which firewall machines are operational by monitoring network traffic passing between the exterior router and the parallel firewall machines.

5. Include an audit module on each machine. This module will timestamp all alerts, display them to the screen, write them to a log file, and coordinate switching among logs files as necessary. The current design requires each function to display its own alerts and no logging is done.
6. Assuming no one-way fiber system is used for connecting the DNS to the control net, test the option of having the DNS monitor the data going to each firewall machine. Since the DNS normally processes only domain name requests, its work load is light. Having it monitor data passing between the exterior router and the parallel firewall machines could provide it with load information for each firewall machine. It could use this information in its load-balancing algorithm. In this case, the DNS would perform a load monitoring function similar to that of the firewall load monitors. However, the DNS would map the amount of data passing across the network, rather than queue length, to TCP connections.
7. Move the controller's routing process to kernel space for improved performance. The current software design places the controller's routing process in user memory space. As a result, the program uses the kernel's BSD Packet Filter service to obtain outbound network traffic and to forward the traffic to a different network interface. The operating system kernel must copy the desired information from kernel memory space to user memory space for the user program to access it. The revised information must then be copied back to kernel space from user space so the kernel can transmit it on the appropriate network interface. Moving the necessary functions from user to kernel memory space will eliminate the overhead associated with these copy routines. Slight modifications to the operating system's normal IP forwarding routine should allow the load-balancing routing functions to be called as the kernel decides which default gateway to forward an IP packet.
8. Extend administrative functions to support remote administration. Having separate control and data nets facilitates remote administration via the control net. Adding a remote access server to the control net would enable administrators to access the components of the system via dial-in or dedicated phone line connections. This

extension to remote administration may necessitate having user and machine authentication between components.

9. Transition to adaptive load sampling and scheduling to account for situations such as wildly fluctuating network traffic patterns (where increasing sampling rate would be appropriate) and changing traffic compositions by protocol (where weight factors for HTTP, SMTP, FTP, etc., may need to be adjusted in the load-balancing algorithm). Adaptive behavior in the system may require the addition of a machine dedicated to monitoring traffic load and composition to the architecture. This machine would be best situated with a tap into the network between the interior router and the controller so it could see all network traffic passing through the firewall architecture. The controller, firewall machines, and possibly the DNS, could use the traffic monitor's results to adapt their behavior appropriately. Investigating the possibility of having the DNS monitor traffic was mentioned earlier. However, the DNS could not convey the results to the other architectural components for two reasons: the DNS cannot transmit on the control net and the other components are designed to ignore control signals arriving on the data net.

10. Provide support for both IPv4 and IPv6 addresses and for hardware formats other than Ethernet. This area may include modifying the system for use in ATM networks.

C. CONCLUSIONS

The proposed load-balancing firewall architecture provides a solution to the limitations and potential performance bottleneck associated with the currently used parallel firewall architecture. These characteristics are obtained with no effect on the commercial products (i.e. the routers, the firewall software product, and the operating system) already in use in the current firewall architecture. Specific software has been designed here and must be implemented to control the proposed system, as opposed to being purchased off-the-shelf. However, the impact in terms of financial, administrative, and manpower resources required on the part of the Navy is negligible compared to specialized software product development.

The modular layered software design has several advantages over other design styles. It orders dependencies in the system, limiting the set of possible conditions in any given layer and thus allowing comprehensive testing in stages. It aids understanding of the system for implementers and for follow-on researchers. It also makes the associated code easily verifiable for correctness. Much of the implementation has been completed but is left for future testing and publication.

The issue of processing overhead, and its cost in data throughput, associated with load sampling and load balancing (i.e. routing in the controller) seems to be the most significant issue in proving the system's worth. Future in-depth testing will reveal these answers. The software system provides an administrative interface for performance tuning. The anticipated issue is not in whether the system is sufficient, but rather is in finding the right balance between sampling frequency for an acceptable estimate of load and performance degradation due to processing overhead.

This thesis provides a foundation upon which to build a parallel firewall system that has marked advantages over those currently in use. The design will yield a scalable product that permits each firewall to be multi-protocol, provides an accurate assessment of firewall load, and is responsive to changing system and network conditions via feedback control. It will accomplish these tasks while maintaining firewall product, router, and firewall machine independence, while isolating the system from external mechanisms, and while isolating control signals from network data. Future work will improve and extend the capabilities of the system, increase its efficiency, and provide extensive test data and analysis as a guide to new research.

APPENDIX A. SOFTWARE REQUIREMENTS SPECIFICATION

1.0 Introduction

1.1 Purpose

The purpose of this document is to define the software requirements for a load-balancing firewall system to be used for multiple parallel firewall machines as described in Reference 1. This document describes the load-balancing mechanism for outbound traffic; that is, network traffic initiating from inside the protected enclave and going to the outside world via the firewall.

1.2 Scope

The software developed must meet several requirements:

- Run on BSD/OS 3.1 operating system
- Support TIS Gauntlet firewall product
- Balance network load (both connection-oriented and connectionless) across all machines
- Provide 100 Mbps data rate to support Fast Ethernet networks with ability to increase throughput later

Additional features to be supported by the developer are as follows:

- Firewall product independence
- Robustness to account for a firewall machine malfunctioning
- Scalability to allow greater firewall throughput by adding more machines in parallel

1.3 Glossary of Definitions and Abbreviations

BSD/OS	Berkeley Software Design Operating System

Connectionless traffic	Network traffic using an OSI Transport Layer protocol that simply transmits data on the network without regard to establishing a network connection between source and destination machines verifying receipt of the data
Connection-oriented traffic	Network traffic using an OSI Transport Layer protocol that establishes an open network connection between source and destination machines prior to sending data, and that verifies receipt of the data prior to closing the connection
COTS	Commercial Off-The-Shelf
DNS	Domain name server
IPv4	Internet Protocol version 4
Mbps	Mmega-bits per second
Network load, Network traffic	The amount of data transiting the network
TCP	Transmission Control Protocol
Throughput	The amount of network traffic being processed through a component or architecture
TIS	Trusted Information Systems

2.0 General Characteristics

2.1 Introduction

The load-balancing firewall software system will be a distributed system intended to allow a heterogeneous mix of firewall machines, running various COTS firewall products, to work together in a parallel architecture. The software will distribute network traffic, both connection-oriented and connectionless, to the firewall machines such that each machine processes only a fraction of total network load proportional to its processing capability.

The physical architecture of the system (see Figure 1) involves 1) interior and exterior COTS routers, 2) multiple firewall machines connected in parallel, 3) an intermediate load-balancing machine (known as the controller) located between the interior router and the firewall machines, and 4) a load-balancing DNS located between the exterior router and the firewalls. The software system will be distributed across all physical components in the architecture. The software components will communicate among themselves the information necessary to perform activities listed in section 2.3.

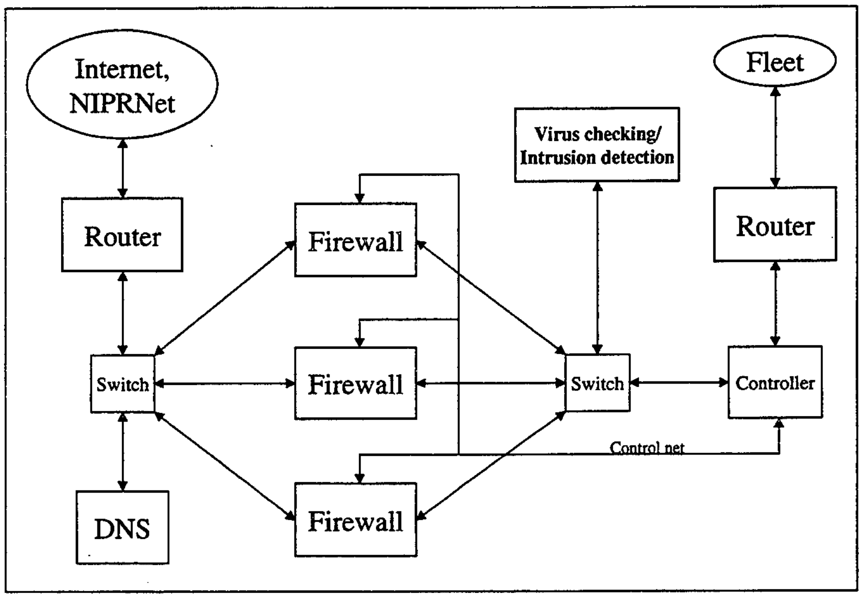

Figure 1 Physical architecture

This document addresses only with outbound network traffic. The DNS will balance inbound network traffic and the controller will balance outbound network traffic. The software components on each firewall machine will monitor load on that machine and report it to the component in which the load-balancing algorithm resides. The controller will distribute outbound network traffic according to the load-balancing plan generated from firewall machine load updates. Inbound network traffic will pass through the controller with no action on the part of the software being developed here.

2.2 Product Perspective

The software system is to operate as a stand-alone package. However, the operating system and hardware on each machine must be configured properly for the whole firewall architecture to work correctly.

2.3 Product Activity

The software will perform the following activities:

- Read system initialization data from a pre-existing file
- Provide an interactive administrative interface for changing system parameters
- Monitor load on each operational firewall machine
- Perform operational checks to verify that each firewall machine is operating and that the firewall product on each firewall machine has not stopped running
- Balance network load, both connection-oriented and connectionless, among operational firewall machines in the parallel firewall scheme

2.4 User Characteristics

1. It is assumed that users will have some experience as UNIX system administrators or technicians and will be familiar with installing software on and configuring UNIX platforms.
2. It is assumed that system technicians performing bench testing of firewall systems will be thoroughly familiar with software installation, UNIX system configuration, and network communications.

2.5 General Constraints

1. The source code will be written in the C programming language.
2. The program will be designed to run on UNIX operating systems, specifically BSD/OS 3.1.

3. The program will support IPv4 protocol addresses and Ethernet hardware format addresses.
4. The program will balance TCP connection-oriented network traffic and connectionless traffic.

2.6 Assumptions

1. Proper bench testing will be conducted to determine the baseline processing capability of each firewall machine. Each machine will be tested in the configuration (both hardware and software) in which it will be used in the parallel architecture.
2. Interactive administrative interfaces will be UNIX foreground processes that must remain active for the software on that physical component to function properly.
3. The physical architecture will be similar to that shown in Figure 1 and described in Reference 1. The network medium will use Ethernet hardware format. The hardware components (e.g. network interface cards) and their configuration will support BSD/OS 3.1 and thus the software being developed here.

3.0 Specific Requirements

3.1 Functional Requirements

3.1.1 Load Monitoring

1. Introduction

   Load will be monitored for each firewall machine.

2. Processing

   Receive queue length will be the metric used to measure load. A weighted average of load will be maintained between updates to the controller. The greater of current queue length and historical weighted average of queue length will be reported as the load.

3.1.2 Load Balancing

1. Introduction

   The software system will act to balance network traffic load among firewall machines.

2. Processing

   Network load will be distributed based on the known processing capability of each firewall machine, the reported load for each machine, and the previous state of the load distribution process.

3.1.3 Routing

1. Introduction

   The controller will route outbound network traffic to the appropriate firewall machine as determined by the load-balancing scheme.

2. Processing

   Once a new connection is established through the firewall, all traffic associated with that connection must go to the same firewall machine. Connectionless traffic may go to any firewall machine.

3.1.4 Data Entry

1. Introduction

   Control parameters and firewall information can be viewed and entered interactively at the keyboard of the controller.

2. Processing

   The user will be able to view and modify the controller's firewall information and operational parameters whose adjustment can tune the program's performance.

3.2 External Interface Requirements

3.2.1 User Interfaces

The software will use menus to allow users to view and modify data described in section 3.1.4. Menu options will be selected via the keyboard.

3.2.2 Hardware Interfaces

The software will operate with multiple Ethernet network interface cards installed per machine.

3.2.3 Software Interfaces

The software will be designed to run under UNIX operating systems, but is required to be tested only under BSD/OS.

3.3 Performance Requirements

1. Status messages will provide the user with immediate feedback on events occurring in response to the user's menu selections. The software will also immediately notify the administrator of runtime errors such as being unable to open a socket, an unrecognized firewall machine communicating with the controller, and a firewall record being deactivated by the controller because the firewall machine does not respond to load update polls. These notifications will be made to the terminal of the machine on which the error occurred and may also be made to a log file.
2. The software system will provide a data throughput capability for the parallel firewall architecture greater than that of the routers and of the network.

3.4 Design Constraints

1. No modifications may be made to commercial software products existing on the physical components (e.g. the commercial router software and the BSD/OS kernel).
2. Code must be written in C, but should be modular and carefully layered.

References

1. William Joyner, "Design of a Load-Balancing Architecture for Parallel Firewalls," Naval Postgraduate School, Monterey, California, March 1999.

APPENDIX B. SOFTWARE DESIGN SPECIFICATION

1. Introduction

Purpose

This document describes the software design for a load-balancing firewall system used for multiple, parallel firewall machines. Specifically, it describes the load-balancing software architecture for outbound traffic; that is, network traffic initiating from inside a protected enclave and going to the outside world via the firewall. The document is intended for use as a roadmap for software designers making future modifications and as direction for source code implementers.

Scope

The software design presented here is intended for use in physical architectures of two or more firewall machines as described in Reference 1. The design will meet the following requirements:

- Run on BSD/OS 3.1 operating system
- Balance network load (both connection-oriented and connectionless) across all firewall machines
- Provide 100 Mbps data rate to support Fast Ethernet networks with ability to increase throughput later
- Firewall product independence
- Router independence
- Robustness to account for a firewall machine malfunctioning
- Scalability to allow greater firewall throughput by adding more machines in parallel
- Support for IPv4 and Ethernet hardware format addresses

Summary of Contents

The remainder of this document covers several topics, starting at a high level of design and working down to detailed explanations of function prototypes. First, the document provides a high level overview of the system (Section 2) and discusses high-level design decisions made based on assumptions, dependencies, and constraints (Section 3). It then discusses architectural strategies (Section 4) used in the design and presents the software architecture from a high-level perspective (Section 5). Finally, it presents a detailed discussion of the design, providing relevant design decisions along the way (Section 6).

Glossary

API	Application programming interface
BSD/OS	Berkeley Software Design Operating System
BPF	BSD Packet Filter
Connectionless traffic	Network traffic using an OSI Transport Layer protocol that simply transmits data on the network without regard to establishing a network connection between source and destination machines verifying receipt of the data
Connection-oriented traffic	Network traffic using an OSI Transport Layer protocol that establishes an open network connection between source and destination machines prior to sending data, and that verifies receipt of the data prior to closing the connection
COTS	Commercial Off-The-Shelf
DNS	Domain name server
IPC	Inter-process communication
IPv4	Internet Protocol version 4
Mbps	Mega-bits per second
Network load, Network traffic	The amount of data transiting the network
TCP	Transmission Control Protocol

Throughput	The amount of network traffic being processed through a component or architecture
TIS	Trusted Information Systems

2. System Overview

The load-balancing firewall software system is a distributed system intended to allow a heterogeneous mix of firewall machines, running various COTS firewall products, to work together in a parallel architecture. The software distributes network traffic, both connection-oriented and connectionless, to the firewall machines such that each machine processes only a fraction of total network load proportional to its processing capability.

The physical architecture of the system [Ref. 1] involves 1) interior and exterior COTS routers, 2) multiple firewall machines connected in parallel, 3) an intermediate load-balancing machine (known as the controller) located between the interior router and the firewall machines, and 4) a load-balancing DNS located between the exterior router and the firewalls. The software system is distributed across all physical components in the architecture. The software components communicate among themselves the information necessary to monitor and balance network traffic in the parallel firewall architecture.

This design document pertains only to balancing outbound network traffic. Thus, it presents the software design for components residing on the firewall machines and the controller. Since the DNS balances inbound network traffic, its software components are not addressed. However, the design takes into account the future incorporation of the DNS software components into the overall architecture.

The software components on each firewall machine monitor load on that machine. They report load values to the controller periodically. The controller's software components generate a load-balancing plan based on the load reports from each firewall machine. The controller routes outbound network traffic to the appropriate firewall machine according to the load-balancing plan. Inbound network traffic is intended to pass through the controller with little or no action on the part of the software designed here.

The software on each machine reads an initialization file at startup. The initialization file contains addresses of machines with which the program will communicate and operational parameters for the program. The controller's software also provides a user interface to allow the user to view and modify the load-balancing system information on the fly.

3. Design Considerations

Assumptions and Dependencies

1. Proper bench testing will be conducted to determine the baseline processing capability of each firewall machine. Each machine will be tested in the configuration (both hardware and software) in which it will be used in the parallel architecture.
2. Interactive user interfaces will be UNIX foreground processes that must remain active for the software on that physical component to function properly.

General Constraints

1. The source code must be written in the C programming language.
2. The program must be designed to run on UNIX operating systems, specifically BSD/OS 3.1.
3. The program must support IPv4 protocol addresses and Ethernet hardware format addresses.
4. The program must balance TCP connection-oriented network traffic and connectionless traffic. All non-TCP traffic will be treated as connectionless.
5. The program must operate on the physical architecture described in Reference 1.
6. No modifications may be made to commercial software products existing on the physical components (e.g. the commercial router software and the BSD/OS kernel).

7. The design must be modularized and layered. These characteristics increase ease of understanding of the code and facilitate verifying the code's correctness. References 2 and 3 are classic papers discussing modularization and layering.

Guidelines

1. The design should be as simple and straightforward as is practicable.
2. The design should focus on speed and efficiency without compromising good design and programming practices. Time-consuming evolutions, such as context switching and busy waiting, should be minimized where practicable.
3. Communications between processes, regardless of whether the processes are on the same or different machines, should be kept to reasonable amounts. High communication volume brings latency and processing overhead to the program.

4. Architectural Strategies

1. The C programming language was chosen because it lends itself to programming with UNIX system APIs.
2. The controller polls the firewall load monitors for load updates. This decision allows the controller to receive load updates in a coordinated manner rather than piecemeal, without having to synchronize clocks in the distributed system. Not having to synchronize clocks in the distributed system saves much processing overhead and latency due to message passing. Since the controller asks for load updates from all firewalls simultaneously, it expects to receive all of the load updates at nearly the same time. It can then run the load-balancing algorithm once for that update iteration. Allowing the firewalls to send load updates asynchronously would either cause the controller to run its load-balancing algorithm more frequently or degrade the accuracy of the load-balancing calculations by having the controller run the algorithm at random times.
3. A central data structure is used with handles to relevant parts of the structure being passed to functions whenever practicable. Passing handles to a single

instance of data eliminates potential data consistency problems that can occur in multi-process programs if multiple instances of data exist.

4. Whenever possible, data fields and the functions that operate on them are partitioned among the multiple running processes such that minimum potential conflict exists that would require mutual exclusion measures such as locks or semaphores.
5. A modular approach to design dictates creating modules by functional responsibility [Ref. 2]. A module is composed of one or more functions with related functional responsibility and perhaps an associated data type. Decomposing a software system into modules makes it easier to understand, to change and maintain (i.e. design changes), and to develop in a piecemeal, independent manner. A layered approach dictates that designs have multiple layers of functions [Ref. 3]. Higher level functions have more abstraction in their tasks than lower level functions. They generally call lower level functions to perform more specific, less abstract tasks for them. Ideally, the entire layered architecture is only downward dependent (i.e. no function calls a function at an equal or greater level than itself). This characteristic creates a partial order among functions, greatly improving ease of understanding and making code verification for correctness far easier.
6. Databases should generally be made private by instantiating them in their modules' source files. This action encapsulates the databases. The methods for a particular database will mediate access to the database for client programs.

5. System Architecture

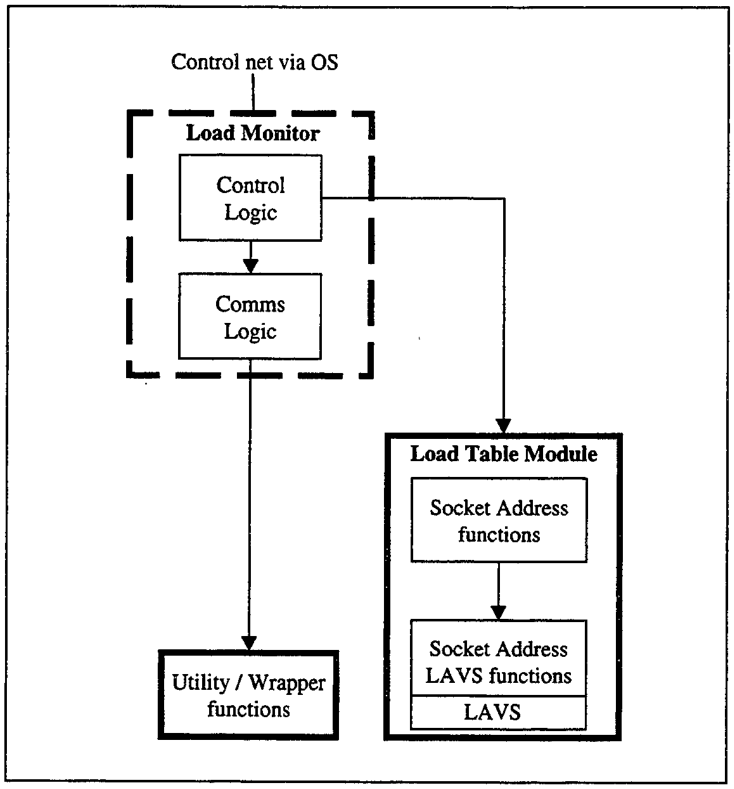

Figure 1 Firewall Load Monitor Architecture

The software system architecture is distributed across each component in the physical architecture. One or more processes related to the load-balancing architecture runs on each physical component. The separation of functional responsibilities that should logically run in parallel determines the number of processes on a component.

Each firewall machine runs a load monitoring process. Figure 1 shows this process's architecture. The decision to place load monitoring on each firewall machine is discussed in Reference 1. The load monitor announces itself to the controller at start-up and then monitors load on the firewall machine. When it receives an update poll from the controller, it answers with its latest load value.

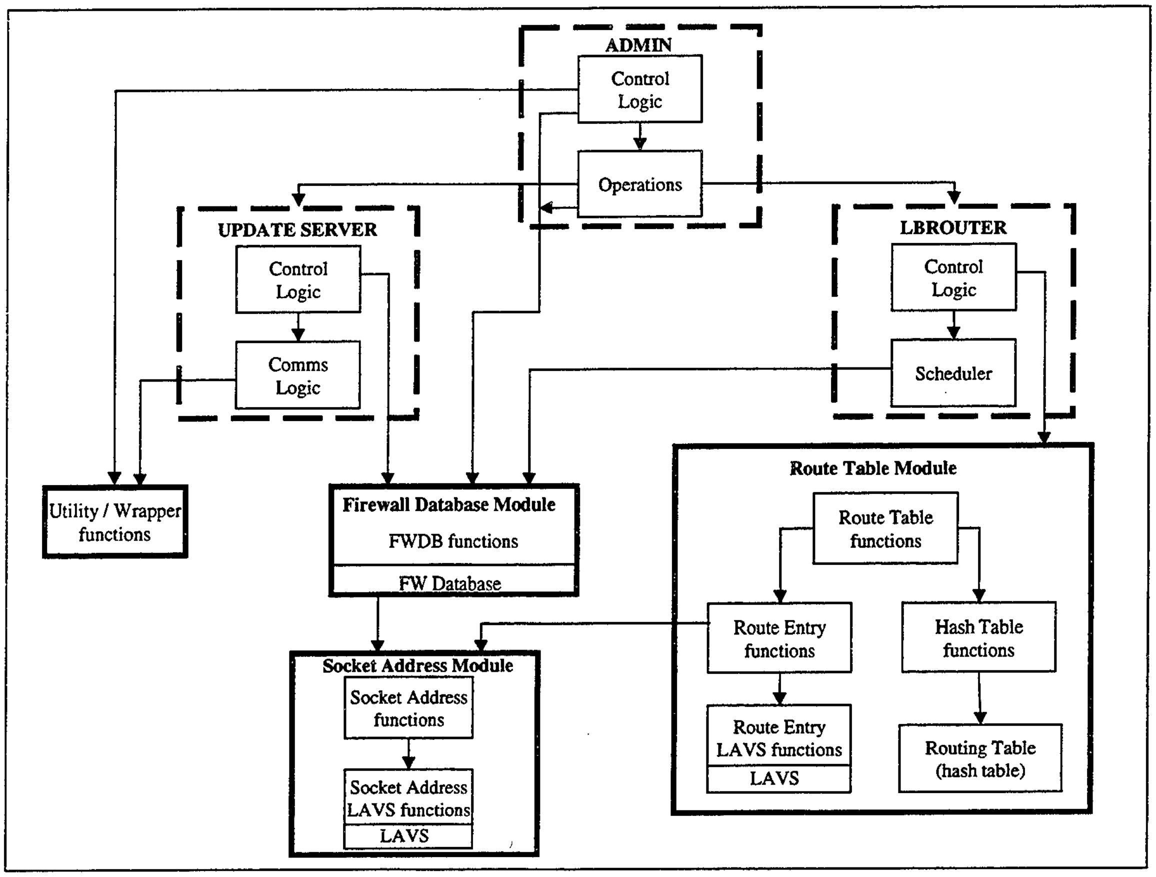

Figure 2 Controller Software Architecture

The controller runs three processes: one for interacting with the firewall load monitors to get load updates, one to route outbound network traffic to the appropriate firewall machine, and one to provide a user interface. Figure 2 shows the controller's software architecture. Chapter 3 of Reference 1 discusses the functioning of each controller process and provides each process's state diagram.

6. Detailed System Design

This section describes the functions and data structures that make up the processes mentioned previously. It lists BSD/OS system structures used in the design and describes the functions generated to use them. It also defines data types generated for this design and their related operations. Finally, it describes the composition of each process.

Lists of available space (LAVS) are used throughout the design to improve performance. A list of available space contains unused instances of the applicable data type or structure. Its use minimizes costly operating system calls to allocate/deallocate memory from the heap for dynamically allocated data structures. When a new instance of the data structure contained in the LAVS is needed, the client program first calls a LAVS function to determine whether any unused instances of the data structure exist. If an instance is available, then it will be used vice calling the operating system to allocate new memory space for an instance of the data structure. When data structures are deleted, they are placed in the LAVS rather than being deallocated by the operating system. A LAVS is intended to be private. That is, it is instantiated with file scope within a module's source file (*.c file) for use by the module's functions, but it may not be accessed via the associated header file.

The following definitions and BSD/OS system data types and structures are used in the design:

```
#define TRUE          1
#define FALSE         0
```

u_char	unsigned character
u_short	unsigned short integer
u_int	unsigned integer
u_long	unsigned long integer

6.1. Utilities Module

This section defines general utility functions and wrapper functions for BSD/OS system function calls used in the software design. The wrappers include error handling and are tailored for IPv4 TCP socket communications.

Function prototype	FILE *Fopen(const char *name, const char *type);
Description	Wrapper function for the UNIX fopen() function. It takes the filename of the desired file to open and the mode in which to open it (e.g. r, w, etc.) as arguments and returns an open file descriptor. If an error occurs while attempting to open the file, the function displays an error message and exits the program.
Input	name – string indicating the name of the file to be opened type – string indicating the permissions with which to open the file

	Note that these are the same arguments used by the system call.
Return	Open file descriptor
Constraints	None
Pseudo code	Call fopen() If error occurs Display error message Exit Return open file descriptor

Function prototype	int Socket();
Description	Wrapper function for the UNIX socket() function. It opens a TCP socket and returns the socket descriptor. If an error occurs while opening the socket, the function displays an error message and exits the program.
Input	void
Return	Socket descriptor
Constraints	None
Pseudo code	Call socket() If error occurs Display error message Exit Return open socket descriptor

Function prototype	void Bind(int sockfd, SOCKAI *addr);
Description	Wrapper for the UNIX bind() function. It binds a local protocol address to a socket. If an error occurs, it displays an error message and exits the program.
Input	sockfd – an open socket descriptor addr – handle to the local address to which to bind the socket
Return	void
Constraints	An address may be bound to only one open socket descriptor at any given time. The address given for binding must be a valid interface address on the particular machine running the program.

Pseudo code	Call bind()
	If error occurs
	  Display error message
	  Exit

Function prototype	void Listen(int sockfd, int backlog);
Description	Wrapper function for the UNIX listen() function. It places a socket in passive (i.e. listening) mode.
Input	sockfd – socket descriptor for the socket of interest backlog – maximum number of connections that the kernel should queue for the socket
Return	void
Constraints	None

Pseudo code	
	```
	If the environment variable LISTENQ is defined
	  Use it as the max queue length vice backlog
	Call listen()
	If error occurs
	  Display error message
	  Exit
	```

Function prototype	int Accept(int sockfd, struct sockaddr *addr, int *len);
Description	Wrapper function for the UNIX accept() function. It accepts a TCP connection on a passive socket. If there are no connections waiting in the queue, then it sleeps until a connection arrives. If an error occurs, then it displays an error message and returns normally.
Input	sockfd – listening socket descriptor addr – address pointer for the client's address len – int pointer for the length of the client's address
Return	The socket descriptor of the connection received. The client's address and address length for a good connection are returned via the addr and len parameters. If an error causes the function to terminate, then FALSE is returned and both addr and len pointers will be NULL.
Constraints	None

Pseudo code	
	```
	Call accept()
	If an error occurs while waiting for a connection
	  If the error was due to a signal being caught, continue waiting
	  Else return FALSE
	When a connection is received
	  Assign the client's socket address to addr
	  Assign the socket address length to len
	Return the socket descriptor of the connection
	```

Function prototype	u_int conv_pton(const char *str);

Description	Wrapper function for the UNIX inet_pton() function. It converts an IPv4 address in dotted notation to numeric format in network byte order.
Input	str – IPv4 address in dotted notation
Return	IPv4 address in numeric format and network byte order or FALSE on an error
Constraints	The address must be a valid IPv4 dotted address
Pseudo code	Call inet_pton() If the result is 1, then return the address If the result is 0 Display message stating that the address given was not a valid IPv4 address Return FALSE If the result is –1 Display message stating an error occurred during conversion Return FALSE

Function prototype	int readn(int fd, char *ptr, int nbytes);
Description	Wrapper function for the UNIX read() function. It reads n bytes from an open file descriptor. It is used for reading a desired number of bytes from an open socket without possibly having to make multiple read() calls.
Input	fd – the open file descriptor ptr – pointer for the data that was read nbytes – the desired number of bytes to read
Return	Integer telling the number of bytes actually read or –1 on an error. This value should always be equal to the desired number of bytes unless it is –1.
Constraints	None
Pseudo code	While less than the desired number of bytes has been read from fd Read the open socket descriptor If an error occurs If the error is due to a caught signal, continue reading Else return –1 Assign ptr to the data read Return the number of bytes read

Function prototype	int writen(int fd, char *ptr, int nbytes);
Description	Wrapper function for the UNIX write() function. It writes n bytes to an open file descriptor. It is used for writing a desired number

	of bytes to an open socket without possibly having to make multiple write() calls.
Input	fd – the open file descriptor ptr – pointer to the data to write nbytes – the desired number of bytes to write
Return	Integer telling the number of bytes actually written or –1 on an error. This value should always be equal to the desired number of bytes unless it is –1.
Constraints	None
Pseudo code	While less than the desired number of bytes has been written to fd Write to the open socket descriptor If an error occurs If the error is due to a caught signal, continue writing Else return –1 Return the number of bytes written

Function prototype	void insert_nt(char *str, int max, char tgt);
Description	Replaces the first instance of the given target character with a null terminator.
Input	str – string to scan max – maximum number of characters to scan tgt – the target character to be replaced with the null terminator
Return	void
Constraints	None
Pseudo code	Scan str until the tgt character is found or a null terminator is encountered or max characters have been scanned If tgt found, replace it with '\0'

Function prototype	void waitkey()
Description	Delays a program and prompts the user to press RETURN to continue. It is used for pauses to allow the user to see data displayed to the screen.
Input	void
Return	void
Constraints	None
Pseudo code	Prompt user to press RETURN to continue Get the return character

6.2. Socket Address Module

This section discusses the design's operations associated with the UNIX socket address structures listed below.

```
#define SOCKAI        struct sockaddr_in


/* Internet address; defined in <netinet/in.h> */
struct in_addr {
  u_long s_addr;        /* 32 bit IPv4 address; network byte ordered */
};


/* Socket address; defined in <netinet/in.h>; abbreviated SOCKAI in code */
struct sockaddr_in {
  u_char  sin_len;      /* total length = 16*/
  u_char  sin_family;   /* address family; value is AF_INET */
  u_short sin_port;     /* 16 bit TCP or UDP port; network byte ordered */
  struct   in_addr sin_addr;     /* IPv4 address */
  char     sin_zero[8];  /* unused */
};


/* Generic socket address structure supporting both IPv4 and IPv6;
   defined in <sys/socket.h> */
struct sockaddr {
  u_char sa_len;        /* total length */
  u_char sa_family;              /* address family; value is AF_xxx */
  char    sa_data[14];           /* protocol-specific address value */
};
```

<u>Socket Address LAVS</u>

The socket address module maintains a LAVS that is implemented as an array of SOCKAI pointers. The array length is set in the source file by a constant. The counter *sockai_num_avail* tracks the number of SOCKAI structures that are available in the LAVS at any given time. The following declarations and methods create and operate on the LAVS. The SOCKAI_lavs_prep() function should be listed in the socket address module's header file so that the LAVS can be populated at program startup. The get() and put() LAVS functions should be found only in the module's source file.

```
#define SOCKAI_LAVS_LENGTH          40

SOCKAI *sockai_lavs[SOCKAI_LAVS_LENGTH] = {0};
int sockai_num_avail = 0;
```

Function prototype	void SOCKAI_lavs_prep();
Description	Populates the LAVS by creating SOCKAI structures for every element of the LAVS
Input	void
Return	void
Constraints	None
Pseudo code	For each element in the LAVS array Call malloc() to allocate memory for a new SOCKAI structure Assign the element's pointer to the structure Set sockai_num_avail = SOCKAI_LAVS_LENGTH
Calls	malloc()
Effects	SOCKAI_LAVS_LENGTH new SOCKAI structures are created and added to the LAVS. The sockai_num_avail counter is assigned a new value.
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	SOCKAI *SOCKAI_lavs_get();
Description	Gets a SOCKAI structure from the LAVS.
Input	void
Return	Pointer to a usable SOCKAI structure or NULL if the LAVS is empty.
Constraints	None
Pseudo code	If sockai_num_avail > 0 Decrement sockai_num_avail Set a temp pointer to an available structure Set the pointer in the LAVS to NULL Return the temp pointer Else return NULL
Calls	None
Effects	Removes a SOCKAI structure from the LAVS. Decrements the sockai_num_avail counter.

Function prototype	int SOCKAI_lavs_put(SOCKAI *addr);
Description	Adds a SOCKAI structure to the LAVS. If the LAVS is already full then no action is taken on the SOCKAI.
Input	addr – the socket address structure to be added to the LAVS
Return	Integer denoting success; TRUE = success, FALSE = full LAVS
Constraints	None
Pseudo code	If sockai_num_avail = SOCKAI_LAVS_LENGTH Return FALSE Else If addr not equal to NULL Set the pointer for an open slot in the LAVS to the given SOCKAI structure Increment counter sockai_num_avail Return TRUE
Calls	None
Effects	Adds a SOCKAI structure to the LAVS. Increments the sockai_num_avail counter.

Socket Address functions

The socket address module creates and deletes socket address structures via the following functions:

Function prototype	SOCKAI *new_sockaddr(const char *dotted, const u_int port);
Description	Creates a new sockaddr_in structure with the given IP address and port number.
Input	dotted – dotted notation IP address port – port number
Return	A pointer to the dynamically allocated address structure
Constraints	Limited to IPv4 addresses.
Pseudo code	Call SOCKAI_lavs_get() If a structure is returned, then use it Else call malloc() to allocate memory for a SOCKAI structure Set the family field to AF_INET Call conv_pton() to convert the dotted address to a numerical address in network byte order Assign the numerical address to the structure's address value Call htons() to convert the port number to network byte order Assign the port number to the structure's port number Return a pointer to the structure

Calls	SOCKAI_lavs_get()
	malloc()

	conv_pton() – utility function htons() – system call
Effects	Creates a SOCKAI structure with the given parameters assigned to its data members in network form.
Comments	Verify that memory requested is actually allocated before continuing. Also, error check the htons() call. Exit on errors after displaying an error message.

Function prototype	void del_sockaddr(SOCKAI *addr);
Description	Deletes the given socket address structure.
Input	addr – the socket address structure to be deleted
Return	void
Constraints	Limited to IPv4 addresses.
Pseudo code	Call SOCKAI_lavs_put() to add the structure to the LAVS If the return value is FALSE Call free() to deallocate the structure

Calls	SOCKAI_lavs_put()
	free()

Effects	Adds the SOCKAI structure to the LAVS or deallocates it.

6.3 Firewall Database Module

This section defines the firewall record structure and describes its associated operations. The firewall database module maintains a private array of firewall records. The size of the array is determined by a constant value defined in the module's source code. The associated functions for the database act upon the it on behalf of a caller, return values from the database, or return handles to specific parts of the database. The controller uses a firewall database to track firewall machines that may be part of the parallel firewall architecture. The administrator, via either the controller's initialization file or its administrator interface, must provide some of a record's data (i.e. the control and data net addresses and the processing capacity of the firewall). The administrator sets the *handle* field of each record when the firewall database is initialized. The administrator controls the *availability* field, telling the system whether a particular record may be activated by the system. The field also tells the system whether a record in the database is empty. The administrator may modify or delete only disabled records. The system controls the *status* field of each record. The system may activate any record that is enabled by the administrator. Likewise, it may deactivate any record whose associated firewall machine it deems to be malfunctioning. The appropriate value for the *capacity* field of a given record should be determined by bench testing the firewall machine in its operational configuration. Appendix C discusses bench testing procedures. The system obtains the Ethernet address of a firewall's data net connection to the controller and records it in the *ether* field as part of the record activation procedure. The *misses* field is used to determine whether a firewall machine is malfunctioning. If the number of misses exceeds some threshold value set by the administrator, then the system deactivates the record. The *distribution* field value is assigned by the load-balancing algorithm. The scheduling procedure uses this value to determine which firewall should mediate a new connection or should process connectionless network traffic. The firewall record module also has a private counter, *fwdb_cnt*, that tracks the number of occupied record slots in the database.

```
/* Firewall record */
typedef  struct fwrecord {
  int handle;           /* used as a reference to the record from outside the database */
  short avail;          /* tells whether the record is available to be activated;
                           values are ENABLED, DISABLED, EMPTY */
  short status;         /* tells whether a firewall is currently active;
                           values are ACTIVE, INACTIVE */
  SOCKAI *caddr;        /* firewall's control net IPv4 address */
  u_long capacity,      /* firewall's processing capacity */
         load;          /* firewall's current load */
  SOCKAI *daddr;        /* firewall's data net Ipv4 address */
  char ether[14];       /* data net Ethernet address of firewall */
  u_short misses;       /* consecutive non-responses or poor health reports */
  u_long distr;         /* used for statistical distribution of network traffic */
} fwrecord;
```

The BSD/OS Address Resolution Protocol request structure is used by the firewall database module when obtaining Ethernet addresses for firewall data net IPv4 addresses. Its structure is shown below:

```
/* ARP request; defined in <net/if_arp.h> */
struct arpreq {
  struct sockaddr arp_pa;          /* protocol address */
  struct sockaddr arp_ha;          /* hardware address */
  int  arp_flags;                  /* flags */
};
```

The following declarations and methods define and operate on a firewall database:

```
#define FWDB_SIZE         10
```

```
fwrecord fwdb[FWDB_SIZE] = {0};
int fwdb_cnt = 0;
```

Function prototype	void fwdb_prep_handles();
Description	Initializes all records in the database with unique handle values.
Input	void
Return	void
Constraints	None
Pseudo code	Set counter to 0 Start at first database record While counter < FWDB_SIZE Set record's handle field to counter Increment counter
Calls	None
Effects	Assigns a value to the handle field of each firewall record

Function prototype	int fwdb_get_ether(int rec_hdl);
Description	Uses an ARP request to obtain the Ethernet address associated with a record's daddr field IP address. The function assumes the record exists.
Input	rec_hdl – handle to the firewall record of concern
Return	Integer indicating success or failure in getting the Ethernet address. TRUE for success, FALSE otherwise.
Constraints	None

Pseudo code	If a record referenced by the handle exists
	  Send ARP request to daddr field address
	  If Ethernet address received
	    Record address in ether field
	    Return TRUE
	  Else
	    Return FALSE
	Else return FALSE

Calls	Operating system calls to send the ARP request
Effects	Assigns a value to the ether field of the particular firewall record

Function prototype	int fwdb_insert_rec(SOCKAI *caddr, const u_long cap, SOCKAI *daddr);
Description	Writes a new firewall record into an empty slot in the firewall

	database. The avail field determines whether a particular slot is empty. If a record with the given control net IP address already exists, then that record is overwritten with the new data. Old data in the empty slot is zeroed prior to writing the new data.
Input	caddr – firewall's control net IP address cap – processing capacity of the firewall daddr – firewall's data net IP address
Return	Integer denoting success of inserting the new record. TRUE for success, FALSE if the database is full.
Constraints	Addresses must be dynamically allocated so the pointer value may be copied into the appropriate field of the record.
Pseudo code	Call fwdb_search_addr() to search the array for a matching caddr address. If a match exists Call fwdb_free_rec() to clear the record Call fwdb_new_rec() to write the new record data Set result to TRUE Else Call fwdb_get_empty_slot() to search for an empty slot in the database If the return value is not NULL Call fwdb_free_rec() to clear the record Call fwdb_new_rec() to write the new data Increment fwdb_cnt Set result to TRUE Else set result to FALSE Return result

Calls	fwdb_search_addr()
	fwdb_free_rec()
	fwdb_new_rec()
	fwdb_get_empty_slot()

Effects	A record is cleared of old data and overwritten with the new data provided as parameters. fwdb_cnt is incremented if the record was empty.
Comments	Existing data in an empty database slot is cleared before writing new data in order to deal with the dynamically allocated address structures it contains. The decision to clear the data at overwrite vice when a record is deleted avoids conflicts requiring mutual exclusion measures to correct. The problem scenario occurs if one process has a pointer to a record and is partially through its operation on the record when the record is deleted by another process.

Function prototype	int fwdb_new_rec(int rec_hdl, SOCKAI *caddr, const u_long cap, SOCKAI *daddr);
Description	Initializes the given firewall record with the given values
Input	rec_hdl – handle to firewall record to be initialized caddr – firewall's control net IP address cap – processing capacity of the firewall daddr – firewall's data net IP address
Return	TRUE = success, FALSE = bad record handle
Constraints	Addresses must be dynamically allocated so they pointer value may be copied into the appropriate field of the record.
Pseudo code	If a record referenced by the handle exists Set avail field to UNAVAIL Set status field to INACTIVE Assign the given parameters to their respective fields Return TRUE Else return FALSE
Calls	None
Effects	Writes data to various fields in a firewall record

Function prototype	int fwdb_del_rec(int rec_hdl);
Description	Makes an inactive firewall record available for overwrite by setting the avail bit to EMPTY.
Input	rec_hdl – reference to firewall record to be deleted
Return	Integer indicating success. TRUE = deleted, FALSE = the record is active so it could not be deleted.
Constraints	None
Pseudo code	result = FALSE If a record referenced by the handle exists If the record's avail field is EMPTY, set result to TRUE Else if the status field is ACTIVE If avail field = ENABLED then set result to FALSE Else Spin wait until status field is INACTIVE Set avail field to EMPTY Set result to TRUE Else /* status is INACTIVE and avail is DISABLED */ Set avail field to EMPTY Set result to TRUE Return result

Calls	None
Effects	A record's avail field is set to EMPTY and fwdb_cnt is decremented.
Comments	The data is cleared upon overwrite rather than upon deletion. This decision avoids conflicts requiring mutual exclusion measures to correct. The problem scenario occurs if one process has a pointer to a record and is partially through its operation on the record when the record is deleted by another process. The case where avail = DISABLED but status = ACTIVE is where the record has been disabled by the user but the process of terminating communications with active firewall is still in progress. Spin waiting is acceptable here because record deletions are expected to be infrequent events and the spin waiting time will be short, when it occurs.

Function prototype	int fwdb_search_dotted(const char *dotted, int *rec_hdl);
Description	Searches the firewall database for the given dotted notation control net IP address. Returns an integer (contents of rec_hdl) as a reference to the record.
Input	dotted – address of the firewall of interest rec_hdl – pointer to storage location for reference to the matching firewall record
Return	TRUE = found match, FALSE = no match
Constraints	Supports only IPv4 addresses
Pseudo code	Call conv_pton() to convert the dotted address to a network byte ordered numeric address While not found and more records exist If converted dotted address = record's caddr field address Copy handle field of the record to contents of rec_hdl Return TRUE Else advance to next record Return FALSE if no match found
Calls	conv_pton() – utility function
Effects	None

Function prototype	int fwdb_search_addr(SOCKAI *addr, int *rec_hdl);
Description	Searches the firewall database for the given socket address. Returns a reference to the matching record via rec_hdl.
Input	addr – address of the firewall of interest rec_hdl – pointer to storage location for a reference to the

	matching record
Return	TRUE = match found, FALSE = no match
Constraints	Supports only IPv4 addresses

Pseudo code	
	```
	While not found and more records exist
	  If the given address = the record's caddr field address
	    Copy handle field of the record to contents of rec_hdl
	    Return TRUE
	  Else advance to next record
	Return FALSE if no match found
	```

Calls	None
Effects	None

Function prototype	void fwdb_free_rec(int rec_hdl);
Description	Clears the data in the given record
Input	rec_hdl – reference to firewall record to be cleared
Return	void
Constraints	None

Pseudo code	
	```
	If a record referenced by the handle exists
	  Call del_sockaddr() to delete the address structures in the record
	  Call bzero() to zero all bytes in the firewall record structure
	```

Calls	del_sockaddr() – socket address function bzero() – system call
Effects	Overwrites all data in a firewall record with zeros

Function prototype	void fwdb_view();
Description	Displays the contents of the firewall database
Input	void
Return	void
Constraints	None

Pseudo code	
	```
	For each non-EMPTY record in the array
	  Display handle,
	          avail,
	          status,
	          control net address,
	          data net address,
	          capacity,
	          load
	```

Calls	None
Effects	None

Function prototype	int fwdb_chg_addr_cap(int rec_hdl, SOCKAI *caddr, SOCKAI *daddr, u_int *cap);
Description	Change the control address, data address, or capacity of a firewall record. Values not intended to be changed should be given a NULL value for the parameter. Active records may not be changed.
Input	rec_hdl – reference to firewall record to change caddr – control net address daddr – data net address cap – processing capacity
Return	TRUE = success, FALSE = active record or bad reference
Constraints	The address structures must be dynamically allocated so that the pointers may be copied directly into the record. Active records may not be changed.
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE If the status field is ACTIVE If avail field = ENABLED then set result to FALSE Else Spin wait until status field is INACTIVE For each non-NULL parameter If the parameters is an address Call del_sockaddr() to delete the existing address structure Change the appropriate field in given record to that value Set result to TRUE Else /* status is INACTIVE and avail is DISABLED */ For each non-NULL parameter If the parameters is an address Call del_sockaddr() to delete the existing address structure Change the appropriate field in the given record to that value Set result to TRUE Return result
Calls	del_sockaddr() – socket address function
Effects	Overwrites the appropriate fields with given values
Comments	Spin waiting is acceptable here because record changes are expected to be relatively infrequent events and the spin waiting time will be short, when it occurs.

Function prototype	int fwdb_get_empty_slot(int *rec_hdl);
Description	Gets a reference to an empty slot in the firewall database.

	Returns the reference via rec_hdl.
Input	rec_hdl – pointer to storage location for reference to empty record
Return	TRUE = success, FALSE = database is full
Constraints	None
Pseudo code	result = FALSE If fwdb_cnt < FWDB_SIZE Start at first element of database While more records exist and result is FALSE If avail field = EMPTY Copy handle field of the record to contents of rec_hdl result = TRUE Break from while loop Else advance to next element of database Return result
Calls	None
Effects	None

Function prototype	int fwdb_incr_misses(int rec_hdl);
Description	Increments the miss count for the given firewall record and returns the resultant count. The function will not roll over the value of misses. That is, when the value of the misses field gets to the maximum int value, it stops incrementing.
Input	rec_hdl – reference to firewall record for which to increment the miss count
Return	The revised miss count or –1 for a bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return –1 If misses < max integer value Increment miss count for record Return miss count
Calls	None
Effects	Adds 1 to the value of the miss field for one record up to limit of the maximum integer value.

Function prototype	int fwdb_reset_misses(int rec_hdl);
Description	Reset the miss count for the referenced firewall record
Input	Reference to firewall record for which to reset misses
Return	TRUE = success, FALSE = bad reference
Constraints	None

Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE Set misses field to 0 for record Return TRUE
Calls	None
Effects	Sets misses field to zero for one record

Function prototype	int fwdb_disable_rec(int rec_hdl);
Description	Disable a firewall record so that it may not be activated by the system
Input	Reference to firewall record to be disabled
Return	TRUE = success, FALSE = bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE Set avail field to DISABLED for record Return TRUE
Calls	None
Effects	Sets the avail field to DISABLED for one firewall record

Function prototype	int fwdb_enable_rec(int rec_hdl);
Description	Enable a firewall record so that it may be activated by the system
Input	Reference to firewall record to be enabled
Return	TRUE = success, FALSE = bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE Set avail field to ENABLED for record Return TRUE
Calls	None
Effects	Sets the avail field to ENABLED for one firewall record

Function prototype	int fwdb_deactivate_rec(int rec_hdl);
Description	Deactivate the referenced firewall record
Input	Reference to firewall record to be deactivated
Return	TRUE = success, FALSE = bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE

	Set status field to INACTIVE for record Return TRUE
Calls	None
Effects	Sets the status field to INACTIVE for one firewall record

Function prototype	int fwdb_activate_rec(int rec_hdl);
Description	Activate the referenced firewall record
Input	Reference to firewall record to be activated
Return	TRUE = success, FALSe = bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE Set misses field to 0 Set load field to 0 Set distrib field to 0 Set status field to ACTIVE Return TRUE
Calls	None
Effects	Sets operational field values to 0 then sets the status field to ACTIVE for one firewall record

Function prototype	int fwdb_set_load(int rec_hdl, const u_int val);
Description	Set the load field for the referenced firewall record to the given value
Input	rec_hdl – reference to firewall record to be changed val – value to be set for load
Return	TRUE = success, FALSE = bad reference
Constraints	None
Pseudo code	Find record referenced by record handle If a record referenced by the handle does not exist, return FALSE Set record's load field to val Return TRUE
Calls	None
Effects	Assigns a value to the load field for one firewall record

Function prototype	const char *fwdb_distr_select(const int val);
Description	Searches the firewall database for the record responsible for the given distribution value.
Input	val – distribution value of interest

Return	The Ethernet address of the record responsible for the given distribution value. If the database has no active records, then the empty string is returned.
Constraints	Assumes that the distr field values of active records in the array increase from first to last record.
Pseudo code	result = empty string result = ether field of the first active record in the array If result is not the empty string Start at first active record in array While distr field of the record is < val and more records exist Go to next active record If not past the end of the database, then result = ether field of the current record Return result
Calls	None
Effects	None

Function prototype	short fwdb_LBalg(const short prev_num);
Description	Calculates the statistical distribution coefficients used to determine where a new connection should be routed.
Input	prev_num – the number of firewalls in the load-balancing scheme the last time the algorithm was run
Return	Number of firewalls in the current load-balancing scheme
Constraints	None

Pseudo code	count = 0
	For each enabled, active record in the firewall database
	  Copy the capacity and load fields
	  Increment count
	Calculate the total capacity, cap_{tot}, of all active firewalls
	Calculate the total load, L_{tot}, of all active firewalls
	For each active record x
	  $m_x = cap_x / cap_{tot}$
	  If count equals prev_num
	    $LF_x(t) = L_x(t) / L_{tot}$ // this is the load factor for x
	    $k_x(t) = LF_x(t) - m_x$ // this is the correction factor for x
	    $df_x(t+1) = df_x(t) - r * k_x(t)$ // this is the corrected distribution
	                            // factor for x with a relaxation
	                            // factor, r, included
	  Else $df_x(t+1) = m_x$
	Set distr of first active record copied equal to $df_1(t+1)$
	For each active record copied, starting at second one,

	Set distr to distr of previous active record + $df_x(t+1)$ Return count
Calls	None
Effects	Writes a new value to the distr field of each active record

6.4 Load Table Module

This section defines load element and load record structures and their associated operations. The load monitor on each firewall machine uses these structures to map load to TCP connections. The load record structure consists of an average and number of load samples taken for a particular TCP connection, identified by its source and destination socket addresses. The load record structure is designed to be a node in a linked list. A linked list data structure is used for a load table since the number of TCP connections can vary greatly and insertions/deletions of nodes will be frequent.

```
/* Load element */
struct ld_element {
  u_int avg,                    /* average of samples taken */
        samples;                /* number of samples */
};


/* Load record */
typedef  struct ld_rec {
  char *source,                 /* source address */
       *dest;                   /* destination address */
  struct ld_element load;
  struct ld_tbl *prev;
  struct ld_tbl *next;
} ld_rec;
```

Load Table LAVS

The load table module maintains a private LAVS for load record structures. The LAVS is implemented as a singly linked list using the *prev* and *next* pointers in the load record structure. The LAVS is initialized by calling function lt_lavs_prep(). This function should be included in the header file. The initial length of the LAVS is determined by a constant in the module's source file. The following declarations and methods create and operate on the load table LAVS:

```
#define LT_LAVS_INITLENGTH          50
```

```
ld_tbl *lt_lavs_head = NULL;    /* Pointer to head of LAVS */
ld_tbl *lt_lavs_tail  = NULL;    /* Pointer to tail of LAVS */
```

Function prototype	void lt_lavs_prep();
Description	Creates LT_LAVS_INITLENGTH ld_rec structures and adds them to the LAVS.
Input	void
Return	void
Constraints	None
Pseudo code	For count = 0 to LT_LAVS_INITLENGTH – 1 Call malloc() to allocate memory for a new ld_rec structure Call lt_lavs_put() to add the new structure to the LAVS

Calls	malloc()
	lt_lavs_put()

Effects	Increases the LAVS length by LT_LAVS_INITLENGTH nodes
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	void lt_lavs_put(ld_rec *node);
Description	Appends the given ld_rec stucture to the tail of the LAVS
Input	node – ld_rec structure to be added to the LAVS
Return	void
Constraints	None
Pseudo code	Set prev and next pointers for node to NULL If lt_lavs_tail is NULL Set lt_lavs_tail to node Set lt_lavs_head to node Else Set next pointer for lt_lavs_tail to node Set lt_lavs_tail to node
Calls	None
Effects	Adds one ld_rec node to the LAVS

Function prototype	void lt_lavs_putlist(ld_rec *ptr);
Description	Appends the given record list to the tail of the LAVS
Input	ptr – pointer to head of list of load records to be added to the LAVS
Return	void
Constraints	None

Pseudo code	If lt_lavs_tail is NULL Set lt_lavs_head to node Set lt_lavs_tail to tail of the list being added to the LAVS Else Set next pointer for lt_lavs_tail to ptr Set lt_lavs_tail to tail of list being added to the LAVS
Calls	None
Effects	Adds a list of load records to the LAVS

Function prototype	ld_rec *lt_lavs_get();
Description	Gets a ld_rec structure from the head of the LAVS.
Input	void
Return	Pointer to usable ld_rec structure or NULL if the LAVS is empty
Constraints	None

Pseudo code	If lt_lavs_head is NULL
	  Return NULL
	Else if lt_lavs_head equals lt_lavs_tail
	  Set a temp pointer to lt_lavs_head
	  Set lt_lavs_head to NULL
	  Set lt_lavs_tail to NULL
	  Return temp pointer
	Else
	  Set a temp pointer to lt_lavs_head
	  Set lt_lavs_head to lt_lavs_head next pointer
	  Return temp pointer

Calls	None
Effects	Removes one ld_rec node from the LAVS

Load Table functions

The load table module maintains a private doubly linked list of load record nodes as a load table. The following declarations and methods operate on the load table for client programs:

```
ld_rec *ld_table = NULL;
```

Function prototype	u_long lt_update_table();
Description	Update the load table with a load sample from the system
Input	void

Return	The total queue length of connectionless traffic in bytes
Constraints	Treats all non-TCP traffic as connectionless

Pseudo code	Call netstat() to get a load sample for network connections
	Sort the output of netstat using merge sort (O(nlogn)) or a O(1) sort function so that entries are in ascending order by source socket address
	For each socket entry in netstat output
	  If the socket addresses do not match the next table entry
	    If the table entry's socket was closed
	      Call lt_del_rec() to delete the entry
	    Else
	      Call lt_new_ld_rec() to create a new ld_rec structure
	      Call lt_insert_rec() to insert the new socket into the table
	  Else update the load record as follows
	      old_total = samples * avg;
	      samples = samples + 1;
	      avg = (old_total + new sample queue length) / samples;

Calls	netstat() – system function call
	lt_new_rec()
	lt_insert_rec()
	lt_del_rec()

Effects	Each record in the load table is updated with a new load sample or deleted, and new records may be added to the table

Function prototype	ld_rec *lt_new_rec(const char *src, const char *dest, u_int ldval);
Description	Generates a new load record structure with the given parameters. All other parameters are set to 0.
Input	scr – source address dest – destination address ldval – value of load for the new record
Return	Pointer to the new load record structure
Constraints	The source and destination addresses must be dynamically allocated so they can be used directly in the record.
Pseudo code	Call lt_lavs_get() to get a record from the LAVS If the return value is non-NULL, then set result pointer to it Else Call malloc() to allocate memory for a new load record Set result pointer to new record Call free() to deallocate the source and dest addresses Call bzero() to clear the record

	Assign the given parameters to the appropriate fields Return result pointer

Calls	lt_lavs_get()
	malloc()
	bzero() – system call
	free()

Effects	None
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	void lt_insert_rec(ld_rec *marker, ld_rec *node);
Description	Inserts a load record into the load table just before the marker.
Input	marker – load record position before which to add the new node node – load record to be added to table
Return	void
Constraints	The pointer manipulation must be implemented such that no problems exist with another process iterating through the list head-to-tail during the insertion.
Pseudo code	If the load table is empty, then set the table pointer to node Else manipulate prev and next pointers to insert node into list
Calls	None
Effects	Inserts a new load record into the load table
Comments	Verify pointers are non-NULL

Function prototype	void lt_del_rec(ld_rec *node);
Description	Deletes the given load record from the load table
Input	node – load record to be deleted from table
Return	void
Constraints	The pointer manipulation must be implemented such that no problems exist with another process iterating through the list head-to-tail during the deletion.
Pseudo code	If the load table is empty or node is null, then return Manipulate prev and next pointers to extract node from table Call lt_lavs_put() to add the record to the LAVS
Calls	lt_lavs_put()
Effects	Deletes a record from the load table and adds it to the LAVS

Function prototype	u_long lt_calc_conn_load();
Description	Calculates total load for the load table and clears the table in

	preparation for the next iteration of sampling.
Input	void
Return	Total load in table
Constraints	None
Pseudo code	Total = 0 For each record in load table Add load to total Call lt_lavs_putlist() to clear the load table of records Return total
Calls	lt_lavs_putlist()
Effects	Empties the load table of records

6.5 Route Table Module

This section describes the routing table entry structure. The routing process in the controller uses this structure to track existing TCP connections through the firewall architecture. Each entry consists of the source and destination socket addresses and the Ethernet address of the firewall machine mediating that connection. The routing table used by the routing process is a hash table. The route entries are data elements in the hash table.

```
/* Routing table entry */
typedef  struct rt_entry {
  SOCKAI *source,           /* source socket address */
         *dest;             /* destination socket address */
  char ether[14];           /* Ethernet addr of firewall handling conn */
} rt_entry;
```

Route Table LAVS

The route table module maintains a LAVS that is implemented as an array of rt_entry pointers. The array length is set in the source file by a constant. The counter *rt_num_avail* tracks the number of rt_entry structures that are available in the LAVS at any given time. The following declarations and methods create and operate on the LAVS. The rt_lavs_prep() function should be listed in the route table module's header file so that the LAVS can be populated at program startup. The get() and put() LAVS functions should be found only in the module's source file.

```
#define RT_LAVS_LENGTH          100

rt_entry *rt_lavs[RT_LAVS_LENGTH] = {0};
int rt_num_avail = 0;
```

Function prototype	void rt_lavs_prep();
Description	Populates the LAVS by creating rt_entry structures for every element of the LAVS
Input	void
Return	void
Constraints	None
Pseudo code	For each element in the LAVS array Call malloc() to create a new rt_entry structure Assign the element's pointer to the structure Set sockai_num_avail = RT_LAVS_LENGTH
Calls	malloc()
Effects	RT_LAVS_LENGTH new rt_entry structures are created and added to the LAVS. The rt_num_avail counter is assigned a new value.
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	rt_entry *rt_lavs_get();
Description	Gets a rt_entry structure from the LAVS. If the LAVS is empty, then a NULL pointer is returned.
Input	void
Return	Pointer to a usable rt_entry structure or NULL if the LAVS is empty.
Constraints	None
Pseudo code	If rt_num_avail > 0 Decrement rt_num_avail Set a temp pointer to an available structure Set the pointer in the LAVS to NULL Return the temp pointer Else return NULL
Calls	None
Effects	Removes a rt_entry structure from the LAVS. Decrements the rt_num_avail counter.

Function prototype	int rt_lavs_put(rt_entry *addr);
Description	Adds a rt_entry structure to the LAVS. If the LAVS is already full then no action is taken on the rt_entry.
Input	addr – the socket address structure to be added to the LAVS

Return	Integer denoting success; TRUE = success, FALSE = full LAVS
Constraints	None

Pseudo code	If rt_num_avail = RT_LAVS_LENGTH
	  Return FALSE
	Else
	  Set the pointer for an open slot in the LAVS to the given
	      rt_entry structure
	  Increment counter rt_num_avail
	  Return TRUE

Calls	None
Effects	Adds a rt_entry structure to the LAVS. Increments the rt_num_avail counter.

Route Table functions

The route table module creates and deletes rt_entry structures via the following functions:

Function prototype	rt_entry *rt_new_entry(SOCKAI *src, SOCKAI *dest, char *ether);
Description	Creates a new route entry or uses one from a LAVS. Assigns the given parameters to the route entry.
Input	src – pointer to source address dest – pointer to destination address ether – Ethernet address of recipient firewall
Return	Route entry with given parameters
Constraints	Assumes the address structures have been dynamically allocated for use in the route entry.
Pseudo code	Call rt_lavs_get() to obtain a rt_entry structure from the LAVS If the result is non_NULL Call del_sockaddr() to delete existing socket address structures in the rt_entry structure Else Call malloc() to allocate memory for a rt_entry structure Assign the given parameter values Return pointer to structure

Calls	rt_lavs_get()
	malloc()
	del_sockaddr() – socket address function

Effects	Generates a new rt_entry structure for use
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	void rt_del_route(rt_entry *tgt);
Description	Deletes a route entry structure
Input	Route entry to delete
Return	void
Constraints	None
Pseudo code	If tgt = NULL, then return Call rt_lavs_put() to place the rt_entry strucutre in the LAVS If the result was FALSE Call del_sockaddr() to delete the socket address structures in the rt_entry structure Call call free() to deallocate the rt_entry structure

Calls	rt_lavs_put()
	free()
	del_sockaddr() – socket address function

Effects	Deletes a rt_entry structure from use

<u>Routing Table data structure</u>

As mentioned earlier, the routing table is an instantiation of a hash table. A pre-existing hash table package [Ref 4] is used for the routing table. The size of the hash table is determined by a constant in the route table module source file. The hash table module defines the following data types:

```
typedef struct bucket {
   char *key;
   void *data;
   struct bucket *next;
} bucket;

typedef struct hash_table {
   u_int size;
   bucket **table;
} hash_table;
```

The following declaratioins and methods create and operate on the routing table:

```
#define RT_SIZE       10000

hash_table *rt_table;
```

Function prototype	void rt_table_prep();
Description	Constructs the routing table
Input	void
Return	void
Constraints	None
Pseudo code	Call construct_table(rt_table, RT_SIZE) to create the table If the return value is NULL Display an error message to the screen Exit
Calls	construct_table() – hash table function
Effects	Generates the hash table to be used as the routing table

Function prototype	char *rt_genkey(SOCKAI *src, SOCKAI *dest);
Description	Generates a key from a socket address for use in the routing table
Input	src – the source socket address dest – the destination socket address
Return	A string that is the key
Constraints	The return value must be dynamically allocated
Pseudo code	Call malloc() to allocated memory for the key Call inet_ntop() to convert the src and dest IP addresses to strings Assign converted src address to key Convert src and dest port numbers to strings Call strcat() to add the src port number, dest address, and dest port to the end of the key value Return key
Calls	construct_table() – hash table function malloc()
Effects	Generates the hash table to be used as the routing table
Comments	Verify that memory requested is actually allocated before continuing. Exit on error after displaying an error message.

Function prototype	void rt_addroute(rt_entry *newrt);
Description	Adds a route to the routing table
Input	newrt – the route to be added to the table
Return	void
Constraints	The rt_entry structure must be dynamically allocated for use in the table.
Pseudo code	If newrt is not NULL Call rt_genkey() with the source address to generate a key

	Call insert(key, newrt, rt_table) to insert the route into the table
Calls	rt_genkey() insert() – hash table function
Effects	Adds an entry to the routing table

Function prototype	rt_entry *rt_lookup(SOCKAI *src, SOCKAI *dest);
Description	Looks up a route entry in the routing table
Input	src – the source socket address dest – the destination socket address
Return	Pointer to the route entry found or NULL if no match found
Constraints	None
Pseudo code	Call rt_genkey() to generate the key for lookup Call lookup(key, rt_table) to look up the key Return result of lookup()
Calls	rt_genkey() lookup() – hash table function
Effects	None

Function prototype	void rt_table_delrt(SOCKAI *src, SOCKAI *dest);
Description	Deletes a route entry from the routing table
Input	src – the source socket address dest – the destination socket address
Return	void
Constraints	None
Pseudo code	Call rt_genkey() to generate the key Call del(key, rt_table) to delete the entry in the hash table and return the rt_entry structure that was its data Call rt_del_route(route) to delete the rt_entry structure

Calls	rt_genkey()
	del() – hash table function
	rt_del_route()

Effects	Deletes an entry from the routing table and deletes the rt_entry

6.6 Controller Administrative Process

The controller has administrative, update server, and routing processes. This section describes functions and modules that make up the administrative process. The administrative process (ADMIN) starts upon execution of file *control*. It provides the program's administrator interface and spawns the controller's operational processes (i.e. the update server and routing processes). ADMIN reads operational parameters and firewall information from an initialization file. It uses the firewall data to generate firewall records in the firewall database. It shares this firewall database and the operational parameters with all of its child processes. As shown in Figure 2, ADMIN consists of control logic, operational functions, and database support modules. The control logic module consists of functions main(), init_ctrl(), view_params(), and change_param(). The operations module consists of functions start_ctrl(), stop_ctrl(), add_fw(), and change_fwt(). The support modules were explained previously.

ADMIN records the process id of each of its children and records the operational state (i.e. operating or not) of the controller in local variables. It uses the process id values to halt operation of the controller. ADMIN halts operational of the controller on three occasions: 1) the administrator selects stop controller operation, 2) the administrator selects exit from the administrative menu, and 3) one of the child processes terminates due to an error. The child processes created by ADMIN are expected never to return until ADMIN signals them to terminate so a child process terminating for any reason other than items one or two above means it terminated due to an error. ADMIN receives a SIGCHLD signal from the operating system when a child process terminates. The state variable prevents more than one instance of the operational processes from existing at any given time.

Function prototype	int main();
Description	This is the ADMIN process. Choices in the user interface are menu driven. Selecting stop or exit causes ADMIN to kill its child processes. Selecting deactivate firewall record causes ADMIN to signal the update server to run the load-balancing algorithm.
Input	void
Return	Integer telling the operating sytem the exit status of the program.
Constraints	None
Pseudo code	Call init_ctrl() to initialize the program by reading from init file While exit not chosen Display menu – menu contains options to start/stop controller, view/add/delete/modify/enable/disable firewall records, view/modify operation parameters, and exit Perform selected option If SIGCHILD signal received Signal children to terminate Display an alert message Set state = NOT OPERATING

Calls	init_ctrl()
	start_ctrl()
	stop_ctrl()

	view_params()
	change_param()
	fwdb_view() – fwrecord library
	change_fwt()

IPC	This is a list of IPC for the ADMIN process: Signals update server that a firewall record has been disabled by the administrator. Signals both the update server and LBrouter to exit when stop controller operation or exit program is selected by administrator, or one of the child processes terminates.

Function prototype	void init_ctrl(char *ifile, u_int *pollptr, u_int *failptr, float *relaxptr, SOCKAI *uaddr);
Description	Reads the controller's initialization file, building the firewall record array and setting the operational parameters
Input	ifile – the initialization file's name pollptr – pointer to storage location for polling increment in seconds failptr – pointer to storage location for count to failure. This is the number of consecutive load update polls for which either a non-response or poor health report is received before the controller deactivates a firewall record relaxptr – pointer to storage location for relaxation factor used in the load-balancing algorithm uaddr – update socket address for controller
Return	void
Constraints	None
Pseudo code	Call Fopen() to open the initialization file Read update address for controller's control net listening socket Read polling increment from file Read fail count from file Read relaxation factor from file While firewall info continues and space exists in firewall table Read firewall record information and add to firewall table
Calls	Fopen() – utlity library new_sockaddr() – socket address library fwdb_insert_rec() – fwrecord library
Comments	Error checking of input file format and values read must be included. Detected errors should cause the program to exit.

Function prototype	void start_ctrl(int *state, u_int *pollptr, u_int *failptr, float *relaxptr);
Description	Starts controller operation – spawns UPDATE SERVER and LB ROUTER processes
Input	state – pointer to the operational state variable pollptr – pointer to poll increment failptr – pointer to fail count relaxptr – pointer to the relaxation factor The parent (ADMIN) has all of the variables so the children need pointers to them.
Return	void
Constraints	Should not be able to start up more than one instance of the child processes.

Pseudo code	If state variable = NOT OPERATING
	  Spawn update server process
	  Spawn routing process
	  Set state variable = OPERATING

Calls	update_server()
	LBrouter()

Function prototype	void stop_ctrl(int *state, pid_t udserver, pid_t lbrouter);
Description	Stops controller operation – signals UPDATE SERVER and LB ROUTER processes to exit
Input	state – pointer to the state variable udserver – PID of UPDATE SERVER process lbrouter – PID of LBROUTER process
Return	void
Constraints	None
Pseudo code	If state variable = OPERATING Signal update server process to terminate using SIGUSR1 Signal routing process to terminate using SIGUSR1 Set state variable = NOT OPERATING Display message saying controller operation stopped Else display message saying controller not operating
Calls	None

Function prototype	void view_params(const SOCKAI *ctrl, const u_int poll, const u_int fail, const float relax);
Description	View the controller's operational parameters
Input	ctrl – pointer to controller's control net address

	poll – polling increment in seconds fail – number of consecutive non-responses or poor health reports required before controller deactivates a firewall record relax – relaxation factor used by load-balancing algorithm
Return	void
Constraints	None
Pseudo code	Display each parameter to screen
Calls	None

Function prototype	void change_param(SOCKAI *ctrl, u_int *poll, u_int *fail, float *relax);
Description	Allows the user to change the controller's operational parameters
Input	ctrl – pointer to controller's control net address poll – pointer to polling increment fail – pointer to fail count relax – pointer to relaxation factor
Return	void
Constraints	None

Pseudo code	While exit not selected
	  Call view_params() to display parameters
	  Prompt user for parameter to change
	  If exit option selected, then return
	  Prompt for values
	  Change parameters

Calls	view_params()

Function prototype	void change_fwt(int *state);
Description	Change the contents of the firewall table
Input	void
Return	void
Constraints	None
Pseudo code	While exit not selected Call fwdb_view() Display numerical codes for changing fields, adding/deleting records, enabling/disabling records, and exiting menu Prompt for a selection code If code is for exit, then return If code is for add record Call add_fw() Else

	```
	  Prompt for record number
	  If code is for delete
	    If fwdb_del_rec() yields FALSE, then tell user that the
	        record must be disabled before delete
	  Else if code is for enable
	    Call fwdb_enable_rec()
	  Else if code is for disable
	    Call fwdb_disable_rec()
	    If contents of state = OPERATING
	      Signal UPDATE SERVER using SIGUSER2 that a record
	          has been disabled
	  Else if code is for modify record
	    Get field code
	    Get new value
	    Call fwdb_change_addr_cap()
	  Else display bad input message
	```

Calls	add_fw()
	fwdb_view()
	fwdb_change_addr_cap()
	fwdb_enable_rec()
	fwdb_disable_rec()
	fwdb_del_rec()

Comments	Include error checking on field codes and values entered.

Function prototype	void add_fw();
Description	Prompts for control net and data net addresses and capacity of a new firewall. Adds a new record to the given firewall array.
Input	void
Return	void
Constraints	None

Pseudo code	
	```
	If fwdb_get_empty_slot() yields TRUE
	  Prompt for control address
	  Prompt for data address
	  Prompt for capacity
	  Call new_sockaddr() to generate control and data net address
	      structures
	  Call fwdb_insert_rec() to add record to firewall database
	Else display message saying database is full
	```

Calls	fwdb_get_empty_slot()
	new_sockaddr() – socket address library
	fwdb_insert_rec()

6.7 Controller Update Server Process

The update server process (UPDATE SERVER) listens on the control net for firewalls to announce themselves as ready to process network traffic. When a new connection is received on the listening port, UPDATE SERVER spawns a duplicate of itself as a child process to handle the new connection. If the new connection passes the tests described in Chapter 3 of Reference 1, then the child process polls the firewall for periodic load updates. The parent UPDATE SERVER continues listening for connections and coordinates both polling of the firewalls and calling the load-balancing algorithm. Figure 2 shows the UPDATE SERVER as having control logic, communications logic, and database support modules. The control logic consists of functions update_server(), processmsg(), and get_load(). The communications logic consists of calls to wrapper functions from the utility library for UNIX socket communication function calls. The other modules are support modules described earlier.

UPDATE SERVER maintains a process id array the size of the firewall database (FWDB_SIZE). As it spawns child processes to handle new connections, it records the child's process in the array slot corresponding to the handle number of the firewall record the child is updating. This PID array serves two purposes: it allows the parent UPDATE SERVER to relay disabled record signals from ADMIN to the child UPDATE SERVERS and it allows the parent to verify that the associated firewall record is deactivated when one of its children terminates. The verification item protects against a child UPDATE SERVER terminating due to an error without deactivating its firewall record.

Function prototype	void update_server(u_int *pollptr, u_int *failptr, float *relaxptr);
Description	Starts the UPDATE SERVER. The update server listens for firewall machines to announce themselves on the control net. It spawns a child for each firewall. The children periodically poll the firewalls for load updates. The parent coordinates polls for load updates and calls the load-balancing algorithm as appropriate.
Input	pollptr – pointer to poll increment failptr – pointer to fail count relaxptr – pointer to relaxation factor
Return	void
Constraints	None

Pseudo code	
	```
	Call Socket(), Bind(), and Listen() to open a listening socket on
	    the control net
	Call Accept() to accept new connections
	When new connection received
	  Spawn a child to handle connection
	    Child calls fwdb_search_addr() to verify the message is from
	```

```
    a recognized firewall (i.e. it is in an enabled record in the
    firewall table)
  If result is non-NULL, call processmsg()
  Else
    Send UNKNOWN_MACH response to firewall
    Close connection
    Exit
  If connection passes tests in processmsg()
    Call fwdb_get_ether() to get data net Ethernet address of
        firewall
    If successful
      Call fwdb_activate_rec() to activate record
      Call get_load() to poll firewall with poll number when
          poll signal received from parent
      If firewall does not respond or gives poor health report
        Call fwdb_incr_misses() to increment miss count
        If miss count return value equals fail count
          Call fwdb_deactivate_rec() to deactivate record
          Print alert to screen
          Close connectioin
          Exit
      Else
        Call fwdb_reset_misses() to reset miss count
      If disable signal (SIGUSER2) received from parent
        Check avail field of the firewall record
        If avail field = DISABLED
          Close socket to firewall
          Call fwdb_deactivate_rec()
          Exit
      Wait for poll signal (SIGUSER1) from parent

Parent does the following:
  Continues listening for new connections
  Maintains the master poll number and next poll time.  Signals
      children to poll (SIGUSER1) when next poll time reached
  Handles disable signals (SIGUSER2) from the admin process
  Signals children (SIGUSER2) when a firewall has been
     disabled by the admin process (i.e. parent receives
     SIGUSER2 from ADMIN)
  Calls load-balancing algorithm after updates received or
     firewall disabled/deactivated.  Increments poll number if
     LBalg() call was from a polling cycle.
  Exits on receipt of SIGUSER1 from ADMIN
```

Calls	processmsg()
	get_load()
	Socket()
	Bind()
	Listen()
	Accept()
	fwdb_LBalg()
	fwdb_incr_misses()
	fwdb_activate_rec()
	fwdb_deactivate_rec()
	fwdb_reset_misses()
	fwdb_search_addr()

IPC	Receives a signal from ADMIN when the administrator disables a firewall record. Parent signals children to check their respective avail field when a firewall record has been disabled by ADMIN. Parent signals children to poll for load update

Function prototype	void processmsg(int connfd, int rec_hdl, const char *dotted);
Description	Processes an announcement message received from a firewall. Sends a bad hello response if the hello message is not the one expected. If the hello message passes its test, then the function sends an ARP request for the firewall's data net Ethernet address. If the ARP request is unsuccessful, then the function sends a bad data net addr message to the firewall. If all checks pass, the function sends an acknowledgement message. It displays a message to the user notifying them of whatever happens.
Input	connfd – socket descriptor to firewall machine rec_hdl – reference to the firewall record associated with the message received dotted – dotted notation IPv4 address from record
Return	void
Constraints	None
Pseudo code	Read hello message If message is that expected Get the data net Ethernet address of the firewall If unsuccessful Send bad data net addr message to firewall Display notification to screen Else Acknowledge the firewall

	Activate the record Display notification to screen Else Send bad hello response Display notification to screen Exit
Calls	readn() – utility library writen() – utility library

Function prototype	int get_load(int rec_hdl, int fd, const int timeout, const int pollnum);
Description	Poll the given socket for a load update and place the result in the given firewall record.
Input	rec_hdl – reference to firewall record for the firewall to poll fd – open socket descriptor to the firewall timeout – max time for firewall to respond pollnum – index number of poll
Return	Integer indicating whether an update was received in the given time period; TRUE = received, FALSE = not received
Constraints	None
Pseudo code	Send poll with poll id number to firewall Sleep until answer received or timeout period expires If answer received Call fwdb_set_load() to update load in firewall record Verify poll id is correct (checks for late responses) If poll id matches the latest one sent Return TRUE Else continue listening Else return FALSE
Calls	readn() – utility library writen() – utility library fwdb_set_load()

6.8 Controller Load-balancing Router Process

The controller's load-balancing routing process (LBROUTER) intercepts outbound network traffic and routes it to the appropriate firewall machine for processing. Figure 2 shows LBROUTER as having routing logic, a scheduler, and database support functions for both the firewall database and a routing table. Function LBrouter() is the routing logic and function scheduler() is the scheduler module. All other modules are firewall database and routing table support modules described earlier.

Function prototype	void LBrouter();
Description	Starts load-balancing router execution. This function routes new TCP connections and connectionless traffic to firewalls according to the load-balancing scheme generated by the load-balancing algorithm.
Input	void
Return	void
Constraints	None

Pseudo code	
	```
	Use BSD Packet Filter (BPF) to intercept all outbound packets
	     and also those inbound packets containing reset commands
	If outbound packet
	  If packet is for connectionless traffic
	    Call scheduler() to determine firewall recipient
	    Make a new Ethernet frame
	    Send packet to firewall using BPF
	  Else if packet is for a new TCP connection
	    Call scheduler() to determine firewall recipient
	    Call rt_addrt() to record entry in routing table
	    Make a new Ethernet frame
	    Send packet to firewall using BPF
	  Else
	    Call rt_lookup() to determine whether packet is for an existing
	       connection
	    If packet is for an existing entry in routing table
	      Make a new Ethernet frame
	      Send packet to appropriate firewall
	      If packet terminated the connection
	        Call rt_table_delrt() to delete the assoc. routing table entry
	        Call rt_del_route() to delete the rt_entry structure
	    Else drop it
	If inbound packet (containing a reset command)
	  Call rt_table_delrt() to delete the associated routng table entry
	  Call rt_del_route() to delete the rt_entry structure

	If signal to exit (SIGUSER1) is received from ADMIN
	  Exit
	```

Calls	scheduler() new_sockaddr() – socket address library

	rt_new_entry()
	rt_addrt()
	rt_lookup()
	rt_table_delrt()
	rt_del_route()

IPC	Receives signal from ADMIN to exit.

Function prototype	const char *scheduler();
Description	Determines which firewall machine should get the next connection
Input	void
Return	The Ethernet address of the selected firewall
Constraints	None
Pseudo code	Generate a random number (u_long) Call fwdb_distr_select(random number) to determine the firewall that has the random number in its distribution range Return the resultant Ethernet address
Calls	fwdb_distr_select()

6.9 Firewall Load Monitor Process

This section describes the modules and functions used in each firewall load monitor. The load monitor process is started by executing file *loadmon*. The load monitor process on each firewall announces itself to the controller after initialization. Upon receipt of an acknowledgement, it starts monitoring load by periodically sampling the queue length of each TCP connection through the firewall. It reports the results of its load sampling to the controller when the controller polls it. The process consists of three modules: control logic, communications logic, and database support modules for its load table. The control logic consists of functions main(), init_lm(), and check_health(). The communications logic consists of function announce() and wrapper functions in the utility library for UNIX system socket communications functions. The load table support module was described earlier.

Function prototype	int main()
Description	Runs the load monitor's control logic. It announces the firewall to the controller, periodically samples the firewall machine's load and checks that the firewall process is still running, and sends load updates to the controller when polled.
Input	void
Return	Integer indicating the exit status of the program

Constraints	None
Pseudo code	Call init_lm() to initialize the load monitor Call announce() to announce firewall to controller While TRUE Call lt_update_table() to monitor firewall load Call check_health() to check firewall health Listen for update request from controller for sample period seconds. When poll recived Call lt_update_table() to get latest TCP update for load table to get and connectionless load Call lt_calc_conn_load() to calculate TCP load Reply to poll with health bit and sum of TCP and connectionless load

Calls	init_lm()
	announce()
	health_check()
	lt_update_table()
	lt_calc_conn_load()

Function prototype	void init_lm(char *ifile, SOCKAI **uaddr, u_int *incr);
Description	Initializes the LOAD MONITOR process by reading the given initialization file.
Input	ifile – name of init file uaddr – pointer to storage location for controller's update address on control net incr – pointer to storage location for load sample increment in seconds
Return	void
Constraints	None
Pseudo code	Call Fopen() to open initialization file Read the controller's update address and the sample increment Close init file
Calls	Fopen() – utility library new_sockaddr()
Comments	Include error checking for file format and values being read

Function prototype	int announce(SOCKAI *addr);
Description	Sends hello message to the socket address provided.
Input	Pointer to destination socket address

Return	Socket descriptor of open socket to controller
Constraints	None
Pseudo code	Call Socket() to open socket to addr Send hello message periodically until response received If bad message or bad data net address response message received Display alert to screen Exit If unknown machine message received Display alert to screen Continue sending hello message every ANNOUNCE INTERVAL seconds If acknowledgement received, return socket descriptor

Calls	Socket()
	readn()
	writen()

Function prototype	int check_health();
Description	Uses BSD Packet Filter (BPF) to check that the firewall process is sending traffic.
Input	void
Return	TRUE = good health, FALSE = bad health
Constraints	None
Pseudo code	Call BPF to listen for incoming and outgoing traffic If incoming but no outgoing traffic seen Check the process status to determine whether the firewall process is able to run If not able to run, then return FALSE If able to run, return TRUE
Calls	None

References

1. William Joyner, "Design of a Load-Balancing Architecture for Parallel Firewalls," Naval Postgraduate School, Monterey, California, March 1999.
2. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.
3. Edsger W. Dijkstra, "The Structure of "THE" – Multiprogramming System", Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 341-346.
4. Jerry Coffin, from Bob Stout's Snippet web page under the section entitled Jerry Coffin's hash table functions, [http://www.brokersys.com/snippets], 02 November 1995.
5. W. Richard Stevens, UNIX Network Programming, 2nd edition, Vol. 1, Networking APIs: Sockets and XTI, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

APPENDIX C. A DISCUSSION OF BENCH TESTING

An assumption of the load-balancing firewall software design is that an accurate processing capability be determined for each firewall machine before it is included in the architecture. The load-balancing algorithm will work only as well as the information provided to it allows. So, a standardized testing method would be helpful. This appendix addresses baseline processing capability measurements.

The concern with processing capability measurements in the parallel firewall architecture is not that every firewall machine be tested before it is included in the architecture, but rather that each machine's configuration be tested. The processor, bus speed, amount of memory, network interface card, operating system, and firewall software product are the types of factors that govern a particular firewall machine's processing capability. Once a given configuration is tested, multiple machines using that configuration may be used in the firewall architecture without further testing. One exception that requires retest of all configurations is where the speed of the test network components was a limiting factor in testing but have been upgraded. For example, if the test network is upgraded from Fast Ethernet to Gigabit Ethernet, then all platform testing may need to be redone.

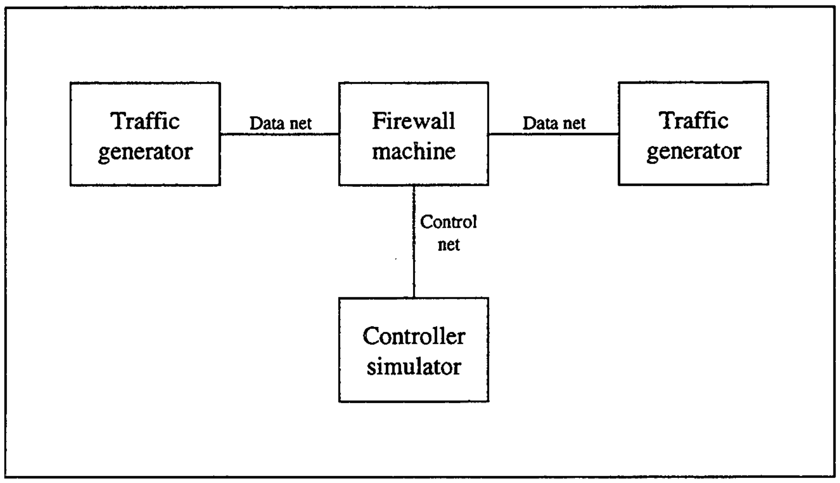

Figure 1 Test network

Figure 1 shows the recommended test network configuration. It has endpoints that generate network traffic destined for each other. All traffic passes through the firewall machine configuration being tested. An accurate measurement requires that all programs that will run in the firewall architecture must run in the test network. Thus, a machine simulating the controller on the control network is used so that the firewall machine may pass its load updates to the controller.

One situation that could have a major impact on the testing process is the case where different protocols (e.g. HTTP, SMTP, FTP) have significantly different processing times in the firewall. In this case, knowledge of the typical network traffic composition, by protocol, of the customer site would be needed for testing. The testing would emulate the given traffic composition.

The same test program should be used to test all machine configurations. However, this rule may need to be applied to varying degrees. If situation of differing protocol processing times occurs, then machine configurations for a given customer would be tested using the same program. If no significant difference in processing time exists, then all machines for all customer sites may be able to be tested together. While there is no one correct answer for the content of the test program, the following recommendations should be considered:

- Test for varying number of simultaneous TCP connections
- Test for varying sizes of data streams per connection (e.g. 1K, 10K, 50K, 100K of data)
- Send varying amounts of connectionless traffic with the TCP traffic
- Measure data rate for the firewall machine by timing end-to-end data transfer

LIST OF REFERENCES

1. Department of the Navy, "Information Technology Guidelines", Version 98-1.1, 15 June 1998, p. 3-12.

2. Department of the Navy, "Naval Virtual Internet Functional Architecture and Concepts of Operations," Draft for Comment (Revision7b), 11 December 1997, pp. 32-32.

3. J. Moy, *OSPF Version 2*, RFC 2328, The Internet Society, April 1998.

4. T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-purpose Distributed Computing Systems," IEEE Trans. on Soft. Eng., vol. 14, no. 2, pp. 141-154, 1988.

5. S. Manoharan, "Taxonomy for Assignment in Parallel Processor Systems," in Proceedings. 5th Annual European Computer Conference CompEuro'91, pp. 143-147, 1991.

6. Hesham El-Rewini, Theodore G. Lewis, Hesham H. Ali, *Task Scheduling in Parallel and Distributed Systems*, PTR Prentice Hall, Englewood Cliffs, NJ, 1994, pp. 8, 9, 25.

7. Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal, "Load Distributing for Locally Distributed Systems," in Computer, vol. 25, no. 12, Dec 1992, pp. 33-44.

8. Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," in IEEE Trans. On Soft. Eng., vol. SE-12, no. 5, May 1986, pp. 662-675.

9. M. Stumm, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster," Proc. Second Conf. Computer Workstations, IEEE CS Press, Los Alamitos, CA, 1988, pp. 12-22.

10. F. Douglas and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," Software - Practice and Experience, vol. 21, no. 8, August 1991, pp. 757-785.

11. M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor – A Hunter of Idle Workstations," Proc. Eighth Int'l Conf. Distributed Computing Systems, IEEE CS Press, Los Alamitos, CA, 1988, pp. 104-111, [http://www.cs.wisc.edu/condor].

12. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," Proc. 11th Int'l Conf. Distributed Computing Systems, IEEE CS Press, Los Alamitos, CA, 1991, pp. 336-343.

13. Katherine M. Baumgartner and Benjamin W. Wah, "GAMMON: A Load Balancing Strategy With Multiaccess Networks," IEEE Trans. on Computers, vol. 38, no. 8, August 1989, pp. 1098-1109.

14. Debra A. Hensgen, Taylor Kidd, et al, "An Overview of MSHN: The Management System for Heterogeneous Networks," a work in progress, 1999.

15. I. Foster, C. Kesselman, "The Globus Project: A Status Report," Proc. IPPSISPDP '98 Heterogeneous Computing Workshop, pp. 4-18, 1998, [http://www.globus.org/documentation/paper.html].

16. David Gelernter, "Generative Communication in Linda", ACM Transactions on Programming and Systems," Vol. 7, No. 1, Jan 1985, pp.80-112.

17. V. Sunderam, J. Dongarra, A. Geist, R. Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, Trends," Parallel Computing, Vol. 20, No. 4, April 1994, pp. 531-547, [http://www.netlib.org/pvm3/ncwn.html].

18. "The Message Passing Interface," International Journal of Supercomputing Applications, Vol. 8, No. 3/4, Fall/Winter 1994, [http://www.netlib.org/mpi/mpi-report.ps].

19. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

20. Edsger W. Dijkstra, "The Structure of "THE" – Multiprogramming System", Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 341-346.

21. W. Richard Stevens, *UNIX Network Programming, 2nd edition, Vol. 1, Networking APIs: Sockets and XTI*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

22. Carlos F. Borges, Naval Postgraduate School, Monterey, CA. Interview, 04 November 1998.

23. Roland J. Schemers III, "lbnamed: A Load Balancing Name Server in Perl", LISA IX, Monterey, CA, 17-22 September 1995, [http://www.stanford.edu/~schemers/docs/lbnamed/lbnamed.html].

24. Department of the Navy, OPNAVINST 5239.1B (draft), December 1997, para. 6.

25. Paul Albitz and Cricket Liu, *DNS and BIND, 3rd ed.*, O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA, chapter 6.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, CA 93943-5101	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Cynthia Irvine Computer Science Department, Code CS/Ic Naval Postgraduate School Monterey, CA 93943	2
5.	Dr. Geoffrey Xie Computer Science Department, Code CS/Xg Naval Postgraduate School Monterey, CA 93943	1
6.	Mr. Joseph O'Kane National Security Agency Research and Development Building R23 9800 Savage Road Fort Meade, MD 20755-6000	1

7. CAPT Dan Galik 1
Space and Naval Warfare Systems Command
PMW 161
Building OT-1, Room 1024
4301 Pacific Highway
San Diego, CA 92110-3127

8. Commander, Naval Security Group Command.. 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
ATTN: Mr. James Shearer

9. Mr. George Bieber.. 1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230

10. CDR Chris Perry 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

11. Mr. John Mildner 1
Director of Technical Operations
Code 72A
SPAWAR Systems Center Charleston
P. O. Box 190022
North Charleston, SC 29419

12. Mr. Scott Ledwell.. 1
Network Systems Security Branch Head
Code J724
SPAWAR Systems Center Charleston
P. O. Box 190022
North Charleston, SC 29419-9022

13. LT Lee Joyner..1
22286 Capote Dr.
Salinas, CA 93908