The HAS Architecture: A Highly Available and Scalable Cluster Architecture for Web Servers

Ibrahim Haddad

A Thesis in the Department of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

May 2006

©Ibrahim Haddad, 2006

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-16284-2
Our file Notre référence
ISBN: 978-0-494-16284-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

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

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

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

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

The HAS Architecture: A Highly Available and Scalable Cluster Architecture for Web Servers

Ibrahim Haddad, Ph.D.

Concordia University, 2006

This dissertation proposes a novel architecture, called the HAS architecture, for highly available and scalable web server clusters. The proof-of-concept of the HAS architecture was validated for performance and scalability, tested for its failover mechanisms, and externally modeled and simulated to study the failure and repair behavior and to calculate the availability of the cluster. The HAS architecture is able to maintain the base line performance per cluster processor, for up to 16 traffic processors in the cluster, achieving close to linear scalability. The architecture supports dynamic traffic distribution, supports heterogeneous cluster nodes, provides a mechanism to keep track of available cluster nodes, and offers connection synchronization to ensure that web connections survive software or hardware failures. Furthermore, the architecture supports different redundancy models and high availability capabilities such as Ethernet and NFS redundancy, and node level redundancy that contribute in increasing the availability of the service, and in eliminating single points of failure.

This dissertation presents current methods for scaling web servers, discusses their limitations, and investigates how clustering technologies can help overcome some of these challenges and enable the design of scalable web servers based on a cluster of workstations. It examines various ongoing research projects in the academia and the industry that are investigating scalable and highly available architectures for web servers. It discusses their scope, architecture, provides a critical analysis of their work, and presents their contributions to this dissertation.

This dissertation contributes the HAS architecture, a highly available and scalable architecture for web servers, and offers contributions in areas of scalability, maintaining baseline performance, and availability.

Acknowledgments

I would like to express my gratitude to my supervisor Professor Greg Butler for his guidance, support, patience, and constructive criticism.

I would like to thank the members of my committee from Concordia University, Professor J. William Atwood, Dr. Ferhat Khendek, and Professor Thiruvengadam Radhakrishnan. The feedback I received from members of my committee as early as during my doctoral proposal was very important and had influence on the direction of the work.

I also would like to thank Dr. Steve Goddard, Associate Professor, from the Department of Computer Science and Engineering at the University of Nebraska-Lincoln for taking time out from his schedule to serve as my external reader, and for his valuable feedback and recommendations that helped improve the work.

I would like to acknowledge the support I received from Ericsson Research for the access to the research lab in Montréal, Canada, where I performed the benchmarking tests. I also would like my current employer, the Open Source Development Labs, for allowing me to work on my thesis during my employment.

Lastly, I would like to thank and express my gratitude to my wife, my parents, my brother and sister, for their encouragement and support.

Ibrahim Haddad

May 2006

Table of Contents

Abstract iii

Acknowledgments iv

Table of Contents v

List of Figures viii

List of Tables xi

Chapter 1 Introduction 1

1.1 Motivations 1

1.2 Hypothesis 2

1.3 Definitions 3

1.4 Scope of the Study 7

1.5 Thesis Contributions 9

1.6 Dissertation Roadmap 10

Chapter 2 Background and Related Work 11

2.1 Cluster Computing 11

2.2 SMP versus Clusters 17

2.3 Cluster Software Components 17

2.4 Cluster Hardware Components 18

2.5 Benefits of Clustering Technologies 18

2.6 The OSI Layer Clustering Techniques 20

2.7 The RR-DNS Approach 26

2.8 Discussion of Software Approaches to Clustering 27

2.9 Discussion of Hardware Approaches to Clustering 28

2.10 Scalability in Internet and Web Servers 30

2.11 Scalability in Telecommunication Servers 30

2.12 How Users Experience Scalability 31

2.13 Principles of Scalable Architecture 32

2.14 Overview of Related Work 33

2.15 Related Work: In-depth Examination 36

Chapter 3 Highly Available and Scalable Web Server Cluster Architecture 63

3.1 Summaries of Contributions 63

3.2 The HAS Architecture 64

3.3 HAS Architecture Components 67
3.4 HAS Architecture Tiers 71
3.5 Characteristics of the HAS Cluster Architecture 72
3.6 Availability and Eliminating Single Points of Failures 76
3.7 Overview of Redundancy Models 79
3.8 HA Tier Redundancy Models 80
3.9 SSA Tier Redundancy Models 83
3.10 Storage Tier Redundancy Models 83
3.11 Redundancy Model Choices 83
3.12 The States of a HAS Cluster Node 85
3.13 Example Deployment of a HAS Cluster 87
3.14 The Physical View of the HAS Architecture 88
3.15 The Physical Storage Model of the HAS Architecture 90
3.16 Types and Characteristics of the HAS Cluster Nodes 96
3.17 Local Network Access 98
3.18 Master Nodes Heartbeat 99
3.19 Traffic Nodes Heartbeat using the LDirectord Module 101
3.20 CVIP: A Cluster Virtual IP Interface for the HAS Architecture 103
3.21 Connection Synchronization 107
3.22 Traffic Management 110
3.23 Ethernet Redundancy Daemon Contribution 120
3.24 Scenario View of the Architecture 122
3.25 Network Configuration with IPv6 137
Chapter 4 Evaluation of the HAS Architecture 139
4.1 Benchmarking Hardware Environment 139
4.2 Benchmarking Tool and Workload 139
4.3 Benchmarking Network Environment 143
4.4 Determining Baseline Performance of a Standalone Traffic Node 143
4.5 Benchmarking the HAS Architecture Proof-of-Concept 146
4.6 Test-4: Experiments with an 18-node HAS Cluster 147
4.7 Overall Results and Discussion 148
4.8 Testing of Failover Mechanisms 150

4.9 Architecture Modeling and Availability Prediction 154
Chapter 5 Contributions, Future Work, and Conclusion 162
5.1 Contributions – The HAS Architecture 162
5.2 Future Work 166
5.3 Conclusion 168
Bibliography 169
Glossary 176

List of Figures

Figure 1: Web server components ... 3
Figure 2: Generic cluster architecture ... 7
Figure 3: The SMP architecture [33] ... 12
Figure 4: The MPP architecture [33] ... 13
Figure 5: Cluster architectures with and without shared disks ... 14
Figure 6: A cluster node stack ... 15
Figure 7: The L4/2 clustering model [69] ... 21
Figure 8: Traffic flow in an L4/2 based cluster [69] ... 21
Figure 9: The L4/3 clustering model [69] ... 22
Figure 10: The traffic flow in an L4/3 based cluster [69] ... 23
Figure 11: The process of content-based dispatching – L7 clustering model [69] ... 24
Figure 12: RR-DNS approach ... 27
Figure 13: Using a router to hide the web cluster ... 29
Figure 14: Hierarchical redirection-based web server architecture [37] ... 38
Figure 15: Redirection mechanism for HTTP requests ... 39
Figure 16: The web farm architecture with the dispatcher as the central component [2] ... 41
Figure 17: The SWEB architecture [2] ... 44
Figure 18: The functional modules of a SWEB scheduler in a single processor [2] ... 45
Figure 19: The LSMAC implementation [20] ... 46
Figure 20: The LSNAT implementation [20] ... 47
Figure 21: The architecture of the IP sprayer [83] ... 48
Figure 22: The architecture with the HACC smart router [83] ... 48
Figure 23: The two-tier server architecture [19] ... 52
Figure 24: The flow of the web server router [19] ... 52
Figure 25: The architecture of the LVS NAT method [74] ... 55
Figure 26: The architecture of the LVS DR method [74] ... 56
Figure 27: Benchmarking results of NAT versus DR ... 57
Figure 28: Scalability of LVS clusters consisting of up to 12 nodes running Apache ... 59
Figure 29: The HAS architecture ... 66
Figure 30: Built-in redundancy at different layers of the HAS architecture ... 77
Figure 31: The process of the network adapter swap ... 79
Figure 32: The 1+1 active/standby redundancy model ... 81
Figure 33: Illustration of the failure of the active node ... 81
Figure 34: The 1+1 active/active redundancy model ... 82
Figure 35: the redundancy models at the physical level of the HAS architecture ... 84
Figure 36: The state diagram of the state of a HAS cluster node ... 86
Figure 37: The state diagram including the standby state ... 87
Figure 38: A HAS cluster using the HA NFS implementation ... 88
Figure 39: The physical view of the HAS architecture ... 89
Figure 40: The *no-shared storage* model ... 91
Figure 41: The HAS storage model using a distributed file system ... 92
Figure 42: The NFS server redundancy mechanism ... 92
Figure 43: DRDB disk replication for two nodes in the 1+1 active/standby model ... 95
Figure 44: A HAS cluster with two specialized storage nodes ... 95
Figure 45: The master node stack ... 97
Figure 46: The traffic node stack ... 98

Figure 47: The redundant LAN connections within the HAS architecture 99
Figure 48: The topology of the heartbeat Ethernet broadcast 100
Figure 49: Generic CVIP configuration 104
Figure 50: Network terminations 105
Figure 51: Step 1 - Connection Synchronization 108
Figure 52: Step 2 - Connection Synchronization 108
Figure 53: Step 3 - Connection Synchronization 109
Figure 54: Step 4 - Connection Synchronization 109
Figure 55: Peer-to-peer approach 110
Figure 56: The direct routing approach – traffic nodes reply directly to web clients 112
Figure 57: The CPU information available in /proc/cpuinfo 114
Figure 58: The memory information available in /proc/meminfo 115
Figure 59: The interaction between the traffic client and the traffic manager 118
Figure 60: Interaction and dependencies of software modules 119
Figure 61: The sequence diagram of a successful request 123
Figure 62: A traffic node reporting its load index to the traffic manager 124
Figure 63: A traffic node joining the HAS cluster 125
Figure 64: The boot process of a diskless node 126
Figure 65: The boot process of a traffic node with disk 127
Figure 66: The process of rebuilding a node with disk 128
Figure 67: The process of upgrading the kernel and application server on a traffic node 129
Figure 68: The sequence diagram of upgrading the hardware on a master node 130
Figure 69: The sequence diagram of a master node becoming unavailable 131
Figure 70: The NFS synchronization occurs when a master node becomes unavailable 131
Figure 71: The sequence diagram of a traffic node becoming unavailable 132
Figure 72: The scenario assumes that node C has lost network connectivity 133
Figure 73: The scenario of an Ethernet port becoming unavailable 133
Figure 74: The sequence diagram of a traffic node leaving the HAS cluster 134
Figure 75: The LDirectord restarting an application process 135
Figure 76: The network becomes unavailable 136
Figure 77: The sequence diagram of the IPv6 autoconfiguration process 138
Figure 78: The architecture of the WebBench benchmarking tool 140
Figure 79: Adding mixes to the WebBench test 141
Figure 80: Configurable WebBench parameters 142
Figure 81: A screen capture of the WebBench software showing 379 connected clients 142
Figure 82: The network setup inside the benchmarking lab 143
Figure 83: The results of benchmarking a standalone processor (requests per second) 144
Figure 84: The throughput of a standalone processor expressed in KB/s 145
Figure 85: The number of failed requests per second on a standalone processor 145
Figure 86: The four benchmarked configurations 146
Figure 87: The results of benchmarking an 18 nodes HAS cluster 148
Figure 88: The results of benchmarking the HAS architecture prototype 149
Figure 89: The scalability chart of the HAS architecture prototype 149
Figure 90: The possible connectivity failure points 150
Figure 91: The tested setup for data redundancy 153
Figure 92: The architecture of a Beowulf cluster 155
Figure 93: The HA-OSCAR architecture 156
Figure 94: The modeled HA-OSCAR architecture, showing the three sub-models 157

Figure 95: Availability improvement analysis of HA-OSCAR [41] 159
Figure 96: Steady State Availability of 99.993% [43] 160

List of Tables

Table 1: Expected service availability per industry type [36] 5
Table 2: Web server performance metrics 6
Table 3: Classification of clusters by usage and functionality 16
Table 4: Characteristics of SMP and cluster systems 17
Table 5: Comparison of the clustering techniques operating at the OSI layer [69] 26
Table 6: The results of benchmarking with Apache 58
Table 7: Evaluation of related work 61
Table 8: Redundancy models per each tier of the HAS architecture 85
Table 9: Supported redundancy models per each tier in the HAS architecture prototype 85
Table 10: The changes made to the Linux kernel to support NFS redundancy 93
Table 11: Example list of traffic nodes and their load index 117
Table 12: The performance results of one standalone processor 144
Table 13: The summary of the benchmarking results of the HAS architecture prototype 148
Table 14: Input parameters for the HA-OSCAR model [43] 158
Table 15: System availability for different configurations [43] 160
Table 16: Status of web clusters 165

Chapter 1
Introduction

1.1 Motivations

The growth of the Internet in the last few years has given rise to a vast range of new online services. Current Internet services span a diverse range of categories that require significant computational and I/O resources to process each request. Furthermore, exponential growth of the Internet population is placing unprecedented demands upon the scalability and robustness of these services [1][11]. Yahoo!, for instance, receives over 1.2 billion page views a day [20], while AOL's web caches service over 10 billion hits daily [1].

Internet services have become critical both for driving large businesses as well as for personal productivity. Global enterprises are increasingly dependent upon Internet-based applications for e-commerce, supply chain management, human resources, and financial accounting, while many individuals consider e-mail and web access to be indispensable. This growing dependence upon Internet services underscores the importance of their availability, scalability, and ability to handle large loads. Popular web sites such as eBay [51], Excite [27], and E*Trade [8] have, at times, experiences costly and high profile outages during periods of high load. As more people rely upon the Internet for managing financial accounts, paying bills, and potentially even voting in elections, it is increasingly important that these services are available at all times, perform well under high demand, and are robust to accommodate to rapid changes in load. Furthermore, the variations of load experienced by web servers intensify the challenges of building scalable and highly available web servers. It is common to experience more than 100-fold increases in demand when a web site becomes popular [47].

When the terrorist attacks on New York City and Washington DC occurred on September 11, 2001, Internet news services reached unprecedented levels of demands. CNN.com, for instance, experienced a two-and-a-half hour outage with load exceeding 20 times the expected peak [47]. Although the site team managed to grow the server farm by a factor of five by borrowing machines from other sites, this arrangement was not sufficient to deliver adequate service during the load spike. CNN.com came back online only after replacing the front page with a text-only summary in order to reduce the load [10]. Web sites are also subject to sophisticated denial-of-service attacks, often launched simultaneously from thousands of servers, which can knock a service out of commission. Denial-of-service attacks have had a major impact on the performance of sites such as Yahoo! and whitehouse.gov [10]. The number of

concurrent sessions and hits per day to Internet sites translates into a large number of I/O and network requests, placing enormous demands on underlying resources.

In recent years, the interest in and the deployment of scalable and highly available web servers has increased rapidly for the wide potential such systems offer. The progress of web servers has been feasible, driven by advances in network, software, and computer technologies. However, there are still many challenges to resolve.

Scalability, availability, and performance are the biggest challenges facing web servers providing interactive services for a large user base, and are crucial factors for the success or failure of an online service.

1.2 Hypothesis

The work in thesis evolves around the possibility of a highly available architecture for web server clusters that is capable of linearly scaling for up to 16 nodes. The goal of this thesis is therefore to design a highly available and scalable cluster architecture for web servers, and to evaluate the architecture for scalability, performance, and high availability. The main motivations are to increase the cluster system capacity, availability, and scalability.

> ***Hypothesis:*** *Can we have an architecture for web server clusters that is highly available, providing over four nines availability, and capable of scaling linearly for up 16 nodes while maintaining baseline performance per each cluster node?*

The architecture has to be flexible and modular, capable of linearly scaling as we add more processors, while maintaining the baseline performance and without affecting the availability of the service. As a result, as we double the number of processors in the web cluster, we expect to double the number of requests per second served, and to maintain the level of throughput per processor. The emphasis of the study is therefore on scalability through clustering, effective resource utilization, and efficient distribution of user requests among the cluster nodes. The highly available web server cluster should also adapt itself dynamically to different numbers of users and amounts of data, with minimal effect on performance.

The goal of the architecture is to achieve maximum scalability and performance levels for up to 16 processors, with each processor in the cluster handling N requests per second. N refers to the baseline performance in terms of requests per second and throughput in terms of KB/s.

The scaling limitation for up to 16 processors is set for two main reasons. First, based on our initial experiments with web clusters [22], we demonstrated that web clusters suffer from scalability problems starting with small clusters that consist of four processors. Our experimental work confirmed performance degradation and loss of scalability as we increased the number of nodes in a clustered web server from four to eight, 10, and then to 12 nodes. The second reason for choosing 16 processors as the upper limit relies on the facts we collected from our literature review (Section 2.15). Sections 2.14 and 2.15 present and discuss ongoing projects that focus on scalability and performance of web servers from an architectural point of view, and experimenting with web clusters that consist of eight processors and fewer. In addition, with an upper limit of 16 processors, we can demonstrate and validate the performance and scalability of the architecture in the lab without resorting to building a theoretical model and simulating it.

1.3 Definitions

This section defines the concepts and terminologies used in this thesis.

Web Server

A *web server* is a program that follows the client/server model, responds to incoming TCP connections, and provides contents to web users over the HTTP protocol.

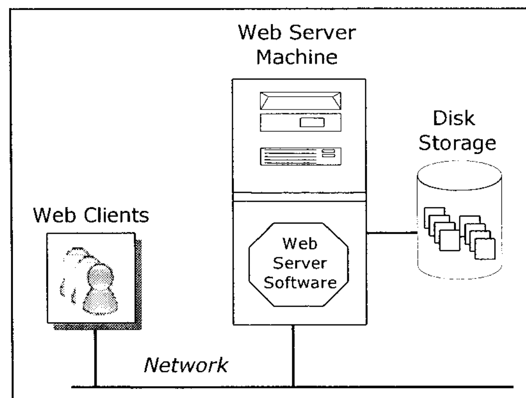


Figure 1: Web server components

Figure 1 illustrates the three core components of a web server: the web server software, a computer system with a connection to the Internet, and the information or documents that are available for serving. In this dissertation, the term *web server* refers to the whole entity, computer platform, server software, and the documents.

Performance

The *performance* of a web server is measured in terms of successful web requests per second.
An objective assessment of the server performance can be based on standardized benchmarking tool and a standardized workload. In this thesis, we measure the baseline performance of a single node web server to determine its capacity and then benchmark the web cluster to determine if the cluster nodes are capable of maintaining the baseline performance as we scale the number of nodes in the cluster.

Scalability

Scalability is the ability to utilize additional resources with a predictable increase in performance, without requiring architectural changes or technology changes, and without imposing additional overhead.
Linear scalability means that if we increase the number of nodes by a factor of n, then the throughput of successful number of requests per second increases by a factor of n.
The common strategy in measuring server scalability is to measure throughput as the number of users or traffic increases and identify important trends. For instance, we measure the throughput of the server with 100 concurrent transactions, then with 1,000, and then with 10,000 transactions. We then examine how throughput changes and observe how it compares with linear scalability. This comparison gives us a measure of the scalability of the architecture.
We consider an acceptable decline in scalability to be less than 10%.

Availability

Availability is the amount of time that a system or service is provided in relation to the amount of time the system or service is not provided. *Availability* is expressed as follows:

$$A = \left(\frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \right) * 100,$$

where A is the percentage of availability, MTBF is the mean time between failures and MTTR is the mean time to repair or resolve a particular problem. *MTTF* is the interval of time in which the system can provide service without failure. *MTTR* is the interval of time it takes to resume service after a failure has been experienced.
We calculate availability A as the percentage of uptime for a given period, taking into account the time it requires for the system to recover from unplanned failures and planned upgrades. When MTTR approaches zero, A increases towards 100 percent. As the MTBF value increases, MTTR has less impact

on A. There are two possible ways to increase availability: increasing MTBF and decreasing MTTR. Increasing MTBF involves improving the quality or robustness of the software and using redundancy to eliminate single points of failures. As for decreasing MTTR, it involves focus on the implementation of the system software to streamline and accelerate fail-over, respond quickly to fault conditions, and make faults more granular in time and scope.

A *highly available system* is capable of providing over four nines of availability.

Four nines availability is as 99.99% of service availability. It is equivalent to 52 minutes and 33 seconds a year of total planned and unplanned downtime of the service provided by the system

Availability	Downtime per year	Example Areas for Deployments
90.00%	36 days 12 hours	Personal clients
99.00%	87 hours 36 minutes	Entry-level businesses
99.90%	8 hours 46 minutes	Internet Service Providers, Mainstream businesses
99.99%	52 minutes 33 seconds	Data centers
99.999%	5 minutes 15 seconds	Telecom system, medical, banking
99.9999%	31.5 seconds	Military defense, carrier grade routers

Table 1: Expected service availability per industry type [36]

Table 1, from [36], presents the various levels of high availability as classified by the industry.

Validation, Testing, and Performance

Validation provides an objective assessment that the architecture meets the defined requirements. In this dissertation, we demonstrate the scalability and performance of the architecture by building a proof-of-concept prototype of the architecture and benchmarking it using a standardized benchmarking tool and workload.

Testing is the process of evaluating software modules under specific conditions and recording the results to identify differences between expected and actual results. The goal of testing is to ensure that the software modules work properly and to try to discover every conceivable fault or weakness.

Web server performance refers to the efficiency of a web server when responding to user requests according to defined benchmarks. We measure web server performance using two common metrics, the number of successful requests per second, and the throughput as KB/s.

Benchmark

A *benchmark* is a publicly defined procedure, designed to evaluate the performance of a system using a well-defined and standardized workload model. Benchmarking helps evaluate the system capacity and response time with respect to an existing, expected, or standardized workload.

A *web server benchmark* is a mechanism to generate a controlled stream of web requests with standard metrics. It aims at reproducing as accurately as possible the characteristics of real traffic patterns, and reports the results. A *web server benchmark* consists of a combination of standardized tests that consist of a mechanism to generate a controlled stream of web requests to the web server, and reports the results with standard metrics.

Metric Name	Description
Throughput	The rate at which data is sent through the network, expressed in Kbytes per second (KB/s)
Connection rate	The number of connections per second
Request rate	The number of client requests per second
Reply rate	The number of server responses per second
Error rate	The percentage of errors of a given type
DNS lookup time	The time to translate the hostname into the IP address
Connect time	The time interval between sending the initial SYN and the last byte of a client request and the receipt of the first byte of the corresponding response
Latency time	Latency is an expression of how much time it takes for a packet of data to get from one designated point to another
Transfer time	The time interval between the receipt of the first response byte and the last response byte
Session time	The sum of all web page response times and user think time in a user session

Table 2: Web server performance metrics

Table 2 presents the metrics for web server performance, which include throughput, connection rate, request rate, reply rate, error rate, connect time, latency time, and transfer time.

1.3.1 Cluster Definitions

A *cluster* is a group of separate computers that are interconnected, and are used as a single computing entity to provide a service or run an application for the purposes of scalability, high availability, or high performance.

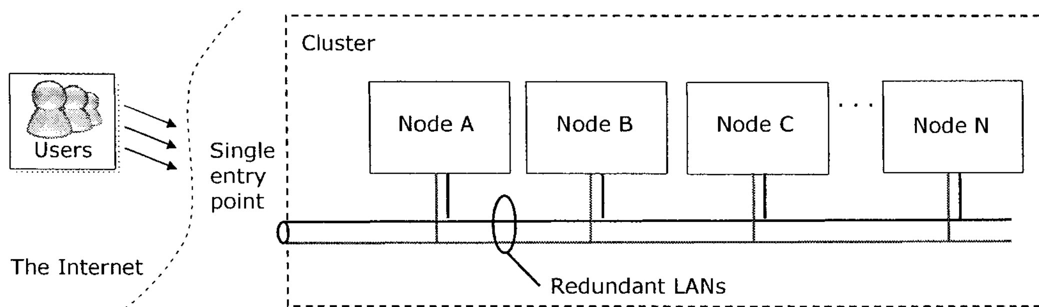


Figure 2: Generic cluster architecture

Figure 2 illustrates a generic cluster architecture, which consists of multiple standalone nodes that are connected through redundant links, and providing a single entry to the cluster. End users are not aware that they are interacting with a cluster and they are not aware as well where the application is running.

A fault tolerant cluster is a cluster that has capabilities, such as redundancy, fault discovery and recovery that allow the cluster to minimize the impact of faults (software or hardware) on the offered services.

A *node*, also called *cluster node* or *cluster processor*, denotes a whole computer in a cluster. An *active node* is a node that is providing a service. A *standby node* is a node that is not currently providing service but prepared to take over the active state when an active node fails.

A *single point of failure* (SPOF) occurs when any single component or communication path within a computer fails and lead to the failure of the system or the failure of the offered service.

1.4 Scope of the Study

This section presents the scope of the study with respect to architecture, type of servers covered by the study, types of web site content, architectural model, scalability, and high availability. The scope is the architecture and its parameters and excludes external factors such as the performance of file systems, networks, and operating systems.

1.4.1 Goal

The goal of this dissertation is to propose and evaluate an architecture for scalable and highly available web server clusters. The architecture should allow the following properties: fast access, linear scalability for up to 16 processors, transparency, and high availability. This dissertation investigates web servers as specific case of an Internet server.

1.4.2 Web Site Content

We limit the experimental work with web server scalability testing to static web site contents, which does not include dynamic transaction or web searches.

1.4.3 Architecture and Servers

The goal with this work is to reduce architecture complexities by providing a generic architecture that can handle massive concurrency demands and deal gracefully with large variations in load. The scope of the proposed architecture is limited to systems that follow the client/server model and run applications characterized by short transactions, short response time, a thin control path, and static delivery.

This dissertation does not address multimedia servers, streams, sessions, states, and applications servers. In addition, it does not address nor try to fix problems with networking protocols.

High performance computing (HPC) is not in the scope of the study. HPC is a branch of computer science that concentrates developing parallel processing algorithms that divide large computational task into small pieces so that separate processors can execute simultaneously. Architectures in this category focus on maximizing compute performance for floating point operations. This branch of computing is unrelated to the dissertation.

The architecture targets servers providing services over the Internet with the characteristics previously mentioned. The architecture applies to systems with short response times such as, but not exclusively, web servers, Authentication Authorization and Accounting (AAA) servers, Policy servers, Home Location Register (HLR) servers, Service Control Point (SCP) servers, without specialized extensions at the architectural level.

1.4.4 High Availability

This work targets the architecture for highly available web servers. From this perspective, the scope of this dissertation covers techniques that allow us to support high availability capabilities to maximize

service availability for the end users. There are two sub-categories: HA stateless, with no saved state information and HA stateful, with state information that allows the web application to maintain sessions across a failover. Our scope focuses on the HA stateless web applications.

1.5 Thesis Contributions

To the best of our knowledge, this work contributes the first highly available and scalable architecture for web server clusters that follows the building block approach and demonstrates close to linear scaling for up to 16 nodes, maintains over 96% of baseline performance, and supports high availability at different layers of the cluster leading to continues service availability.

The HAS architecture supports multiple redundancy models in each tier of the architecture and independently from other tiers. The HAS architecture uses common-off-the-shelf hardware and software and does not require any specialized hardware or software.

The HAS architecture contributes a dynamic traffic distribution mechanism that monitors the load of the traffic nodes using multiple metrics, and uses this information to distribute incoming traffic among the traffic nodes. The distribution mechanism does not assume that all nodes in the cluster have the same hardware configuration, and achieves efficient resource utilization taking into consideration the nodes hardware configuration. Two contributions provide these functionalities: the traffic client daemon and the traffic manager daemon. Furthermore, the HAS traffic distribution scheme integrates a keep-alive mechanism, which allows the master node to know when a traffic node is available for service and when it is not available because of either software or hardware problems.

The HAS architecture supports mechanisms to detect failures and trigger recovery for traffic nodes, master nodes, file system, Ethernet cards, traffic client, web server software, and ongoing connections, and provide correction action when the failures are detected. The HAS architecture proof-of-concept contributed a modified version of the NFS server with HA extension to provide storage to the HAS cluster nodes, and a specialized mount program that allows mounting of two redundant NFS servers over the same mount point. Furthermore, it contributes the Ethernet Redundancy Daemon that that monitors the link status of the primary Ethernet port and switches control to the second Ethernet port upon the failure of the first port.

The HAS architecture provides continuous service through supporting online operating system and software upgrade, and provides the capability to synchronize connections, and continue servicing ongoing connections even in the event of software or hardware failures.

The HAS architecture proof-of-concept supports both IPv4 and IPv6 and was testing with traffic over both IP protocols.

This work made significant contributions to the HA-OSCAR project [45], whose architecture is based on the work presented in this dissertation. Furthermore, the Carrier Grade industry initiative at the Open Source Development Labs has adopted the HAS architecture as the base standard architecture for carrier grade servers running telecommunication applications.

Other contributions include benchmarking existing approaches, providing enhancements to their capabilities, adding functionalities to existing system software, and providing best practices for building benchmarking environment for large-scale systems.

1.6 Dissertation Roadmap

Chapter 2 focuses on three main topics: clustering technologies, scalability challenges and related work. The clustering section sets the grounds for all cluster related definitions and approaches to build clustered web servers. It presents clustering technologies and the benefits resulting from using these technologies to design and build Internet and web servers. The chapter examines clustering technologies and techniques for designing and building web servers. We argue that traditional standalone server architecture fails to address the scalability and high availability needed for large-scale Internet and web servers. We introduce software and hardware clustering technologies, their advantages, and drawbacks. Then we present the various ongoing research projects in the industry and academia, their focus areas, results, and contributions.

Chapter 3 describes the HAS architecture. It presents the architecture, its characteristics, its software components, and their characteristics. It presents the conceptual, physical, and scenario views of the architecture, the supported redundancy models, the traffic distribution scheme, the cluster virtual IP interface, and the dependencies between the various components.

Chapter 4 presents the evaluation of the HAS architecture through benchmarking the performance and scalability of the architecture, testing its failover mechanisms, and modeling and simulation for high availability. The results demonstrate that the HAS architecture is able to reach close to linear scaling for up to 16 processors and to maintain 96% of the baseline performance per cluster node. Furthermore, the modeling and simulation results demonstrate that the architecture is capable of achieving over four nines availability.

Chapter 5 presents the contributions and future work.

Chapter 2
Background and Related Work

2.1 Cluster Computing

Cluster computing has become increasingly popular for a number of reasons, which include low cost, high performance of commodity-off-the-shelf (COTS) hardware, availability of rapidly maturing proprietary and open source software components to support high performance and high availability applications. Furthermore, the high-speed networking and the improved microprocessor performance have positioned networks of workstations as a more appealing vehicle for distributed computing compared to the Symmetric Multiprocessor (SMP) and Massively Parallel Processor (MMP) systems. As a result, clusters built using COTS hardware and software are playing a major role in redefining the concept of supercomputing and are becoming a popular alternative to SMP and MPP systems. However, there are still several challenges in the areas of performance, availability, manageability, and scalability that are key issues.

A web server is an example of an application that requires more computing power than what a single computer can provide, and is a candidate to run on a computer cluster. A viable and cost-effective web cluster solution consists of connecting multiple computers together and coordinating their efforts to serve web traffic. The resulting system is a distributed web server that responds to incoming traffic and processes them on multiple processors.

The following sub-sections introduce the three general classes of distributed systems: SMP, MPP, and clusters, discuss their advantages and drawbacks, and identify the model that is more suitable for running web servers.

2.1.1 Symmetric Multiprocessors (SMP)

An SMP machine consists of tightly coupled series of identical processors, operating on a single shared bank of memory. Figure 3 illustrates the architecture of an SMP system. The figure is a contribution from [33] with modifications to illustrate the memory and IO modules. There are no multiple memories, input and output (I/O) systems, or operating systems. SMP systems are *share-everything* systems, where each processor has access to the shared memory system and all of the attached devices and peripherals of the system, perform I/O operations, and interrupt other processors [63][75]. SMP systems have a master

processor at boot time, and then the operating system starts up the second (or more) processor(s) and manages access to the shared resources among all the processors. A single copy of the operating system is in charge of all the processors. SMP systems available on the market (at the time of writing) do not exceed 16 processors, with configurations available in two, four, eight, and 16 processors.

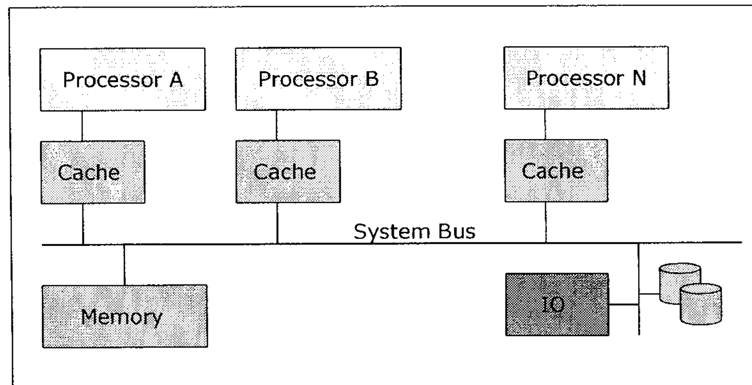


Figure 3: The SMP architecture [33]

SMP systems are not scalable because all nodes have to access the same resources. In addition, SMP systems have a limit to the number of processors they can have. They require considerable investments in upgrades, and an entire replacement of the system to accommodate a larger capacity. Furthermore, an SMP system runs a single copy of the operating system, where all processors share the same copy of the operating system data. If one processor becomes unavailable because of either hardware or software error, it leaves locks unlocked, data structures in partially updated states, and potentially, I/O devices in partially initialized states. As a result, the entire system becomes unavailable on the account of a single processor. In addition, SMP architectures are not highly available. SMP systems have several single points of failure (cache, memory, processor, bus); if one subsystem becomes unavailable, it brings the system down and makes the service unavailable to the end users.

2.1.2 Massively Parallel Processors (MPP)

As we add more processors to SMP systems and more hardware such as crossbar switches, to make memory access orthogonal, these parallel processors become Massively Parallel Processors. Figure 4, from [33], illustrates an MPP system that consists of several processing elements interconnected through a high-speed interconnection network. Each node has its own processor(s), memory, and runs a separate copy of the operating system. The key distinction between MPP and SMP systems relies in the use of

fully distributed memory. In an MPP system, each processor is self-contained with its own cache and memory chips.

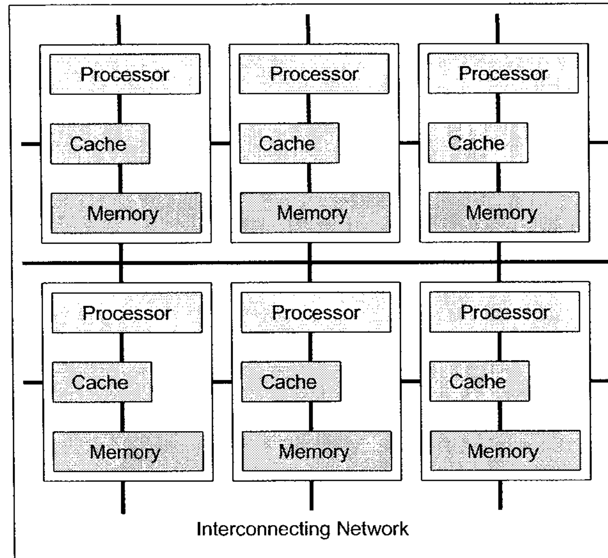


Figure 4: The MPP architecture [33]

Another distinct characteristic of MPP systems is the job scheduling subsystem. We achieve job scheduling through a single run queue. MPP systems tackle one very large computational problem at a time and serve to solve HPC problems. In addition, MPP systems suffer from the same issues as SMP systems in the areas of scalability, single points of failures, and the impact on high availability, and the need to shut down the system to perform either software or hardware upgrades.

2.1.3 Computer Clusters

In his book *"In Search of Clusters"*, Greg Pfister defines a cluster as a parallel or distributed system consisting of a collection of interconnected whole computers that appear as a single, unified computing resource [63]. In this dissertation, we consider a cluster as a group of separate computers that are interconnected, and are used as a single computing entity to provide a service or run an application for the purposes of scalability, high availability, or high performance. Our definition of a cluster is complimentary to Pfister's definition. We both agree that a cluster consists of a number of independent computers that appear as a single compute entity to the end users. End users are not aware that they are interacting with a cluster and they are not aware as well where the application is running.

Cluster nodes interconnect in different ways. Figure 5 illustrates two common variations. In Figure 5 (A), cluster nodes share a common disk repository. In Figure 5 (B), cluster nodes do not share common resources and use their own local disk for storage.

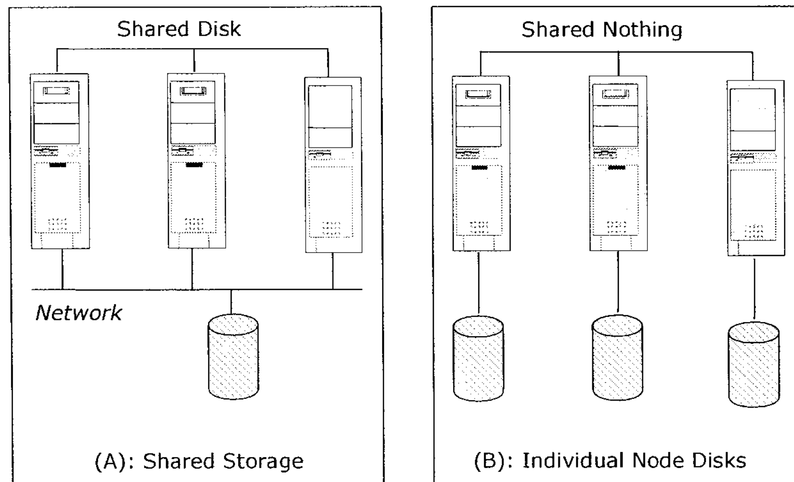


Figure 5: Cluster architectures with and without shared disks

The phrase *single, unified computing resource* in Greg Pfister definition of a cluster invokes a wide variety of possible applications and uses, and is deliberately vague in describing the services provided by the cluster. At one end of the spectrum, a cluster is nothing more than the collection of whole computers available for use by a sophisticated distributed application. At the other end, the cluster creates an environment where existing non-distributed programs can benefit from increased availability because of the cluster-wide fault masking, and increased performance because of the increased computing capacity.

A cluster is a group of independent COTS servers interconnected through a network. The servers, called cluster nodes, appear as a single system, and they share access to cluster resources such as shared disks, network file systems and the network. A network interconnects all the nodes in a cluster and it is separate from the cluster's external environment such as the local Intranet or the Internet. The interconnection network employs local area network or systems area network technology.

Clusters can be highly available because of the of build-in redundancy that prevents the presence of a single point of failure (SPOF). As a result, failures are contained within a single node. Monitoring software continually runs checks, by sending signals also called *heartbeats*, to ensure that the cluster node and the application running on are up and available. If these signals stop, then the system software initiates the failover to recover from the failure. The presumably dead or unavailable system or

application is then isolated from I/O access, disks, and other resources such as access to the network; furthermore, incoming traffic is redirected to other available nodes within the cluster. As for performance, clusters allow the possibility to add nodes and scale up the performance, the capacity, and the throughput of the cluster as the number of users or traffic increases.

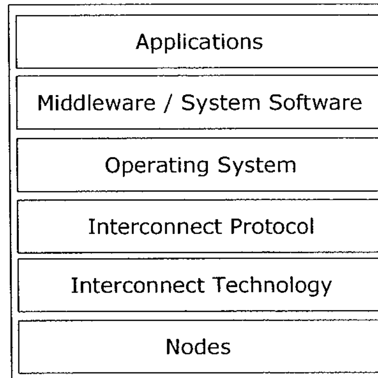


Figure 6: A cluster node stack

Figure 6 illustrates at a high level a cluster node stack. The stack consists of the processor, the interconnect technology and protocol, the operating system, the middleware or system software, and the application servers. At the lowest level is the node hardware. One level up from the Node level is the Interconnect Technology (such as Ethernet), followed by the Interconnect Protocol. Up one level from the Interconnect Protocol is the Operating System, followed by the System Software, which provides all of the support functionalities. Finally at the top level is the Applications Layer. We utilize clusters in many modes including but not limited to high performance computing, high capacity or throughput, scalability, and high availability.

Table 3 presents the different types of clusters depending on their functionalities and the type of applications they host. Clustering for scalability (Table 3, 3rd column) focuses on distributing web traffic among cluster nodes using distribution algorithms such as round robin DNS.

Clustering for high availability (Table 3, 4th column) relies on redundant servers to ensure that critical applications remain available if a cluster node fails. There are two methods for failover solutions: software-based failover solutions and hardware-based failover devices. Software-based failover detects when a server has failed and automatically redirects new incoming HTTP requests to the cluster members that are available. Hardware-based failover devices have limited built-in intelligence and require an administrator's intervention when they detect a failure.

Purpose of Clustering	*High Performance Computing (HPC)*	*Scalability*	*High Availability (HA)*	*Server Consolidation*
Goal	Maximize floating point computation performance	Maximize throughput and performance	Maximize service availability	Maximize ease of management of multiple computing resources
Description	1. Many nodes working together on a single-compute based problem. - Performance is measured as the number of floating point operations (FLOP) per second	- Many nodes working on similar tasks, distributed in a defined fashion based on system load characteristics - Performance is measured as throughput in terms of KB/s - Adding more nodes to the cluster to increase its capacity - Network oriented (throughput), or Data oriented (data transactions)	- Redundancy and failover for fault tolerance of services - Support stateless and stateful applications - Availability is measured as a percentage of the time the system is up and providing service. Section 3.6 presents the formula for calculating the availability.	- Also called Single System Image (SSI) - Provide a central management of cluster resources and treat the cluster as a single management unit.
Examples	Beowulf clusters such as MOSIX [5], Rocks [76], OSCAR [9] [34], and Ganglia [50]	Examples include the Linux Virtual Server [74], TurboLinux [80], in addition to commercial products	Examples include the HA-OSCAR project [45], in addition to commercial clustering products	Examples include the OpenSSI project [60], the OpenGFS project [59], and the Oracle Cluster File System [61], in addition to commercial database products

Table 3: Classification of clusters by usage and functionality

Many of the clustering products available fit into more than one of the above categories. For instance, some products include both failover and load-balancing components. In addition, SSI products that fit into the server consolidation category (Table 3, 5th column) provide certain HA failover capabilities.

Our goal with this dissertation is a cluster architecture that targets both scalability and high availability.

2.2 SMP versus Clusters

Table 4 summarizes the comparisons between SMP and cluster architectures. SMP systems have limited scalability, while clusters have virtually unlimited scaling capabilities since we can always continue to add more nodes to the cluster. As for high availability, an SMP system has several single points of failure; one single error can lead to a system downtime; in contrast to a cluster, where functionalities are redundant and spread across multiple cluster nodes. As for management, an SMP system is a single system, while a cluster is composed of several nodes, some of which can be SMP machines.

	Scaling	*High Availability*	*System Management*
SMP	- Limited scaling capabilities - Requires a complete upgrade of the system	- Not highly available - Single points of failure in hardware and operating system	- Single system - Single image of the operating system
Clusters	- *Virtually* unlimited scaling by adding nodes to the cluster	- Can be configured to have no SPOF through redundancy of key cluster components - There exist several challenges to achieve continuous availability of service	- Multi-node system - Each node runs its own copy of the operating system and application - Allows flexibility in configuration

Table 4: Characteristics of SMP and cluster systems

2.3 Cluster Software Components

We classify the software components that comprise the environment of a commodity cluster in four major categories: the operating system that runs on each of the cluster nodes, application execution environment such as libraries and debuggers, cluster installation infrastructure, and cluster services components such as cluster membership, storage, management, traffic distribution and application services.

The critical software components include cluster membership, storage, fault management, and traffic distributions services. The cluster membership service includes functions to recognize and manage the nodes membership in the cluster. The cluster storage service includes the replication and retrieval of cluster configuration and application data. The fault management service includes functions to recognize

hardware and software faults and recovery mechanisms. The traffic distribution service includes functions to distribute the incoming traffic across the nodes in the cluster.

2.4 Cluster Hardware Components

The key components, which comprise a commodity cluster, are the nodes performing the computing and the dedicated interconnection network providing the data communication among the nodes. A cluster node is different from an MPP node in that a cluster node is an operational standalone computing system. A cluster node integrates several key subsystems that include the processor, memory, storage, external interfaces, and network interfaces.

2.5 Benefits of Clustering Technologies

Computer clusters provide several advantages including high availability, scalability, high performance compared to single server architectures, rapid response to technology improvements, manageability of multiple servers as a single server, transparency, and flexibility of configuration. The following sub-sections present these advantages and benefits of clusters.

2.5.1 High Availability

High availability (HA) refers to the availability of resources in a computer system. We achieve HA through redundant hardware, specialized software, or both. With clusters, we can provide service continuity by isolating or reducing the impact of a failure in the node, resources, or device through redundancy and fail over techniques.

It is important that a service not only be down except for N minutes a year, but also that the length of outages be short enough, and the frequency of outages be low enough, that the end user does not perceive it as a problem. Therefore, the goal is to have a small number of failures and a prompt recovery time. This concept is termed *Service Availability,* meaning whatever services the user wants are available in a way that meets the user's expectations.

2.5.2 Scalability

Clusters provide means to reach high levels of scalability by expanding the capacity of a cluster in terms of processors, memory, storage, or other resources, to support users and traffic growth [1].

2.5.3 Performance

We can achieve a better performance characterized by improved processing speed by using clusters and the cluster-wide resources instead of the resources of a standalone server.

2.5.4 Rapid Response to Technology Improvements

Commodity clusters are most able to track technology improvements and respond rapidly to new offerings. Clusters benefit from the latest mass-market technology, as it becomes available at low cost. As new devices including processors, memory, disks, and network interfaces become available in the market, we can integrate them into cluster nodes allowing clusters to be the first class of parallel systems to benefit from such advances. Similarly, clusters benefit from the latest operating system and networking features, as they become available.

2.5.5 Manageability

Clusters require a management layer that allows us to manage all cluster nodes as a single entity [63]. Such cluster management facilities help reduce system management costs. There exists a significant number of cluster management software, almost all of them originating from research projects, and are now adopted by commercial vendors.

2.5.6 Cost Efficient Solutions

Clusters take advantage of COTS hardware, which allows a better price to performance ratio when compared to a dedicated parallel supercomputer. In addition, the availability of open source operating systems, system software, development tools, and applications has contributed to the provisioning of cost effective clusters built with open source software.

2.5.7 Expandability and Upgradeability

We can expand clusters by adding more nodes, disk storage, and memory, as necessary. As hardware, software, operating system and network upgrades become available, we can upgrade a cluster node independently of the others; this presents a major advantage over SMP and MMP systems.

2.5.8 Transparency

The SSI layer represents the nodes that make up the cluster as a single server. It allows users to use a cluster easily and effectively without the knowledge of the underlying system architecture or the number

of nodes inside the cluster. This transparency frees the end-user from having to know where an application runs.

2.5.9 Flexibility of Configuration

Clusters allow flexibility in configurations that is not available through conventional SMP and MPP systems. The number of cluster nodes, memory capacity per node, number of processors per node, and interconnect topology, are all parameters of the cluster structure that may be specified in fine detail on a per system basis without incurring additional cost. Furthermore, we can modify the cluster structure or augment it over time as need and opportunity dictates. This expanded control over the cluster structure not only benefits the end user but the cluster vendor as well, yielding a wide array of system capabilities and cost tradeoffs to meet customer demands.

2.6 The OSI Layer Clustering Techniques

The following sub-sections present a survey of the clustering techniques that operate at OSI layer two (data link layer), OSI layer three (network layer), and OSI layer seven (application layer). The material and figures presented in subsections 2.6.1, 2.6.2, and 2.6.3 are paraphrased from [69].

2.6.1 L4/2 Clustering

Figure 7 from [69] illustrates the L4/2 clustering model. The level 4 web switch works at the TCP/IP level. The dispatcher, also called web switch, is the entry point to the cluster; it receives incoming traffic, and maintains a binding table to associate each session with it the assigned server in the cluster. The dispatcher forwards incoming requests to the cluster servers based on the traffic distribution algorithm such as RR or weighted RR. The servers in the cluster process the incoming traffic and reply directly to the users. Each server in the cluster processes the packets belonging to the same connection. When the dispatcher receives an incoming request, it creates an entry in a connection map that includes information such as the origin of the connection and the cluster server servicing it. The layer two destination address is then rewritten to the hardware address of the chosen cluster server, and the frame is placed back on the network.

The dispatcher and the cluster servers share the same cluster network layer address using the primary and secondary IP addresses. The primary address of the dispatcher is the same as the cluster address. Each cluster server is configured with the cluster address as a secondary address, either by using the interface *aliasing* or by changing the address of the loopback device. The gateway is configured such that all

packets arriving for the cluster address are addressed to the dispatcher at layer two using a static Address Resolution Protocol (ARP) cache entry.

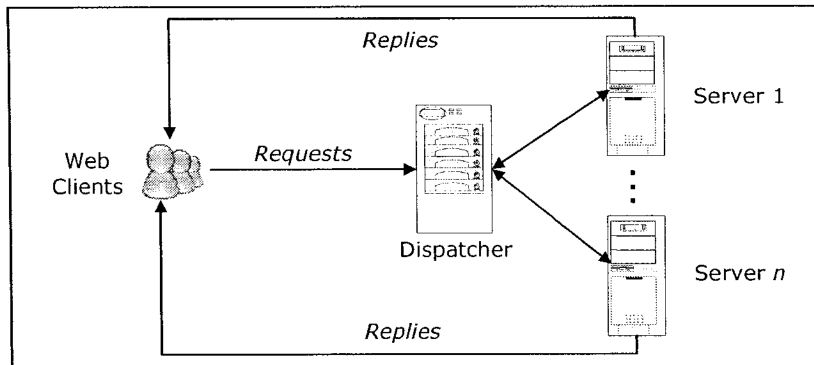


Figure 7: The L4/2 clustering model [69]

The dispatcher can receive three types of connections: a connection initiation, a connection that belongs to an existing stream of connection, or a connection that is neither. If the dispatcher receives a packet that corresponds to a TCP/IP connection initiation, the dispatcher selects one of the servers in the cluster to service the request. If the dispatcher receives a packet that is not for a connection initiation, the dispatcher examines its connection map to determine if the packet belongs to a currently established connection. If the packet belongs to an existing connection, then the dispatcher rewrites the layer two destination address to be the address of the cluster server previously selected to service this request, and forwards the packet to the cluster server. If the packet does not correspond to an established connection and if it is not a connection initiation packet, then the dispatcher drops it.

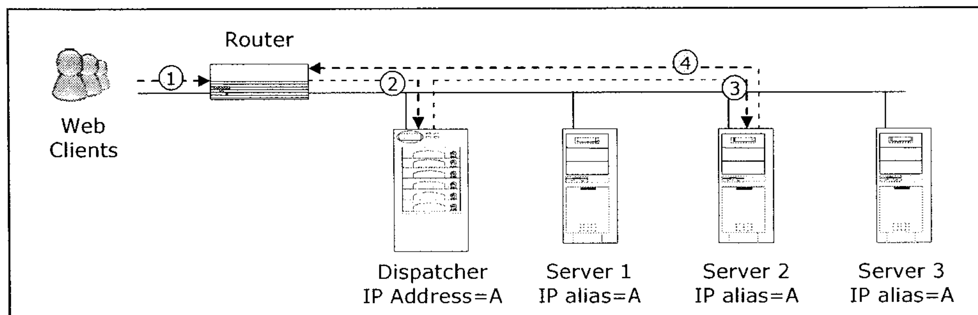


Figure 8: Traffic flow in an L4/2 based cluster [69]

Figure 8, from [69], illustrates the traffic flow in an L4/2 clustered environment [69]. A web client sends an HTTP packet (1) with A as the destination IP address. The immediate router sends the packet to the

dispatcher at IP address A (2). Based on the traffic distribution algorithm and the session table, the dispatcher decides which back-end server will handle this packet, Server 2 for instance, and sends the packet to Server 2 by changing the MAC address of the packet to Server 2's MAC address and forwarding it (3). Server 2 accepts the packet and replies directly to the web client.

Several research projects and commercial products implement layer two clustering such as the ONE-IP [18] developed at Bell Laboratories, the IBM's eNetwork Dispatcher, and the LSMAC implementation, the later is discussed in Section 2.15.4. Furthermore, the Linux Virtual Server (LVS) is an open source project that aims to provide a high performance and highly available software clustering implementation for Linux [74]. It implements layer 4 switching in the Linux kernel, and provides a virtual server layer built on a cluster of real servers and allowing TCP and UDP sessions to be load balanced between multiple cluster servers. LVS support L4/2 clustering through its Direct Routing (DR) implementation, which we discuss in Section 2.15.7.2.

2.6.2 L4/3 Clustering

Figure 9, from [69], illustrates the L4/3 clustering approach with the dispatcher appearing as a gateway for the servers in the cluster. The incoming traffic arrives to the dispatcher. The dispatcher can receive three types of connections: a connection initiation, connection that belongs to an existing stream of connection, or a connection that is neither.

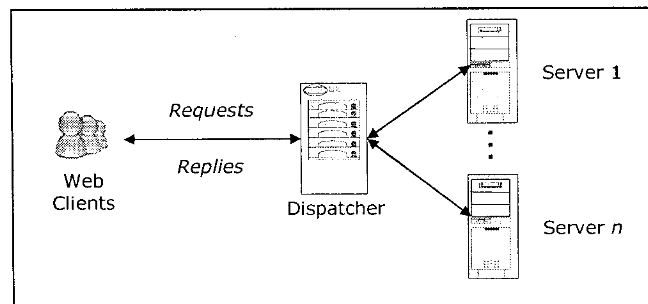


Figure 9: The L4/3 clustering model [69]

If the packet received corresponds to a TCP/IP connection initiation, the dispatcher selects one of the servers in the server pool to service the request. The selection of the cluster server relies on a traffic distribution algorithm. The dispatcher then creates an entry in the connection map noting the origin of the connection, the destination server, and other relevant information. However, unlike the L4/2 approach, in L4/3, the dispatcher rewrites the destination IP address of the packet as the address of the cluster server

selected to service this request. Furthermore, the dispatcher re-calculates any integrity codes affected such as packet checksums, cyclic redundancy checks, or error correction checks. Next, the dispatcher sends the modified packet to the cluster server corresponding to the new destination address of the packet. The cluster server then processes the traffic and sends it to the web clients through the dispatcher. The dispatcher rewrites the source address to the cluster address, re-computes the integrity codes, and forwards the packet to the web client.

If the incoming web client traffic is not a connection initiation, the dispatcher examines its connection map to determine if it belongs to a currently established connection. If it does, the dispatcher rewrites the destination address as the destination server previously selected, re-computes the checksums, and forwards the packet to the cluster server as described earlier.

If the packet does not correspond to an established connection and it is not a connection initiation packet, then the dispatcher drops the packet.

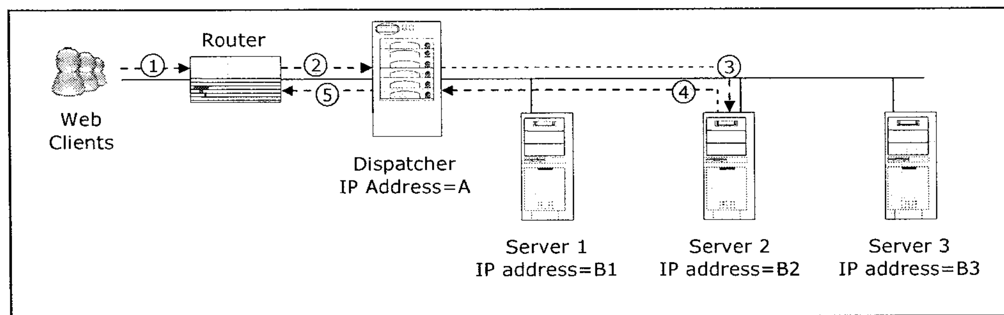


Figure 10: The traffic flow in an L4/3 based cluster [69]

Figure 10, from [69], illustrates the traffic flow in an L4/3 clustered environment [69]. A web client sends an HTTP packet with A as the destination IP address (1). The immediate router sends the packet to the dispatcher (2) being the owner of the IP address A. The dispatcher forwards this packet to the back-end server, Server 2 (3) based on the traffic distribution algorithm and the session table. The dispatcher then rewrites the destination IP address as B2, recalculates the IP and TCP checksums, and sends the packet to B2 (3). Server 2 accepts the packet and replies to the client via the dispatcher (4), which the back-end server sees as a gateway. The dispatcher rewrites the source IP address of the replying packet as A, recalculates the IP and TCP checksums, and sends the packet to the web client (5).

RFC 2391, Load Sharing using IP Network Address Translation, presents the L4/3 clustering approach [71]. The LSNAT project at the University of Nebraska-Lincoln provides a non-kernel space implementation of the L4/3 clustering approach [20], which we discuss in Section 2.15.4. Furthermore,

LVS supports the L4/3 clustering approach through its Network Address Translation method, which we discuss in Section 2.15.7.1.

2.6.3 L7 Clustering

Level 7 web switch works at the application level. The web switch establishes a connection with the web client and inspects the HTTP request content to decide about dispatching. The L7 clustering technique is also known as content-based dispatching since it operates based on the contents of the client request. The Locality-Aware Request Distribution (LARD) dispatcher developed by the researchers at Rice University is an example of the L7 clustering. LARD partitions the folder containing the web document tree disjoint sub-folders. The dispatcher then allocates each server in the cluster to one of these sub-folders to serve. As such, LARD provides content-based dispatching as the dispatcher receives web clients requests.

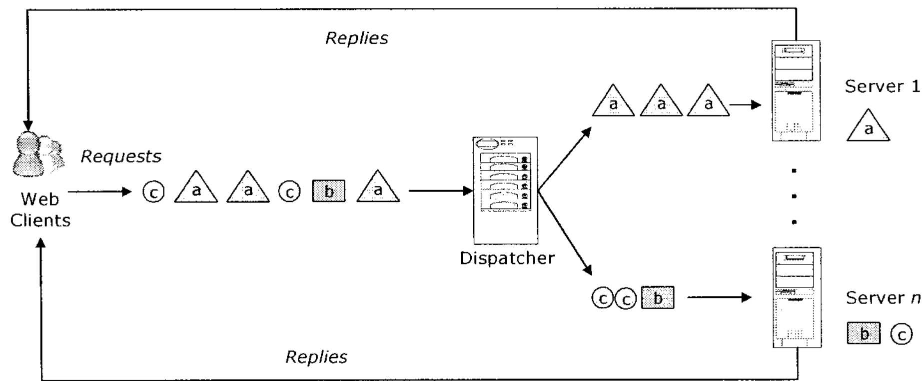


Figure 11: The process of content-based dispatching – L7 clustering model [69]

Figure 11, from [69], presents an overview of the processing with the L7 clustering approach. Server 1 processes request of type a; Server 2 processes requests of types b and c. The dispatcher separates the stream of requests into two streams of requests: one stream for Server 1 with requests of type a, and stream for Server 2 with requests of types b and c. As requests arrive from clients for the web cluster, the dispatcher accepts the connection and the request. It then classifies the requested document and dispatches the request to the appropriate server. The dispatching of requests requires support from a modified kernel that enables the connection handoff protocol. After establishing the connection, identifying the request, and choosing the cluster server, the dispatcher informs the cluster server of the status of the network connection, and the cluster server takes over that connection, and communicates directly with the web client. Following this approach, the LARD allows the file system cache of each cluster server to cache a separate part of the web tree rather than having to cache the entire tree, as it is the

case with L4/2 and L4/3 clustering. Additionally, it is possible to have specialized server nodes, where the dynamically generated content is offloaded to special compute servers while other requests are dispatched to servers with less processing power. The LARD requires modifications to the operating system on the servers to support the TCP handoff protocol.

2.6.4 Discussion of the OSI Layer Clustering Techniques

The transparent server clustering approaches can be broadly classified into three categories: L4/2, L4/3, and L7. Table 5, from [69], summarizes these OSI layer clustering technologies, and highlights their advantages and disadvantages. Each of the approaches has specific drawbacks such as creating bottlenecks that limit the scalability, or presenting single points of failure.

The L4/2 clustering approach has a performance advantage over L4/3 clustering. In L4/2 clustering, the network address of the cluster server to which the packet is delivered is identical to the one the web client used originally in the request packet. As a result, the cluster server handling that connection responds directly to the client rather than through the dispatcher. Therefore, the dispatcher processes only the incoming data stream. Furthermore, the dispatcher does not re-compute integrity codes (such as the IP checksums) in software since only layer two parameters are modified. Therefore, the two parameters that limit the scalability of the cluster are the network bandwidth and the sustainable request rate of the dispatcher, which is the only portion of the transaction actually processed by the dispatcher.

The dispatcher in the L4/2 clustering approach must have a direct physical connection to all network segments that house servers (due to layer two frame addressing). For L4/2 dispatchers, system performance is constrained by the ability of the dispatcher to set up, look up, and tear down entries. However, the most critical performance metric is the sustainable request rate.

This contrasts with L4/3 clustering, where the server may be anywhere on any network with the sole constraint that all client-to-server and server-to-client traffic must pass through the dispatcher. In practice, this restriction on L4/2 clustering has little impact since servers in a cluster are most likely to reside on the same network. The limitation of L4/3 dispatchers is their ability to rewrite and recalculate the checksums for the massive numbers of packets they process. Therefore, the most critical performance metric is the throughput of the dispatcher.

L4/2 clustering theoretically outperforms L4/3 clustering due to the overhead imposed by L4/3 clustering with the necessary integrity code recalculation and the fact that all traffic must go through the dispatcher. As a result, the L4/3 dispatcher processes more traffic than an L4/2 dispatcher does. Therefore, the total

data throughput of the dispatcher limits the scalability of the system more than the sustainable request rate.

As for the L7 clustering approach, it has limitations related to the complexity of the content-based routing algorithm, size of their cache, and the limitation to only having nodes with disks in the cluster.

	L4/2	L4/3	L7
Mechanism	Link-layer address translation	Network address translation	Content-based routing
Flows	Incoming only	Incoming/outgoing	Incoming/ outgoing varies
HA and fault tolerance	Varies, several single point of failures	several single point of failures	Varies, several single point of failures
Restrictions	- Incoming traffic passes through dispatcher	- Dispatcher lies between client and server. - All incoming and outgoing traffic passes through dispatcher	- Incoming traffic passes through dispatcher - Cluster nodes must have disks
Bottleneck	- Connection dispatching	- Connection dispatching - Integrity code calculations	- Connections dispatching - Dispatcher complexity
Client information	At TCP/IP level	At TCP/IP level	TCP/IP information and HTTP header content

Table 5: Comparison of the clustering techniques operating at the OSI layer [69]

2.7 The RR-DNS Approach

The round robin DNS maps a single server name to the different IP addresses in a round robin manner. Figure 12 illustrates this approach. The web user interacts with web client software (1) to request a certain document from a web site. The web client software sends the name translation request to a domain name server, to convert the web site name to an IP address (2). The request arrives to the RR-DNS, the domain name server of the web cluster (3). The RR-DNS selects a web server from the cluster in a round robin method (4). The IP address of the selected web cluster node and a TTL (Time To Life) value of the connection are sent to the web client software via the intermediate name servers (5). The address of the selected web cluster node (Node 2) arrives to the web client software (6). The web client software sends the request to Node 2 (7). Node 2 processes the request and replies back (7).

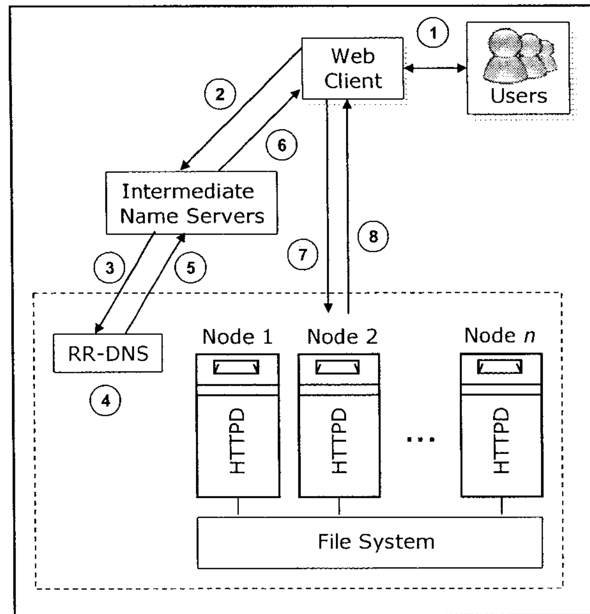


Figure 12: RR-DNS approach

Following this approach, the round robin DNS distributes the traffic among the servers, and maps clients to different servers in the cluster. The round robin DNS method is not very efficient because of the number of intermediate name servers that cache the name-to-IP mapping. Furthermore, because of the caching nature of clients and the hierarchical DNS system, it can lead to dynamic load imbalance among the servers, making it unrealistic for a server to handle its peak load. It is difficult to choose the Time-To-Live (TTL) value of a name mapping: with small values, round robin DNS is a bottleneck, and with high values, the dynamic load imbalance gets worse. Even when the TTL value is set to zero, the scheduling granularity is per host; different users' access pattern may lead to dynamic load imbalance, because some people pull many pages from the site, and others just surf a number of pages and go away. In addition, if a web cluster node fails, some clients continue trying to access the failed node using the cached IP address, which leads to long delays.

2.8 Discussion of Software Approaches to Clustering

Software clustering approaches have three main advantages that make them a better alternative than clustering solutions: flexibility, intelligence, and availability. First, software-*clustering* approaches can augment existing hardware devices, thereby providing a more robust traffic distribution and failover

solution. Secondly, they provide a level of intelligence that enables preventive traffic distribution measures that actually minimize the chance of servers becoming unavailable. In the event that a server becomes overloaded or actually fails, some software approaches can automatically detect the problem and reroute the HTTP requests to other available nodes in the cluster. Thirdly, with software clustering approaches, we can support high availability capabilities to avoid single points of failure. An individual server failure does not affect the service availability since functionalities and failover capabilities are distributed among the cluster servers.

However, we need to consider several issues when evaluating cluster software solutions, mainly the differences among feature sets, the platform constraints, their HA and scaling capabilities. Software clustering solutions have different capabilities and features, such as their capabilities of providing automatic failure detection, notification, and recovery. Some solutions have significantly delayed failure detection; others allow the configuration of the load thresholds to enable preventive measures. In addition, they can support different redundancy models such as the 1+1 active/standby, 1+1 active/active, N+M and N-way. Therefore, we need to determine the needs or requirements for scalability and failover and pick the solution accordingly. In addition, software solutions have limited platform compatibility; they are available to run on specific operating system or computing environments. Furthermore, the capability of the clustering solution to scale is important. Some solutions have limited capabilities restricted to four, eight, or 16 nodes, and therefore have scaling limitations.

2.9 Discussion of Hardware Approaches to Clustering

Figure 13 illustrates the hardware clustering approach using a network device called the packet router. The packet router sits in front of a number of web servers and directs incoming HTTP requests to available web servers in the cluster. The web server software running on cluster nodes have access to the same storage repository. The packet router distributes traffic to the cluster servers based on a predefined distribution policy. The router device with the web servers comprises a virtual server. Load balancing switches, such as the Cisco LocalDirector, redirect TCP requests to servers belonging to a cluster. The LocalDirector provides traffic distribution service by presenting a common virtual IP address to the web clients and then forwarding their incoming requests to an available node within the web cluster. If a cluster node becomes unavailable, the LocalDirector detects the failures and stops forwarding traffic to the node.

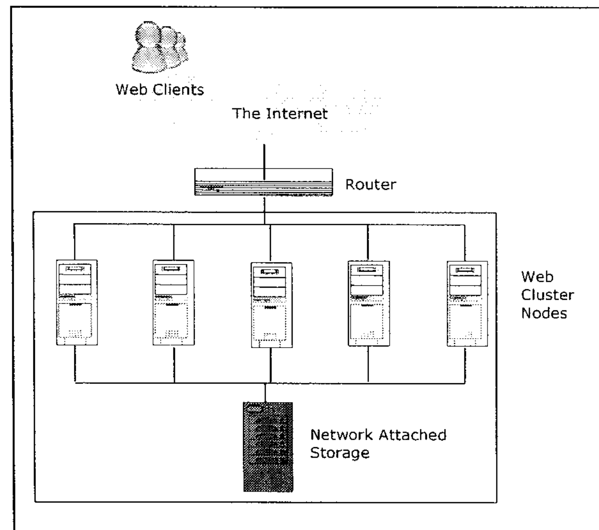


Figure 13: Using a router to hide the web cluster

Hardware-based clustering solutions use routers to provide a single IP interface to the cluster and to distribute traffic among various cluster nodes. These solutions are a proven technology; they are neither complicated nor complex by design. However, they have certain limitations such as limited intelligence, un-awareness of the applications running on the cluster nodes, and the presence of SPOF.

Limited intelligence: Packet routers can load balance in a round robin fashion, and some can detect failures and automatically remove failed servers from a cluster and redirect traffic to other nodes. These routers are not fully intelligent network devices. They do not provide application-aware traffic distribution. While they can redirect requests upon discovering a failure, they do not allow configuring redirection thresholds for individual servers in a cluster, and therefore, they are unable to manage load to prevent failures.

Lack of Dynamism: A router cannot measure the performance of a web application server or make an intelligent decision on where to route the request based on the load of the cluster node and its hardware characteristics.

Single point of failure: Packet router constitutes a SPOF for the entire cluster. If the router fails, the cluster is not accessible to end users and the service becomes unavailable.

2.10 Scalability in Internet and Web Servers

Scalability is the ability of an application server, such as a web server, to grow to support a large number of concurrent users without performance degradation. Generally, scalability refers to how well a hardware or software system can adapt to increased demands. Perfect scalability is linear: if we double the number of processors in the system, we expect the system to serve double the number of requests per second it normally serves. We consider this optimum performance. Unfortunately, this is not the case in real systems.

As the number of online services continues to grow as well as the number of Internet users, Internet and web servers have to meet new requirements in areas of availability, scalability, performance, reliability, and security. They have to cope with the explosive growth of the Internet [1][11], continuous increasing traffic, and meet all expectations in terms of stringent requirements in those areas. One additional capability includes supporting geographical mirroring for increased high availability, especially when providing critical services such as electronic banking and stock brokering. However, these servers experience scalability problems – they are slow; they are continuously hacked and attacked by malicious users, and at times they are not available for service, making businesses liable to losing money when users are not able to access the services [68]. As such, scalability presents itself as a crucial factor for the success or failure of online services and it is certainly one important challenge faced when designing servers that provide interactive services for a wide clientele.

Many factors can affect negatively the scalability of systems [39]. The first common factor is the growth of user base, which cause capacity problems for servers that can only serve a certain number of transactions per second. A second key factor negatively affecting the scalability of servers is the number and size of data objects and documents, particularly the size of audio and video files strains the network and I/O capacity causing scalability problems. Finally, the non-uniform request distribution imposes strains on the cluster servers and network at certain times of the day. These factors can cause servers to suffer from bottlenecks, and run out of network, processing, and I/O resources.

2.11 Scalability in Telecommunication Servers

Telecommunication servers provide a wide range of IP applications for mobile phones. These servers require scaling capabilities to support an increasing number of subscribers and services. They support some of the rigorous requirements in terms of reliability, performance, availability, and scalability. They aim to provide *five nines* availability, which translates into a maximum of five minutes and fifteen

seconds of downtime per year that includes downtime associated with operating system, software upgrades, and hardware maintenance. These expectations place an unprecedented burden on telecommunication equipment manufacturers to ensure that all the elements needed to support a service are functioning whenever a user requests that particular service.

While achieving a 100 percent of uptime is desirable, it is difficult to achieve. A fault-tolerant system needs to function correctly, given the small but practically inevitable presence of a fault in the system. Supporting five nines service availability depends on the near-flawless interaction of applications, operating system, management middleware, hardware, as well as on environmental and operational factors.

The requirements of telecommunication servers are getting more complex as the telecom industry is moving towards an *always connected, always online* paradigm with a new suite of third generation interactive and multimedia services. These servers suffer from scalability problems as the number of mobile subscribers is increasing at a fast pace [1]. To cope with the increased number of users and traffic, mobile operators are resorting to upgrading servers or buying new servers with more processing power, a process that proved to be expensive and iterative.

When the servers are not able to cope with increased traffic, it results in failure to meet the high expectations of paying customers who expect services to be available at all times with acceptable performance levels, and meeting and managing service level agreements. Service level agreements dictate the percentage of the time services will be available, the number of users that can be served simultaneously, specific performance benchmarks to which actual performance will be periodically compared, and access availability. If ISPs, for instance, are not able to cope with the increasing number of users, they will break their service level agreements causing them to loose money and potentially loose customers. Similarly, mobile operators have to deal with huge money losses if their servers are not available for their subscribers.

2.12 How Users Experience Scalability

Variations in the scalability of a system are transparent to the end users. Users experience the capacity of a system in various ways such as how fast and accurately the system responds to their requests. The user expects to receive the response in an acceptable time with no errors [57][53][58]. Therefore, the server needs to adapt to different numbers of users and amounts of data, without resource problems or performance bottleneck. We can measure these qualities via the system response time.

The response time is the space of time that exists between the moment a user gives an input, or posts a request, to the moment when a user receives an answer from the server. Total response time includes the time to connect, the time to process the request on the server, and the time to transmit the response back to the client:

```
Total response time = connect time + process time + response transit time
```

When throughput is low, the response transmit time is insignificant. However, as throughput approaches the limit of network bandwidth, the server has to wait for bandwidth to become available before it can transmit the response.

The response time in a distributed system consists of all the delays created at the source site, in the network, and at the receiver site. The possible reasons for the delays and their length depend on the system components and the characteristics of the transport media. The response time consists of the delays in both directions.

2.13 Principles of Scalable Architecture

This section discusses the principles of scalable architectures. The material presented in this section is paraphrased from [67] with adaptations to reflect our focus on cluster architectures.

Server applications are characterized by their consumption of four primary system resources: processor, memory, file system bandwidth, and network bandwidth. We can achieve scalability by simultaneously optimizing the consumption of these resources and designing an architecture that can grow modularly by adding more resources.

Several design principles are required to design a scalable system. The core principles include divide and conquer, asynchrony, encapsulation, concurrency, and parsimony [67]. Each of these principles presents a concept that is important in its own right when designing a scalable system. There are also tensions between these principles; we can sometimes apply one principle at the cost of another. The root of a solid system design is to strike the right balance among these principles.

The following subsections present on each of these principles.

2.13.1.1 Divide and Conquer

The divide and conquer principle is based on the idea that the system needs to be divided into smaller sub-systems, where each sub-system carries out a very specific function [67]. The advantages of the divide and conquer approach is that it allows the distribution of the system load.

2.13.1.2 Asynchrony

The asynchrony principle indicates that the system carries out work based on its available resources [67]. "Asynchrony decouples functions and lets the system schedule resources more freely and thus potentially more completely" [67]. In our specific focus with web server, the principle of asynchrony will allows us to implement strategies that effectively deal with stress conditions such as peak load.

2.13.1.3 Encapsulation

The encapsulation principle is the concept of building the system using loosely coupled components, with little or no dependence among components [67] . "This principle correlates with asynchrony". "Highly asynchronous systems tend to have well encapsulated components and vice versa"[67]. "Loose coupling means that components can pursue work without waiting for work from others" [67].

2.13.1.4 Concurrency

The concurrency principle indicates that there are many dynamic moving parts in a system and the goal is to split the activities across hardware, processes, and threads [67]. "Concurrency aids scalability by ensuring that the maximum possible work is active at all times and addresses system load by spawning new resources on demand within predefined limits" [67]. "The Concurrency principle also maps directly to the ability to scale by rolling in new hardware" [67]. The more concurrency the system supports, the better the possibilities to expand by adding new hardware.

2.13.1.5 Parsimony

The parsimony principle relies on the ability of the system designer and the developer of the system to be economical in their design by paying attention to all the details of design and implementation resulting in a higher throughput system [67].

2.14 Overview of Related Work

Server scalability is a recognized research area both in the academia and in the industry and it is an essential factor in the client/server dominated network environments. Researchers around the world are investigating clusters and commodity hardware as an alternative to expensive specialized hardware for building scalable Internet and web servers. Although the area of cluster computing is relatively new, there is an abundance of research projects in the areas of cluster computing and scalable cluster-based servers.

This section serves as a brief overview of some of the other surveyed work that helped us get a better understanding of the areas of research related to scalable web servers.

In [4], the authors present and evaluate an implementation of a prototype scalable web server. The prototype consists of a load-balanced cluster of hosts that collectively accept and service TCP connections. The host IP addresses are advertised using the Round Robin DNS technique, allowing any host to receive requests from any client. Once a client attempts to establish a TCP connection with one of the hosts, a decision is made as to whether or not the connection should be redirected to a different host – namely, the host with the lowest number of established connections. They use the low-overhead Distributed Packet Rewriting (DPR) [7] technique to redirect TCP connections. In the prototype, each host keeps information about the remaining hosts in the system. Load information is maintained using periodic multicast amongst the cluster hosts. Performance measurements suggest that the prototype outperforms both pure RR-DNS and the stateless DRP solutions.

In [67], the authors address strategies for designing a scalable architecture: divide and conquer asynchrony, encapsulation, concurrency, and parsimony. The authors argue that the strategies presented are not comprehensive; however, they represent critical strategies for server-side scaling.

In [19], the authors describe a prototype of a scalable and highly available web server, built on an IBM SP-2 system, and analyze its scalability. The system architecture consists of a set of logical front-end or network nodes and a set of back-end or data nodes connected by a switch, and a load-balancing component. A combination of TCP routing and Domain Name Server (DNS) techniques are used to balance the load across the front-end nodes that run the Web (httpd) daemons. The scalability achieved is quantified and compared with that of the known DNS technique. The load on the back-end nodes is balanced by striping the data objects across the back-end nodes and disks. High availability is provided by detecting node or daemon failures and reconfiguring the system appropriately. The scalable and highly available web server is combined with parallel databases, and other back-end servers, to provide integrated scalable and highly available solutions.

In [14], the authors propose a new scheduling policy, called multi-class round robin (MC-RR), for web switched operating at the layer 7 of the open system interconnection (OSI) protocol stack to route requests reaching the web cluster. The authors demonstrate through a set of simulation experiments that MC-RR is more effective than round robin for web sites providing highly dynamic services.

In [6], the authors describe their architecture for a web server designed to cope with the ongoing increase of the Internet requirements. The proposed architecture addresses the need for a powerful data

management system to cope with the increase in the complexity of users' requests, and a caching mechanism to reduce the amount of redundant traffic. The author extends the architecture with a caching system that builds up an adaptive hierarchy of caches for web servers, which allow them to keep up the changes in the traffic generated by the applications they are running.

In [11], the authors present a comparison of different traffic distribution methods for HTTP traffic in scalable web clusters. The authors present a classification framework for the different load balancing methods and compare their performance. In particular, they discuss the rotating name server method in comparison with alternative load balancing method based on remapping requests and responses in the network. Their results demonstrate that the remapping requests and responses yield better results that the rotating name serve method.

The researchers at the Korea Advanced Institute of Science and Technology have developed an adaptive load balancing method that changes the number of scheduling entities according to different workload [38]. It behaves similarly like a dispatcher based scheme with low or intermediate workload, taking advantage of fine-grained load balancing. When the dispatcher is overloaded, the DNS servers distribute the dispatching jobs to other entities such as back-end servers. In this way, they relax the hot spot of the dispatcher. Based on simulation results, they demonstrated that the adaptive dispatching method improves the overall performance on realistic workload simulation.

In [4], the authors present and evaluate an implementation of a prototype scalable web server consisting of a balanced cluster of hosts that collectively accept and service TCP connections. The host IP addresses are advertised using round robin DNS technique allowing a host to receive requests from a client. They use a low-overhead technique called the distributed packet rewriting (DPR) to redirect TCP connections. Each host keeps information about the remaining hosts in the system. Their performance measurements suggest that their prototype outperforms round robin DNS. However, their benchmarking was limited to a five-node cluster, where each node reached a peak of 632 requests per second.

In [69], the authors examine the seminal work, early products, and a sample of contemporary commercial offerings in the field of transparent Web server clustering. They broadly classify transparent server clustering into three categories, L4/2, L4/3, and L7, discuss their approaches, advantages and disadvantages.

In [64], the authors present their two implementations for traffic manipulation inside a web cluster: MAC-based dispatching (LSMAC) and IP-based dispatching (LSNAT). The authors discuss their results, and the advantages and disadvantages of both methods. Section 2.15.4 discusses those approaches.

In [55], the researchers from Lucent Technologies and the University of Texas at Austin present their architecture for a scalable web cluster. The distributed architecture consists of independent servers sharing the load through a round robin traffic distribution mechanism. The "redirection-based" hierarchical server architecture eliminates bottlenecks in the server complex and allows hardware to be added seamlessly to handle increases in load. In this approach, two levels of servers are used: redirection servers and normal HTTP servers. Data are partitioned according to their content and stored on different HTTP servers. Redirection servers are used to distribute the users' requests to the corresponding HTTP servers using the redirection mechanism supported by the HTTP protocol. This approach is completely transparent to the user, allows for distribution of load among servers, and achieves better caching efficiency compared with other load balancing schemes.

In [37], the authors present on optimizations to the National Center for Supercomputing Applications (NCSA) http server to make it more scalable and allow it to serve more requests. The paper outlines the methodology used at the NCSA in building a scalable web server. The implementation described allows for dynamic scalability by rotating through a pool of http servers that are alternately mapped to the hostname alias of the web server.

Section 2.15 The following section examines six projects that are close to our work in terms of scope and goal, discuss their approaches, advantages and drawbacks, discuss their prototypes and implementations, and present the learned lessons from their experiences.

2.15 Related Work: In-depth Examination

This section discusses seven projects that share the common goal of increasing the performance and scalability of web clusters, presents their respective area of research, their architectures, highlights their status and plans, and discusses the contributions of their research into our work. The works discussed are the following:

- "*Redirectional-based Web Server Architecture*" at University of Texas (Austin): The goal of this project is to design and prototype a redirectional-based hierarchical architecture that eliminates bottlenecks in the cluster and allows the administrator to add hardware seamlessly to handle increased traffic [38]. Section 2.15.1 discusses this project.
- "*Scalable policies for Scalable Web clusters*" at the University of Roma: The goal of the project is to provide scalable scheduling policies for web clusters [14][3]. Section 2.15.2 discusses this project.

- "*The Scalable Web Server (SWEB)*" at the University of California (Santa Barbara) : The project investigates investigate the issues involved in developing a scalable World Wide Web (WWW) server on a cluster of workstations and parallel machines, using the Hypertext Transport Protocol [2]. The main objective is to strengthen the processing capabilities of such a server by utilizing the power of multicomputers to match huge demands in simultaneous access requests from the Internet. The authors have implemented a system called SWEB on a distributed memory machine, the Meiko CS-2, and networked SUN and DEC workstations. The scheduling component of the system actively monitors the usages of CPU, I/O channels and the interconnection network to distribute effectively HTTP requests across processing units to exploit task and I/O parallelism. Their results indicate that the system delivers good performance on multi-computers and obtains significant improvements over other approaches [2]. Section 2.15.3 discusses this project.
- "*LSMAC and LSNAT*": The project at the University of Nebraska-Lincoln investigates server responsiveness and scalability in clustered systems and client/server network environments [20]. The project is focusing on different server infrastructures to provide a single entry into the cluster and traffic distribution among the cluster nodes [64]. Section 2.15.4 examines this project.
- "*Harvard Array of Clustered Computers (HACC)*": The HACC project aims to design and prototype cluster architecture for scalable web servers [83]. The focus of the project is on a technology called "IP Sprayer", a router component that sits between the Internet and the cluster and is responsible for traffic distribution among the nodes of the cluster. Section 2.15.5 discusses this project.
- "*IBM Scalable and Highly Available Web Server*": This project is investigating scalable and highly available web clusters [72]. The goal with the project is to develop a scalable web cluster that will host web services on IBM proprietary SP-2 and RS/6000 systems. Section 2.15.6 discusses this project.
- "*The Linux Virtual Server*" [74]: The goal of the Linux Virtual (LVS) project is to offer load balancing for web servers using Layer 4 switching. Section 2.15.7 discusses this project and presents the results of testing an LVS cluster for performance and scalability.

2.15.1 Hierarchical Redirection-Based Web Server Architecture

The University of Texas at Austin and Bell Labs are collaborating on a research project to design scalable web cluster architecture. The authors confirm that a distributed architecture consisting of independent servers sharing the load is most appropriate for implementing a scalable web server [37]. Their study of

currently available approaches, such as RR DNS, demonstrates that these approaches do not scale effectively because they lead to bottleneck in different parts of the system. This section presents their redirection-based hierarchical server architecture, discusses its advantages and drawbacks, and presents their contributions related to our work.

The study proposes a redirectional-based hierarchical server architecture that includes two levels of servers: redirectional servers and normal HTTP servers. The administrator of the cluster partitions data and stores it on different cluster nodes. The directional servers distribute the requests of the web users to the corresponding HTTP server.

Figure 14 illustrates the architecture of the hierarchical redirection based web server approach. Each HTTP server stores a portion of the data available at the site. The round robin DNS distributes the load among the redirection servers [37]. The redirection servers in turn redirect the requests to the HTTP servers where a subset of the data resides. The redirection mechanism is part of the HTTP protocol and it is completely transparent to the user. The browser automatically recognizes the redirection message, derives the new URL from it, and connects to the new server to fetch the file.

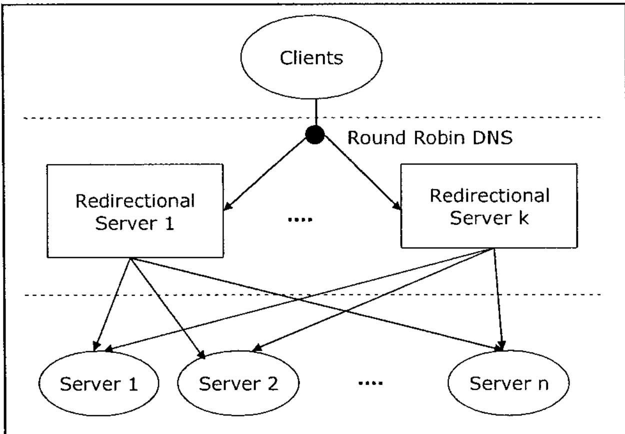


Figure 14: Hierarchical redirection-based web server architecture [37]

The original goal of the redirection mechanism supported in HTTP was to facilitate moving files from one server to another. When a client uses the old URL from its cache or from the bookmark after, and if the file referenced by the old URL was moved to a new server, the old server returns a redirection message, which contains the new URL. The cluster administrator partitions the documents stored at the site among the different servers based on their content. For instance, server 1 could store stock price data, while

server 2 stores weather information and server 3 stores movie clips and reviews. All requests for stock quotes are directed to server 1 while requests for weather information are directed for server 2.

It is possible to implement the architecture described with server software modifications. However, in order to provide more flexibility in load balancing and additional reliability, there is a need to replicate contents on multiple servers. Implementing data replication requires modifying the data structure containing the mapping information. If there is replication of data, a logical file name is mapped to multiple URLs on different servers. In this case, the redirection server has to choose one of the servers containing the relevant information data. Intelligent strategies for choosing the servers can be implemented to better balance the load among the HTTP servers. Many approaches are possible including RR and weighted.

Figure 15 illustrates the steps a web request goes through until the client gets a response back from the HTTP server. The web user types a web request into the web browser (1). The DNS server resolves the address and returns the IP address of the server, which in this case is the address of the redirectional server (2). When the request arrives to the redirectional server (3), it is examined and forwarded to the appropriate HTTP server (4,5). The HTTP server processes the request and replies to the web client (6).

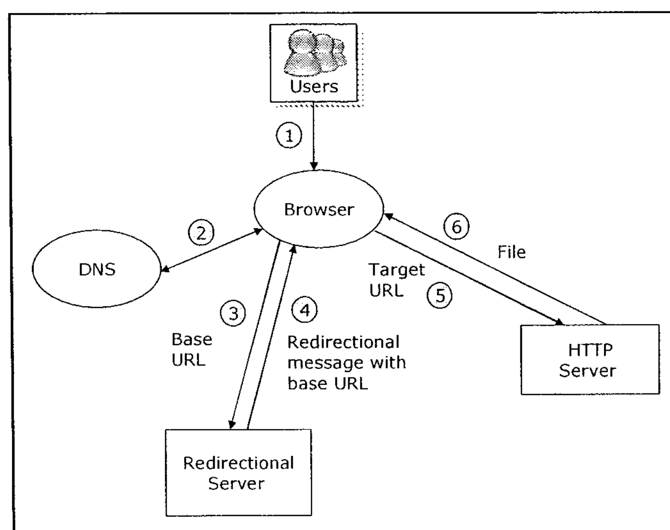


Figure 15: Redirection mechanism for HTTP requests

The authors implement load balancing by having each HTTP server report its load periodically to a load monitoring coordinator. If the load on a particular server exceeds a certain threshold, the load balancing procedure is triggered. Some portions of the content on the overloaded server are then moved to another

server with lower load. Next, the redirection information is updated in all redirection servers to reflect the data move.

The authors implemented a prototype of the redirection-based server architecture using one redirectional server and three HTTP servers. Measurements using the WebStone [81] benchmark demonstrate that the throughput scales up with the number of machines added. Measurements of connection times to various sites on the Internet indicate that additional connection to the redirection server accounts for a 20% increase in latency [55].

This architecture is implemented using COTS hardware and server software. Web clients see a single logical web server without knowing the actual location of the data, or the number of current servers providing the service. The administrator of the system partitions the document store among the available cluster nodes however using tedious and mechanical mechanisms and does not provide dynamic load balancing; rather, it requires the interference of a system administrator to move data to a different server(s) and to update manually the redirection rules.

One important characteristic of the implementation is the size of the mapping table. The HTTP server stores the redirection information in a table that is created when the server is started and stored in main memory. This mapping table is searched on every access to the redirection server. If the table grows too large, it increases the processing time searching in the redirection server.

The architecture assumes that all HTTP servers have disk storage, which is not very realistic as many real deployments take advantage of diskless nodes and network storage. The maintenance and update of all copies of the data is difficult. In addition, web requests require an additional connection between the redirection server and the HTTP server.

Other drawbacks of the architecture include the lack of redundancy at the main redirectional server. The authors did not focus on incorporating high availability capabilities within the architecture. In addition, since the architecture assumes a single redirectional server, there was no effort to investigate a single IP interface to hide all the redirectional servers. As a result, the redirectional server poses a SPOF and limits the performance and scalability of the architecture. Furthermore, the authors did not investigate the scaling limitation of the architecture. Overall, the architecture promises a limited level of scalability.

We classify the main inputs from this project in four essential points. First, the research provided us with a confirmation that a distributed architecture is the right way to proceed forward. A distributed architecture allows us to add more servers to handle the increase in traffic in a transparent fashion. The second input is the concept of specialization. Although very limited in this study, node specialization can

be beneficial where different nodes within the same cluster handle different traffic depending on the application running on the cluster nodes. The third input to our work relates to load balancing and moving data between servers. The redirectional architecture achieves load balancing by manually moving data to different servers, and then updating the redirection information stored on the redirectional server. This load-balancing scheme is a concept that can work for small configurations; however, it is not practical for large web clusters. The fourth input to our work is the need for a dynamic traffic distribution mechanism that is efficient and lightweight.

2.15.2 Scheduling Policies for Scalable Web Clusters

The researchers at the University of Roma are investigating the design of efficient and scalable scheduling algorithms for web dispatchers. The research investigates clusters as scalable web server platform. The researchers argue that one of the main goals for scalability of distributed web system is the availability of a mechanism that optimally balances the load over the server nodes [2]. Therefore, the project focuses on traditional and new algorithms that allow scalability of web server farms receiving peak traffic. The project recognizes that clustered systems are leading architectures for building web sites that require guaranteed scalable services when the number of users grows exponentially. The project defines a web farm as a web site that uses two or more servers housed together in a single location to handle requests [2].

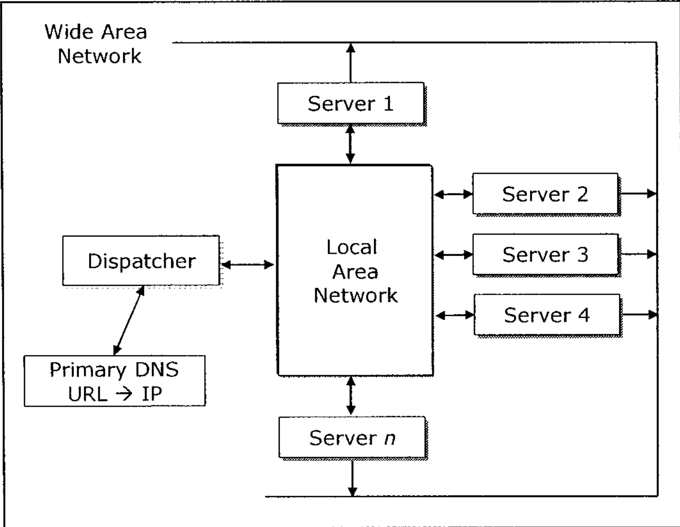


Figure 16: The web farm architecture with the dispatcher as the central component [2]

Figure 16, from [2], presents the architecture of the web cluster with n servers connected to the same local network and providing service to incoming requests. The dispatcher server connects to the same network as the cluster servers, provides an entry point to the web cluster, and retains transparency of the distributed architecture for the users [35]. The dispatcher receives the incoming HTTP requests and distributes it to the back-end cluster servers.

Although web clusters consist of several servers, all servers use one hostname site to provide a single interface to all users. Moreover, to have a mechanism that controls the totality of the requests reaching the site and to mask the service distribution among multiple back-end servers, the web server farm provides a single virtual IP address that corresponds to the address of front-end server(s). This entity is the dispatcher that acts as a centralized global scheduler that receives incoming requests and routes them among the back-end servers of the web cluster. To distribute the load among the web servers, the dispatcher identifies uniquely each server in the web cluster through a private address. The researchers argue that the dispatcher cannot use highly sophisticated algorithms for traffic distribution because it has to take fast decision for hundreds of requests per second. Static algorithms are the fastest solution because they do not rely on the current state of the system at the time of making the distribution decision. Dynamic distribution algorithms have the potential to outperform static algorithms by using some state information to help dispatching decisions. However, they require a mechanism that collects, transmits, and analyzes that information, thereby incurring in overheads.

The research project considered three scheduling policies that the dispatcher can execute [35]: random (RAN), round robin (RR) and weighted round robin (WRR). The project does not consider sophisticated traffic distribution algorithms to prevent the dispatcher from becoming the primary bottleneck of the web farm.

Based on modeling simulations, the project observed that burst of arrivals and skewed service times alone do not motivate the use of sophisticated global scheduling algorithms. Instead, an important feature to consider for the choice of the dispatching algorithm is the type of services provided by the web site. If the dispatcher mechanism has a full control on client requests and clients require HTML pages or submit light queries to a database, the system scalability is achieved even without sophisticated scheduling algorithms. In these instances, straightforward static policies are as effective as their more complex dynamic counterparts are. Scheduling based on dynamic state information appears to be necessary only in the sites where the majority of client requests are of three or more orders of magnitude higher than providing a static HTML page with some embedded objects.

The project observes that for web sites characterized with a large percentage of static information, a static dispatching policy such as round robin provides a satisfactory performance and load balancing. Their interpretation for this result is that a light-medium load is implicitly balanced by a fully controlled circular assignment among the server nodes that is guaranteed by the dispatcher of the web farm. When the workload characteristics change significantly, so that very long services dominate, the system requires dynamic routing algorithms such as WRR to achieve a uniform distribution of the workload and a more scalable web site. However, in high traffic web sites, dynamic policies become a necessity.

The researchers did not prototype the architecture into a real system and run benchmarking tests on it to demonstrate the performance, scalability, and high availability. In addition, the project did not design or prototype new traffic distribution algorithms for web servers; instead, it relied on existing distribution algorithms such as the DNS routing and RAN, RR, and WRR distribution. The architecture presents several single points of failure. In the event of the dispatcher failure, the cluster becomes unreachable. Furthermore, if a cluster node becomes unavailable, there is no mechanism is place to notify the dispatcher of the failure of individual nodes. Moreover, the dispatcher presents a bottleneck to the cluster when under heavy load of traffic.

The main input from this project is that dynamic routing algorithms are a core technology to achieve a uniform distribution of the workload and a reach scalable web cluster. The key is in the simplicity of the dynamic scheduling algorithms.

2.15.3 Scalable World Wide Web Server

The Scalable Web server (SWEB) project grew out of the needs of the Alexandria Digital Library (ADL) project at the University of California at Santa Barbara that has a potential to become the bottleneck in delivering digitized documents over high speed Internet [3]. For web-based network information systems such as digital libraries, the servers involve much more intensive I/O and heterogeneous processor activities. The SWEB project investigates the issues involved in developing a scalable web server on a cluster of workstations and parallel machines [2]. The objective of the project is to strengthen the processing capabilities of such servers by utilizing the power of computers to match the huge demand in simultaneous access requests from the Internet. The project aims to demonstrate how to utilize inexpensive commodity networks, heterogeneous workstations, and disks to build a scalable web server, and to attempt to develop dynamic scheduling algorithms for exploiting task and I/O parallelism adaptive to run time change of system resource load and availability. The scheduling component of the system

actively monitors the usages of processors, I/O channels, and the interconnection network to distribute effectively HTTP request across cluster nodes.

Figure 17, from [2], illustrates the SWEB architecture. The DNS routes the user requests to the SWEB processors using round robin distribution. The DNS assigns the requests without consulting the dynamically changing system load information. Each processor in the SWEB architecture contains a scheduler, and the SWEB processors collaborate with each other to exchange system load information. After the DNS sends a request to a processor, the scheduler on that processor decides whether to process this requests or assign it to another SWEB processor. The architecture uses URL redirection to achieve re-assignment. The SWEB architecture does not allow SWEB servers to redirect HTTP requests more than once to avoid the ping-pong effect.

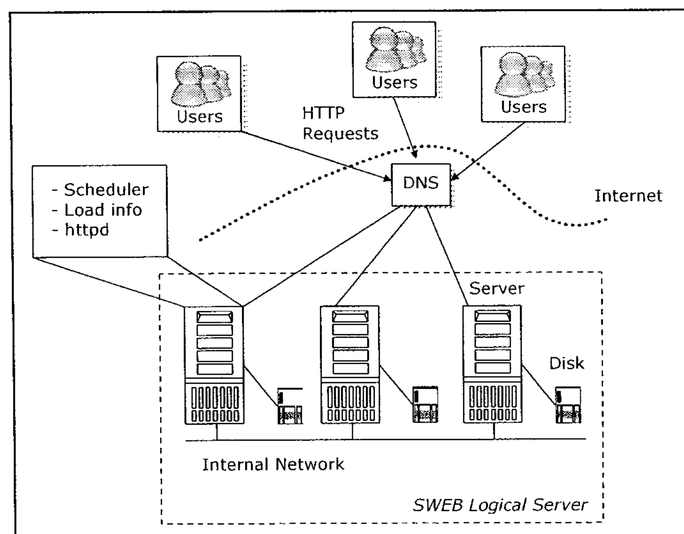


Figure 17: The SWEB architecture [2]

Figure 18, from [2], illustrates the functional structure of the SWEB scheduler [2]. The SWEB scheduler contains a HTTP daemon based on the source code of the NCSA HTTP for handling http requests, in addition to the broker module that determines the best possible processor to handle a given request. The broker consults with two other modules, the *oracle* module and the *loadd* module. The *oracle* module is a miniature expert system, which uses a user-supplied table that characterizes the processor and disk demands for a particular task. The *loadd* module is responsible for updating the system processor, network and disk load information periodically (every 2 to 3 seconds), and making the processors, which have not responded within the time limit, unavailable. When a processor leaves or joins the resource pool,

the *loadd* module is aware of the change as long as the processor has the original list of processor that was setup by the administrator of the SWEB system.

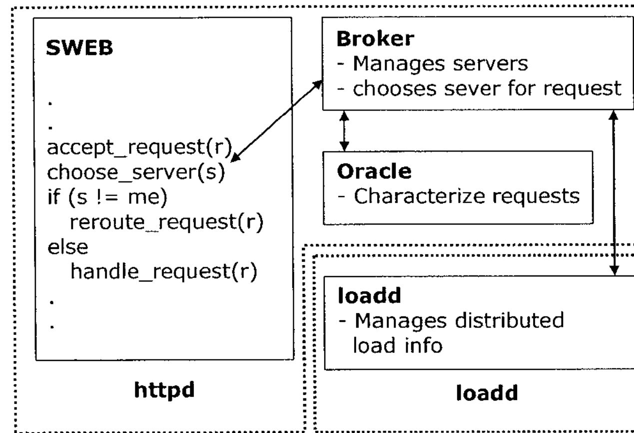


Figure 18: The functional modules of a SWEB scheduler in a single processor [2]

The SWEB architecture investigates several concepts. It supports a limited flavor of dynamism while monitoring the processor and disk usage on processors. The *loadd* module collects processor and disk usage information and feed back this information to the broker to make better distribution decisions. The drawback of this mechanism is that it does not report available memory as part of the metrics, which is as important as the processor information; instead it reports local disk information for an architecture that relies on a network file system for storage.

The SWEB architecture does not provide high availability features, making it vulnerable to single points of failures. The *oracle* module expects as input from the administrator a list of processors in the SWEB system and the processor and disk demands for a particular task. It is not able to collect this information automatically. As a result, the administrator of the cluster interferes every time we need to add or remove a processor.

The SWEB implementation modified the source code of the web server and created two additional software modules [2][83]. The implementation is not flexible and does not allow the usage of those modules outside the SWEB specific architecture.

The researchers have benchmarked the SWEB architecture built using a maximum of four processors with an in-house benchmarking tool, not using a standardized tool such as WebBench with a standardized workload. The results of the tests demonstrate a maximum of 76 requests per second for 1 KB/s request

size, and 11 requests per seconds for 1.5 MB/s request size, which ranks low compared to our initial benchmarking results [2].

The project contributes to our work by providing us how-to on actively monitoring the usages of processor, I/O channels, and the network load. This information allows us to distribute effectively HTTP requests across cluster nodes. Furthermore, the concept of web cluster without master nodes, and having the cluster nodes provide the services master nodes usually provide, is a very interesting concept to consider for large clusters.

2.15.4 University of Nebraska-Lincoln: LSMAC and LSNAT

This research project recognizes that server responsiveness and scalability are important in client/server network environments. The researchers are considering clusters that use commodity hardware as an alternative to expensive specialized hardware for building scalable web servers. The project investigates different server infrastructures namely MAC-based dispatching (LSMAC) and IP-based dispatching (LSNAT), and focuses on the single IP interface approach [20]. The resulting implementations are the LSMAC and LSNAT implementations, respectively. LSMAC dispatches each incoming packet by directly modifying its media access control (MAC) addresses.

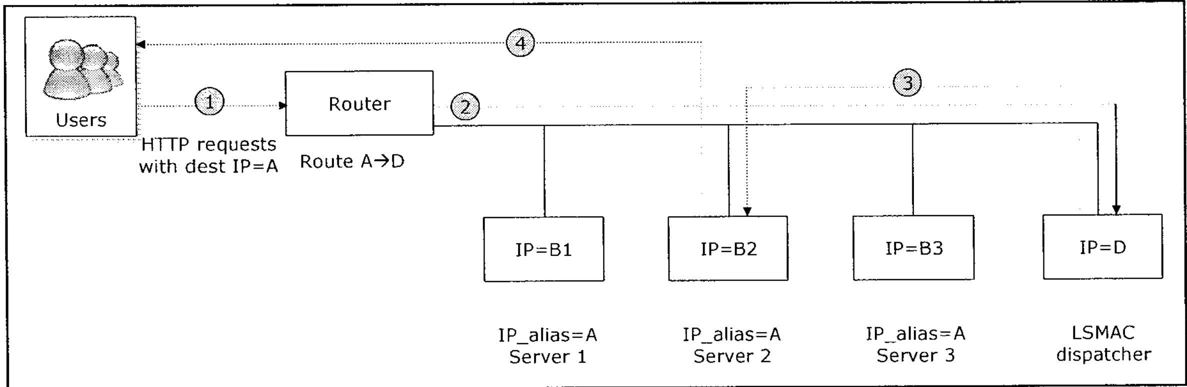


Figure 19: The LSMAC implementation [20]

Figure 19, from [20], presents the LSMAC approach. The following description of the LSMAC approach is a contribution from [20]. A client sends an HTTP packet (1) with A as the destination IP address. The immediate router sends the packet to the dispatcher at IP address A (2). Based on the load sharing algorithm and the session table, the dispatcher decides that this packet should be handled by the back-end-server, Server 2, and sends the packet to Server 2 by changing the MAC address and forwarding it (3). Server 2 accepts the packet and replies directly to the client (4).

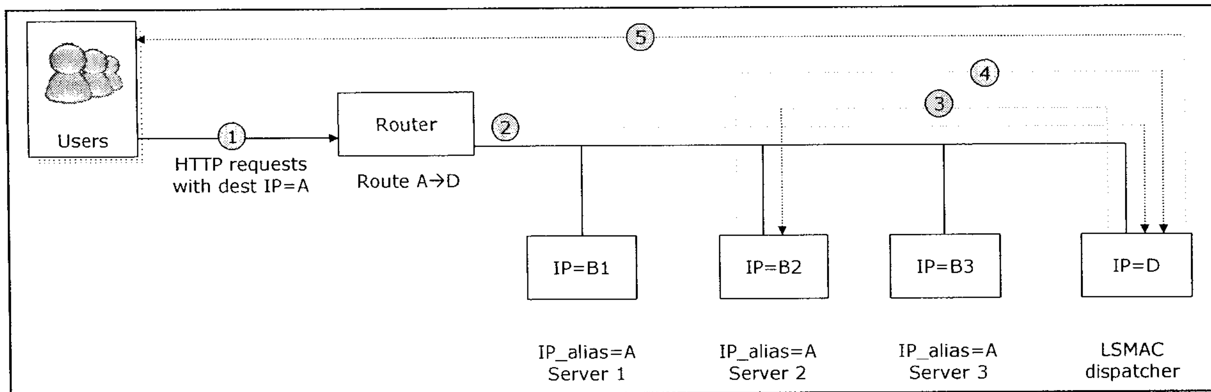


Figure 20: The LSNAT implementation [20]

Figure 20, from [20], illustrates the LSNAT approach. The LSNAT implementation follows RFC2391 [71] [64]. The following description of the LSNAT approach is a contribution from [20]. A client (1) sends an HTTP packet with A as the destination IP address. The immediate router sends the packet to the dispatcher (2) on A, since the dispatcher machine is assigned the IP address A. Based on the load sharing algorithm and the session table, the dispatcher decides that this packet should be handled by the back-end server, Server 2. It then rewrites the destination IP address as Server 2, recalculates the IP and TCP checksums, and sends the packet to Server 2 (3). Server 2 accepts the packet (4) and replies to the client via the dispatcher, which the back-end servers see as a gateway. The dispatcher rewrites the source IP address of the replying packet as A, and recalculates the IP and TCP checksums, and send the packet to the client (5).

The dispatcher in both approaches, LSMAC and LSNAT, is not highly available and presents a SPOF that can lead to service discontinuity. The work did not focus on providing high availability capabilities. The largest setup tested was a cluster that consists of four nodes. The authors did not demonstrate the scaling capabilities of the proposed architecture beyond four nodes [64][20]. The performance measurements were performed using the benchmarking tool WebStone [81]. The LSMAC implementation running on a four-node cluster averaged 425 transactions per second per traffic node [20]. The LSNAT implementation running on a four nodes cluster averaged 200 transactions per second per traffic node [20].

The proof-of-concept implementations do not provide adaptive optimized distribution. The dispatcher does not take into consideration the load of the traffic nodes nor their heterogeneous nature to optimize its traffic distribution. It assumes that all the nodes have the same hardware characteristics such as the same processor speed and memory capacity.

2.15.5 Harvard Array of Clustered Computers (HACC)

The goal of the HACC project is to design and develop cluster architecture for scalable and cost effective web servers [83]. In [83], the authors discuss the approach that places a router called *IP Sprayer* between the Internet and a cluster of web servers.

Figure 21, from [83], illustrates the architecture of the web cluster with the IP Sprayer. The IP Sprayer is responsible for distributing incoming web traffic evenly between the nodes of the cluster. A number of commercial products, such as the Cisco Local Director and the F5 Networks Big IP employ this approach to distribute web site requests to a collection of machines typically in a round robin fashion.

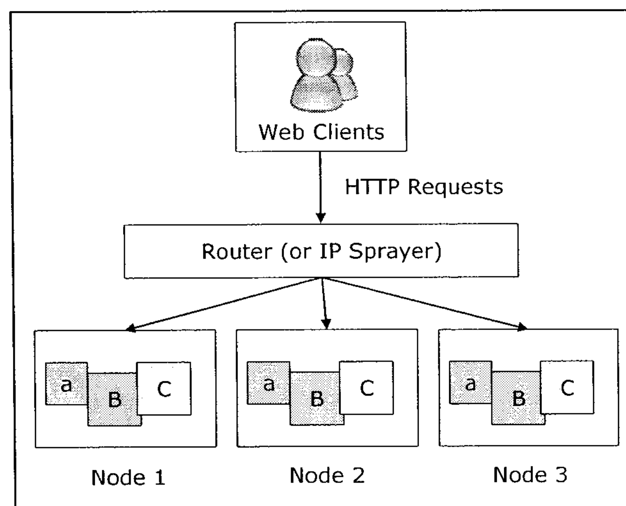


Figure 21: The architecture of the IP sprayer [83]

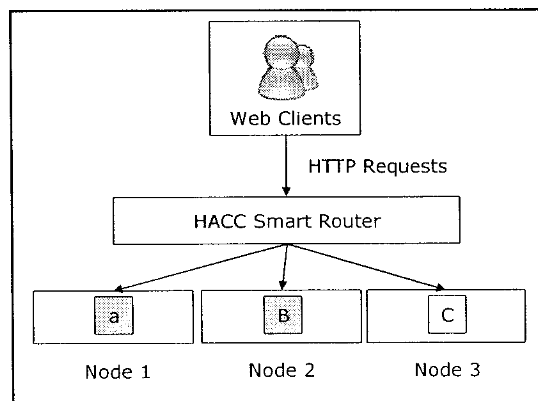


Figure 22: The architecture with the HACC smart router [83]

The HACC architecture focuses on locality enhancements by dividing the document store among the cluster nodes and dynamic traffic distribution. Rather than distributing requests in a round robin fashion,

the HACC smart router distributes requests so that as to enhance the inherent locality of the requested documents in the server cluster.

Figure 22, from [83], illustrates the concept of the HACC smart router. Instead of being responsible for the entire working set, each node in the cluster is responsible for only a fraction of the document store. The size of the working set of each node decreases each time we add a node to the cluster, resulting in a more efficient use of resources per node. The smart router uses an adaptive scheme to tune the load presented to each node in the cluster based on that node's capacity, so that it can assign each node a fair share of the load. The idea of HACC bears some resemblance to the affinity based scheduling schemes for shared memory multiprocessor systems [79][70], which schedule a task on a processor where relevant data already resides.

2.15.5.1 HACC Implementation

The main challenge in realizing the potential of the HACC design is building the Smart Router, and within the Smart Router, designing the adaptive algorithms that direct requests at the cluster nodes based on the locality properties and capacity of the nodes [83].

The smart router implementation consists of two layers: the low smart router (LSR) and the high smart router (HSR). The LSR corresponds to the low-level kernel resident part of the system and the HSR implements the high-level user-mode brain of the system. The authors conceived this partitioning to create a separation of mechanism and policy, with the mechanism implemented in the LSR and the policy implemented in the HSR.

The Low Smart Router: The LSR encapsulates the networking functionality. It is responsible for TCP/IP connection setup and termination, for forwarding requests to cluster nodes, and forwarding the result back to clients. The LSR listens on the web server port for a connection request. When the LSR receives connection request, TCP passes a buffer to the LSR containing the HTTP request. The HSR extracts and copies the URL from the request. The LSR queues all data from this incoming request and waits for the HSR to indicate which cluster node should handle the request. When the HSR identifies the node, the LSR establishes a connection with it and forwards the queued data over this connection. The LSR continues to ferry data between the client and the cluster node serving the request until either side closes the connection.

The High Smart Router: The HSR monitors the state of the document store, the nodes in the cluster, and properties of the documents passing through the LSR. It uses this information to decide how to distribute

requests over the HACC cluster nodes. The HSR maintains a tree that models the structure of the document store. Leaves in the tree represent documents and nodes represent directories. As the HSR processes requests, it annotates the tree with information about the document store to the applied in load balancing. This information could include node assignment, document sizes, request latency for a given document, and in general, sufficient information to make an intelligent decision about which node in the cluster should handle the next document request. When a request for a particular file is received for the first time, the HSR adds nodes representing the file and newly reached directories to its model of the document store, initializing the file's node with its server assignment. In the current prototype, incoming new documents are assigned to the least loaded server node. After the first request for a document, subsequent requests go to the same server and though improve the locality of references.

Dynamic Load Balancing: Dynamic load balancing is implemented using Windows NT's performance data helper (PDH) interface [52]. The PDH interface allows collecting a machine's performance statistics remotely. When the smart router initializes, it spawns a performance monitoring thread that collects performance data from each cluster node at a fixed interval. The HSR then uses the performance data for load balancing in two ways. First, it identifies a least loaded node and assigns new requests to it. Second, when a node becomes overloaded, the HSR tries to offload a portion of the documents for which the overloaded node is responsible to the least loaded node.

2.15.5.2 HACC Evaluation and Lessons Learned

There are several drawbacks to the HACC architecture. The architecture is vulnerable to SPOF, has limited performance and scalability, and suffers from various challenges in aspects such as data distributions, in addition to specific implementation issues.

Firstly, the smart router is a SPOF. In the HACC architecture if the smart router becomes unavailable because of a software or hardware error, the HACC cluster becomes unusable. Secondly, the performance and scalability of the HACC cluster with the smart router scalability are limited. The benchmarking tests demonstrate that the prototype of the smart router is capable of handling between 400 and 500 requests per second (requests are of size 8 KB) [83].

The limited performance of the smart router suggests that the HACC cluster cannot scale because of the limitation imposed by the bottleneck at the smart router level. In addition, the prototype of the HACC architecture consists of one node acting as the smart router and three nodes acting as cluster nodes, suggesting a limited configuration. Furthermore, the HACC architecture relies on distributing data among

the different cluster nodes; we cannot perform software of hardware updates while the HACC cluster is in operation. This approach is not realistic for systems that are in operation and require an update to their data store online. In addition, the tree-structured name space only works for the case when the structure of the document store is hierarchical. Moreover, the *Keep Alive* feature of the HTTP poses some potential problems with the Smart Router. If we enable the *Keep Alive* option, the browser allows reusing the TCP connection for subsequent requests that the smart router does not intercept, which interfere with the traffic distribution decision of the smart router.

2.15.6 IBM Scalable and Highly Available Web Server

IBM Research is investigating the concept of a scalable and highly available web server that offers web services via a Scalable Parallel (SP-2) system, a cluster of RS/6000 workstations [19][72] . The goal is to support a large number of concurrent users, high bandwidth, real time multimedia delivery, fine-grained traffic distribution, and high availability [19]. The server will provide support for large multimedia files such as audio and video, real time access to video data with high access bandwidth, fine-grained traffic distribution across nodes, as well as efficient back-end database access. The project is focusing on providing efficient traffic distribution mechanisms and high availability features. The server achieves traffic distribution by striping data objects across the back-end nodes and disks. It achieves high availability by detecting node failures and reconfiguring the system appropriately. However, there is no mention of the time to detect the failure and to recover.

Figure 23, from [19], illustrates the architecture of the web cluster. The architecture consists of a group of nodes connected by a fast interconnect. Each node in the cluster has a local disk array attached to it. The disks of a node either can maintain a local copy of the web documents or can share it among nodes. The nodes of the cluster are of two types: front-end (delivery) nodes and back-end (storage) nodes. The round robin DNS is used to distribute incoming requests from the external network to the front-end nodes, which also run *httpd* daemons. The logical front-end node then forwards the required command to the back-end nodes that have the data (document), using a shared file system. Next, the *httpd* daemons send the results to the front-end nodes through the switch and then the results are transmitted to the user. The front-end nodes run the web daemons and connect to the external network. To balance the load among them, the client spread the load across nodes using RR DNS. All the front nodes are assigned a single logical name and the RR DNS maps the name to multiple IP addresses.

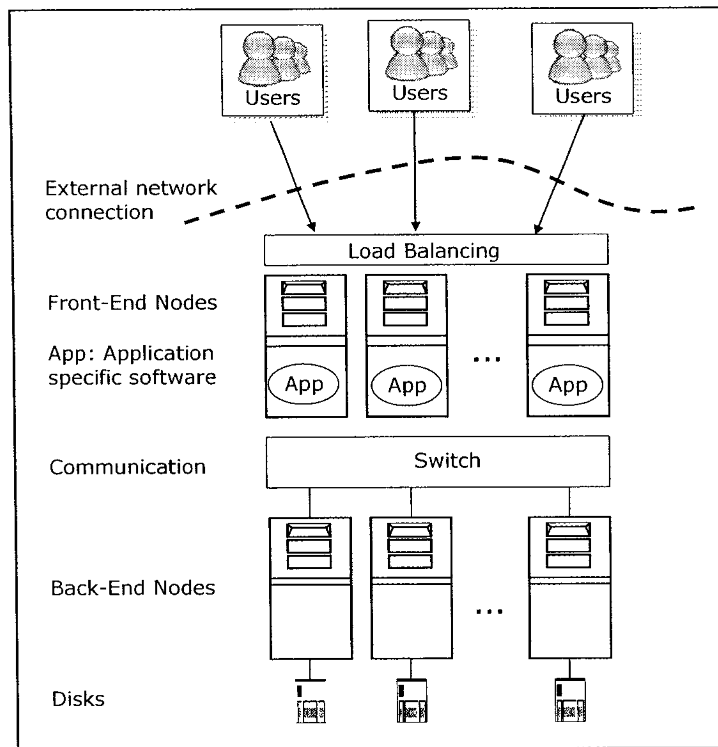


Figure 23: The two-tier server architecture [19]

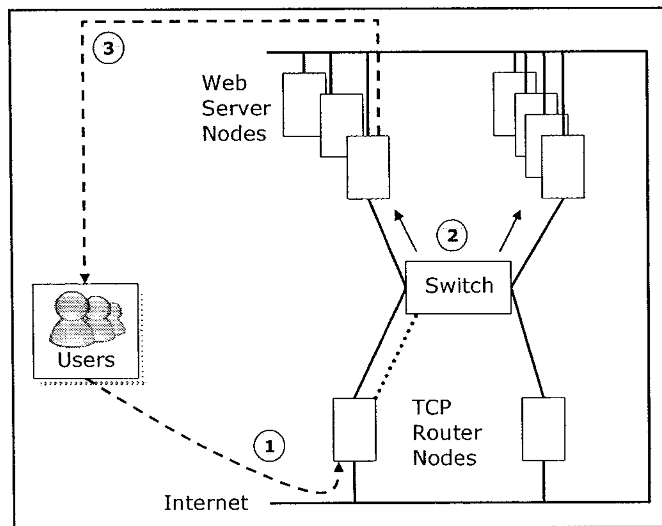


Figure 24: The flow of the web server router [19]

Figure 24, from [19], illustrates another approach for achieving traffic distribution. One or more nodes of the cluster serve as TCP routers, forwarding client requests to the different frond-end nodes in the cluster in a round robin order. The name and IP address of the router is public, while the addresses of the other

nodes in the cluster are private. If there is more than one router node, a single name is used and the round robin maps the name to the multiple routes. The flow of the web server router (Figure 24) is as follows. When a client sends requests to the router node (1), the router node forwards (2) all packets belonging to a particular TCP connection to one of the server front-end nodes. The router can use different algorithms to select which node to route to, or use round robin scheme. The server nodes directly reply to the client (3) without using the router. However, the server nodes change the source address on the packets sent back to the client to be that of the router node. The back-end nodes host the shared file system used by the front-ends to access the data.

2.15.6.1 Evaluation

There are five main drawbacks to the architecture: limited traffic distribution performance, limited scalability, lack of high availability capabilities, the presence of several SPOF, and the lack of a dynamic feedback mechanism.

The architecture relies on round robin DNS to distribute traffic among server nodes. The scheme is static, does not adjust based on the load of the cluster nodes, and does not accommodate the heterogeneous nature of the cluster nodes. The authors proposed an improved traffic distribution mechanism [72] that involves changing packet headers but still relies on round robin DNS to distribute traffic among router server nodes. The concept was prototyped with four front-end nodes and four back-end node. The project did not demonstrate if the architecture is capable of scaling beyond four traffic nodes and if failures at node level are observed and accommodated for dynamically. The architecture does not provide features that allow service continuity. The switch as shown in Figure 23 and Figure 24 is a SPOF. The network file system where data resides is also vulnerable to failures and presents another SPOF. Furthermore, the architecture does not support a dynamic feedback loop that allows the router to forward traffic depending on the capabilities of each traffic node.

2.15.7 Linux Virtual Server

The LVS is an open source project to cluster many servers together into one virtual server. It implements a layer four switching in the Linux kernel that allows the distribution of TCP and UDP sessions between multiple real servers. The real servers, also called traffic nodes, interconnect through either a local area network or a geographically dispersed wide area network. The front-end of the real servers is the LVS director. The LVS director is the load-balancing engine, and it runs on the master cluster processor(s). It

provides IP-level traffic distribution to make parallel services of the cluster appear as a virtual service on a single IP address. All requests come to the front-end LVS director, who owns the virtual IP address, and then the LVS director distributes the traffic among the real servers. LVS provides three implementations of its IP load-balancing techniques based on NAT, IP tunneling, and DR. We experimented with both the NAT and DR methods. We did not test the IP tunneling method for two reasons: the implementation of the IP tunneling method is still experimental and it does not offer added advantage over the DR method.
The following sub-sections discuss the three implementations of the LVS, with a focus on NAT and DR, and present the results of benchmarking the LVS cluster using NAT and then using DR. The material presented in the subsections 2.15.7.1, 2.15.7.2, and 2.15.7.3 are paraphrased from [74] and [32] with adaptations to focus on the workings on the implementations.

2.15.7.1 LVS via NAT

Network address translation (NAT) relies on manipulating the headers of Internet protocols appropriately so that web clients believe they are contacting one IP address, and servers at different IP addresses believe that web clients are contacting them directly. LVS uses this NAT feature to build a virtual server; parallel services at the different IP addresses can appear as a virtual service on a single IP address via NAT. A hub or a switch interconnects the master node(s), which run LVS, and the real servers. The real servers usually run the same service and provide access to the same contents, available on a shared storage device through a distributed file system.
Figure 25, from [74], illustrates the process of address translation in LVS. The description of the LVS NAT approach presented in this section is paraphrased from [74]. The user first accesses the service provided by the server cluster (1). The request packet arrives at the load balancer through the external IP address. The LVS load balancer examines the packet's destination address and port number (2). If destination address of the packet and the port match a virtual server service offered by the LVS cluster, the scheduling algorithm, round robin by default, chooses a server from the LVS cluster to process the request, and adds the connection into the hash table. The hash table of the LVS load balancer records all the established connections. The load balancer server rewrites the destination address and the port of the packet to match those of the chosen server, and forwards the packet to the server. The server processes the request (3) and returns the reply to the LVS load balancer, and not to the web client. When an incoming packet belongs to this connection and the established connection exists in the hash table, the load balancer rewrites and forwards the packet to the chosen server. When the reply packets come back from the server

to the LVS load balancer, the load balancer rewrites the source address and port of the packets (4) to those of the virtual service, and submits the response back to the client (5). The LVS load balancer removes the connection record from the hash table when the connection terminates or timeouts.

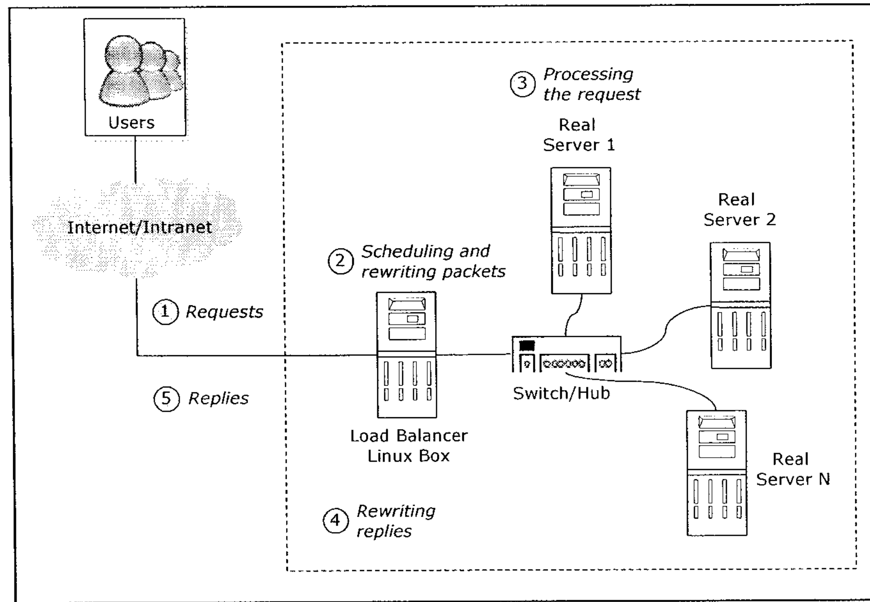


Figure 25: The architecture of the LVS NAT method [74]

2.15.7.2 LVS via DR

The direct routing method allows the real servers and the load balancer server(s) to share the virtual IP address. The load balancer has an interface configured with the virtual IP address. The load balancer uses this interface to accept request packets and directly route them to the chosen real server. Figure 26, from [74], illustrates the architecture of the LVS following the DR method. The description of the LVS DR approach presented in this section is paraphrased from [74].

When a user accesses a virtual service provided by the LVS cluster (1), the packet destined for the virtual IP address arrives to the LVS load balancer. The LVS load balancer examines (2) the packet's destination address and port. If the destination address of the packet and the port match a virtual service that is available on the LVS cluster, the LVS load balancer chooses a server (3) from the LVS cluster to serve the request and adds the connection into the hash table that records connections. Next, the LVS load balancer forwards the request to the chosen server. If the LVS balancer receives new incoming packets that belong to the ongoing connection, and the chosen server is available in the hash table, the LVS load balancer directly routes the packets to the server. When the server receives the forwarded packet, it finds

that the packet is for the address on its alias interface or for a local socket, so it processes the request (4) and returns the result directly to the web user (5). The LVS load balancer removes the connection record from the hash table when the connection terminates or timeouts.

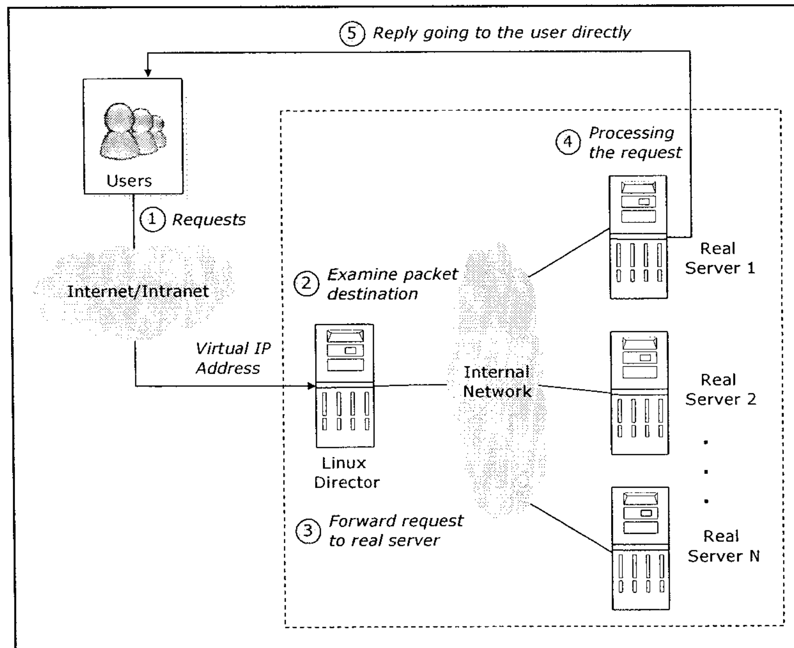


Figure 26: The architecture of the LVS DR method [74]

2.15.7.3 LVS via IP Tunneling

LVS supports IP Tunneling method, which allows packets addressed to an IP address to be redirected to another address, possibly on a different network. The LVS IP Tunneling method is discussed in [74]. In the context of layer four switching, the behavior of the IP Tunneling method is very similar to that of the DR method, except that when packets are forwarded to the server in the cluster, the LVS load balancer encapsulates the requests in an IP packet, rather than just manipulating the Ethernet frame.

The main advantage of using tunneling is that the servers can reside on a different network that the LVS load balancer. We did not experiment with the IP Tunneling method for two reasons: first, the implementation is unstable, and second the IP Tunneling method does not provide additional capabilities over the DR method. However, we present it for completion purpose.

2.15.7.4 Direct Routing versus Network Address Translation

We performed a test to benchmark the same web cluster using the NAT approach for traffic distribution and then using the direct routing approach. With direct routing, the LVS load balancer distributes traffic to the cluster processors, which in turn would reply directly to the web clients, without going through address translation. We run the same test on the same cluster running Apache, however, using different traffic distribution mechanisms.

Figure 27 illustrates the results of benchmarking two identical clusters with different LVS distribution methods in terms of successful requests per second.

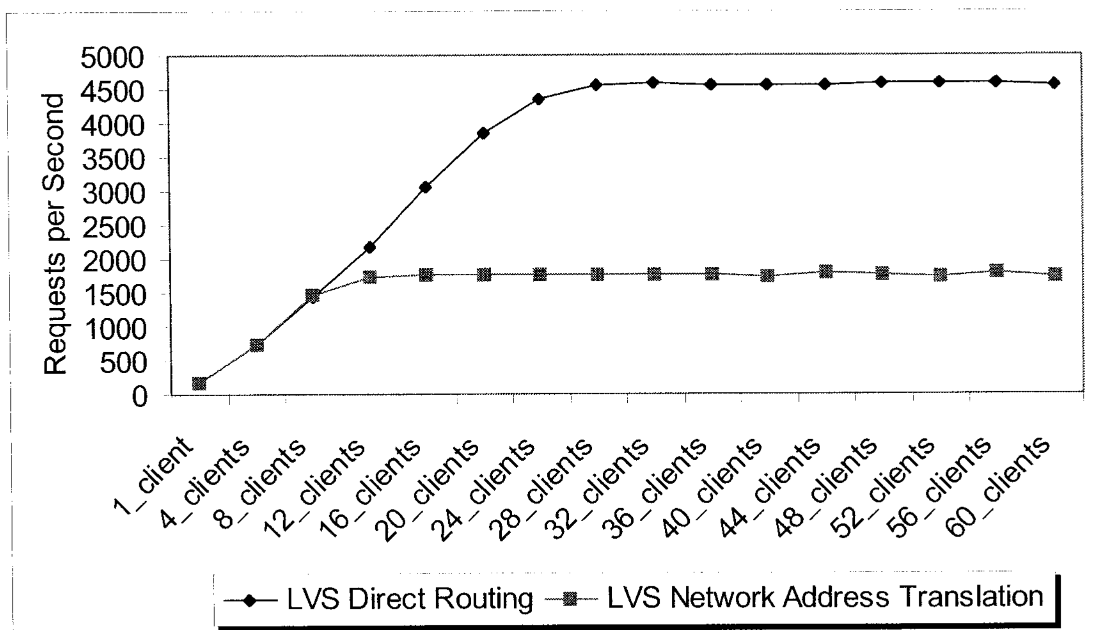


Figure 27: Benchmarking results of NAT versus DR

Following the NAT approach, the load balancer node saturates at approximately 2,000 requests per second compared to 4,700 requests per second with the DR method. In both tests with NAT and DR, the bottleneck occurs at the load balancer node that was unable to accept more traffic and distribute it to the traffic servers. Instead, the LVS director was not accepting incoming connections resulting in unsuccessful requests. This test demonstrates that the DR approach is more efficient than the NAT approach and allows better performance and scalability. In addition, it demonstrates the bottleneck at the director level of the LVS.

2.15.7.5 Benchmarking an LVS Cluster

The goal of the benchmarking activities was to demonstrate performance and scalability of an LVS cluster. We carried out six benchmark tests, starting with two processors and scaling up to 12 processors. Each node in the cluster runs a copy of the Apache web server software. The benchmarking tool, WebBench, generates the web traffic to the virtual IP address of the LVS cluster. Since the LVS DR method is more efficient and scalable than its NAT equivalence, we used it as the distribution mechanism for the incoming traffic.

Processors in the cluster	*Maximum requests per second*	*Transaction per second per processor*
2	1890	945
4	4012	1003
6	5847	974
8	7140	892
10	7640	764
12	8230	685

Table 6: The results of benchmarking with Apache

Table 6 presents the results of Apache benchmarking. For each cluster configuration, we present the maximum performance achieved by the cluster in terms of requests per second; the third column presents the average number of requests per second per processor. As we add more processors into the cluster, the number of requests per second per processor decreases (Table 6, 3rd column). We collected benchmarking results achieved (Table 6) for LVS clusters with two, four, six, eight, 10 and 12 processors. For each testing scenario, we recorded the maximum number of requests per second that each configuration can service. When we divide this number by the number of processors, we get the maximum number of request that each processor can process per second in each configuration.

Figure 28 illustrate the transaction capability per processor plotted against the cluster size. The figure illustrates that the curve is not flat; as we add more processors into the LVS cluster, the number of transactions per second per node drops. In the case of Apache, when the cluster scaled from two to 12 processors, the number of successful transactions per second per processor drops by a factor of -35% in comparison to the baseline performance of a single standalone node. The more processors we have in the cluster, the less performance we get per processor. These results present performance degradation. Theoretically, as we add more processors into the cluster, we would like to achieve linear scalability and

maintain the baseline performance per each node. In [22], we discuss the benchmarking environment and the benchmarking we conducted with LVS clusters.

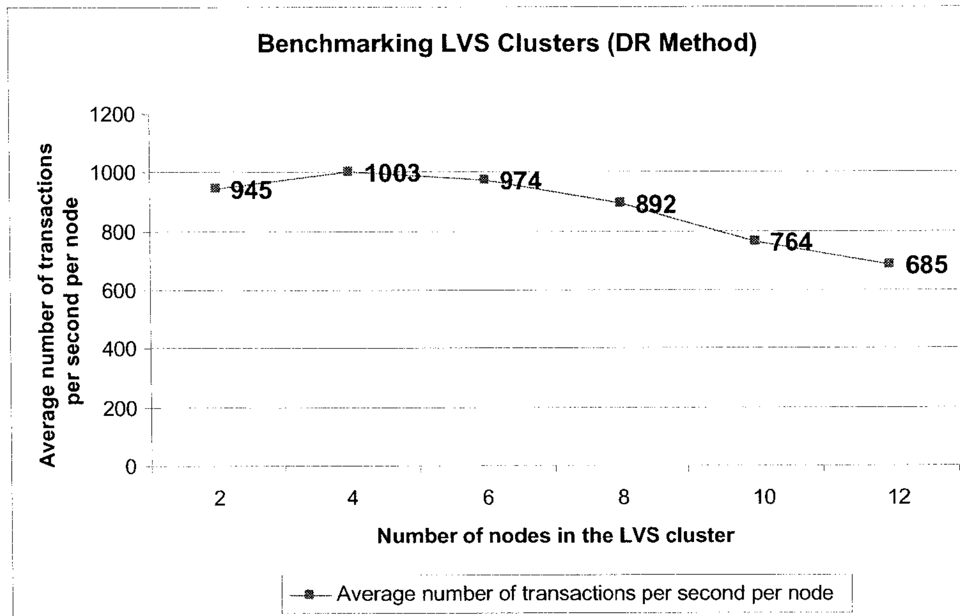


Figure 28: Scalability of LVS clusters consisting of up to 12 nodes running Apache

2.15.8 Discussion of Related Work and Lessons Learned

When we were evaluating related work, we decided on 11 criteria that are important in a highly available and scalable cluster architecture for web servers.

Table 7 illustrates how the surveyed works map to the 11 criteria. The 11 criteria are the following:

1. *Support for high availability at different cluster layers:* This criterion examines if the related work support high availability at all the layers inside the cluster such as at the front node layer, traffic nodes, storage, and data, networks, and connections.
2. *Support for over eight nodes in the proof-of-concept:* many of the surveyed work limited their proof-of-concept to four and eight nodes. We are looking for an architecture proof-of-concept that was demonstrated with more than eight nodes and that is capable of linear scalability.
3. *Maintained baseline performance as we increased the number of traffic nodes:* this criterion examines if the related work was able to maintain baseline performance as the number of nodes increases in the cluster.

4. *Support online operating system and software upgrades:* Several of the related work requires downtime associated with software upgrades for both the operating system and the software modules. This criterion examines if the related work assumes high availability with scheduled downtime or high availability without scheduled downtime.
5. *Support multiple redundancy models:* Some of the surveyed works limit their support for a certain number of redundancy models. This criterion examines if the related work supports the following redundancy models at all tiers of the architecture: 1+1 active/standby and active/active, N+M, and N-way.
6. *Uses common-off-the-shelf hardware and software:* This criterion examines if the proposed architecture in the related work require specialized hardware or software.
7. *Support dynamic traffic distribution:* This criterion indicates if the related work supports dynamic traffic distribution and does not rely exclusively of static algorithms.
8. *Provide mechanism to detect failures and trigger recovery:* This criterion examines if the related work provides mechanisms to detect failures in the front-end nodes, traffic nodes, data, and connections, and if it is capable of recovering from the failures and continue to provide service to end users.
9. *Support heterogeneous hardware:* Most surveyed work assume that the cluster consists of homogenous nodes. This criterion indicates if the work supports nodes with heterogeneous hardware configurations and if it is capable to maximize resource utilization on each node based on its configuration.
10. *System software is modular and usable in other environments:* This criterion indicates if the contributed software modules are useable outside the specific deployment use case or if they are useful only in one specific case.
11. *Provide a single entry to the cluster using software:* This criterion examines if the work relies on hardware to provide a single IP interface or if it provides it using software.

The answers in the table can be either 'Yes', 'No', or and empty field '-'. A 'Yes' indicates that the related work support this capability and that it was demonstrated in their proof-of-concept. A 'No' indicates that the related work did not support or demonstrate this feature or capability. Empty fields indicate that we did not have sufficient data or that the criterion was not within the focus of the work.

	IBM	SWEB	HACC	LVS	LS-MAC/NAT	Redirectional	SWC
Support for HA at different cluster layers (nodes, connectivity, service, data)	No	No	No	Yes	No	No	No
Architecture proof-of-concept tested with more than eight nodes	No	No	No	No	-	No	-
Maintain base line performance as we add more nodes	No	No	No	No	-	-	-
Online operating system and software upgrade	No	No	No	No	-	No	-
Support multiple redundancy models	No	No	No	No	-	No	No
Use common-off-the-shelf hardware and software	No	Yes	Yes	Yes	Yes	Yes	Yes
Support dynamic traffic distribution	No	Yes	Yes	Yes	No	No	No
Provide mechanism to detect failures and trigger recovery	Yes	No	-	Yes	Yes	No	No
Support heterogeneous cluster nodes hardware	No	Yes	Yes	Yes	Yes	No	-
System software is modular and can be used in other environments	Yes	No		Yes	Yes	Yes	-
A cluster IP Interface	No	Yes	Yes	Yes	Yes	No	Yes

Table 7: Evaluation of related work

As a result of this exercise, the realization was that very few of these works offered a comprehensive solution that met a large subset, or all the criteria that, when put together, offers a scalable and HA architecture for web servers. Instead, most of the works were focus on specific problem areas and provided many contributions in the specific challenge areas, but there was no focus on the comprehensive overall solution, which the HAS architecture aims to target.

This exercise was very useful and we have learned many lessons. For traffic distribution, lesson was to aim for a dynamic distribution using a lightweight implementation that takes into consideration the load on the traffic nodes and their heterogeneous nature. When it comes to high availability, the input was to keep track of the availability of cluster nodes, support online kernel and application upgrades, and eliminate single points of failure across all the layers of the architecture. Regarding the transparency of the cluster nodes, the cluster technology should be invisible to the web server software, not just users. As for routing approaches, the lessons were to avoid NAT approaches and aim for a direct routing approach. As for the proof of concept activity, the input was to avoid complex implementation, focus on flexibility and simplicity, and provide software modules that are usable outside the specific usage model. Furthermore, other input was a recommendation to develop an automated cluster installation infrastructure to allow easier testing with multiple configurations.

Chapter 3
Highly Available and Scalable Web Server Cluster Architecture

This chapter presents the HAS cluster architecture, its tiers and characteristics. It discusses the architecture components, presents how they interact with each other, illustrates the supported redundancy models, and discusses the various types of cluster nodes and their characteristics. Furthermore, the chapter includes examples of sample deployments of the HAS architecture as well as case studies that demonstrate how the architecture can scale to support increased traffic. The chapter addresses the subject of fault tolerance and the high availability. It discusses the traffic management scheme responsible for dynamic traffic distribution and discuss the cluster virtual IP interface, which presents the HAS architecture as a single entity to the outside world. The chapter concludes with the scenario view of the HAS architecture and examines several use case scenarios.

3.1 Summaries of Contributions

To the best of our knowledge, this work contributes the first highly available and scalable architecture for web server clusters that follows the building block approach and demonstrates close to linear scaling for up to 16 nodes, maintains over 96% of baseline performance, and supports high availability at different layers of the cluster leading to continues service availability.

The HAS architecture contributes a dynamic traffic distribution mechanism that monitors the load of the traffic nodes, compute its load index through an original formula using multiple metrics, and provides this information to the executor of the distribution. The distribution mechanism does not assume that all nodes in the cluster have the same hardware configuration, and achieves efficient resource utilization taking into consideration the nodes hardware configuration.

The HAS architecture contributes mechanisms to detect failures of traffic nodes, master nodes, file system, Ethernet cards, traffic clients, web server software, and ongoing connections. These mechanisms are embedded across all layers of the cluster, and provide correction action when the failures are detected.

The HAS architecture contributes a high availability extensions to the network file server which allows it to provide highly available storage to all the HAS cluster nodes. Furthermore, the work contributes a specialized mount program that allows mounting of two redundant network file servers over the same mount point.

The HAS architecture contributes the Ethernet redundancy daemon which monitors the link status of the primary Ethernet port and switches control to the second Ethernet port upon the failure of the first port.
The HAS architecture contributes a keep-alive mechanism, which allows the master nodes to know when a traffic node is available for service and when it is not available because hardware or software failures.
The HAS architecture provides continuous service through supporting online operating system and software upgrade for maintenance activities, and by providing the capability to synchronize connections, and to continue servicing ongoing connections even in the event of software or hardware failures.
The HAS architecture appears to be the first architecture for HA and scalable web servers that support multiple redundancy models in each tier of the architecture and independently from other tiers.
This HAS architecture offers significant contributions to the HA-OSCAR project, whose architecture is based on the work presented in this dissertation.
The HAS architecture is the base architecture for clustered telecommunication servers as defined by the Carrier Grade standardization industry initiative at the Open Source Development Labs. The Carrier Grade initiative has adopted the HAS architecture as the base standard architecture for carrier grade servers running telecommunication applications.

3.2 The HAS Architecture

The HAS architecture consists of a collection of loosely coupled computing elements, referred to as nodes or processors, that form what appears to users as a single highly available web cluster. There are no shared resources between nodes with the exception of storage and access to networks. The HAS architecture allows the addition of nodes to the cluster to accommodate increased traffic, without performance degradation and while maintaining the baseline performance for up to 16 processors. Figure 29 illustrates the conceptual model of the architecture showing the three tiers of the architecture, the software components running on nodes inside each tier, and shows the supported redundancy models per tier. For instance, the high availability (HA) tier supports the 1+1 redundancy model (active/active and active/standby) and can be expanded to support the N+M redundancy model, with N nodes are active and M nodes are standby. Similarly, the scalability and service availability (SSA) tier supports the N-way redundancy model where all traffic nodes are active and servicing requests. We can expand this tier to support the N+M redundancy model. Sections 3.8, 3.9, and 3.10 discuss the supported redundancy models.

The HAS architecture is composed of three logical tiers: the high availability (HA) tier, the scalability and service availability (SSA) tier, and the storage tier. This section presents the architecture tiers at a high level.

The high availability tier: This tier consists of front-end systems called master nodes. Master nodes provide an entry-point to the cluster acting as dispatchers, and provide cluster services for all HAS cluster nodes. They forward incoming web traffic to the traffic nodes in the SSA tier according to the scheduling algorithm. Section 3.4.1 covers the characteristics of this tier. Section 3.16.1 presents the characteristics of the master nodes. Section 3.8 discusses the supported redundancy models of the HA tier.

The scalability and service availability tier: This tier consists of traffic nodes that run application servers. In the event that all servers are overloaded, the cluster administrator can add more nodes to this tier to handle the increased workload. As the number of nodes increases in this tier, the cluster throughput increases and the cluster is able to respond to more traffic. Section 3.5.2 describes the characteristics of this tier. Section 3.16.2 presents the characteristics of the traffic nodes. Section 3.9 discusses the supported redundancy models of the SSA tier.

The storage tier: This tier consists of nodes that provide storage services for all cluster nodes so that web servers share the same set of content. Storage sits on separate subnets from the HA and SSA tiers to reduce network traffic. Section 3.5.3 describes the characteristics of this tier and Section 3.16.3 describes the characteristics of the storage nodes. Section 3.10 presents the supported redundancy models of the storage tier. The HAS cluster prototype did not utilize specialized storage nodes. Instead, it utilized a contributed extension to the NFS to support HA storage.

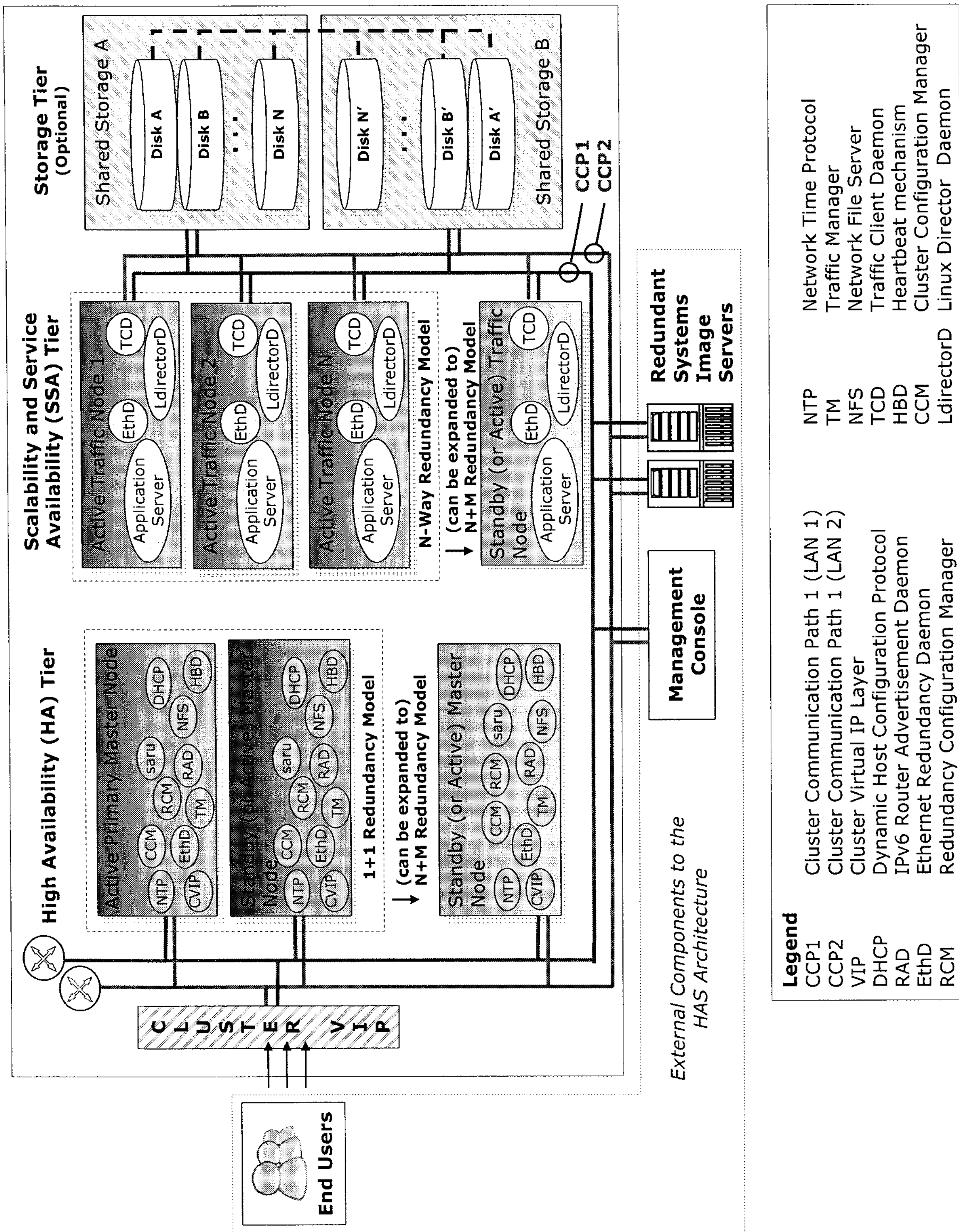


Figure 29: The HAS architecture

3.3 HAS Architecture Components

Each of the HAS architecture tiers consists of several nodes, and each node runs specific software component. A software component (or system software) is a stand-alone set of code that provides service either to users or to other system software. A component can be internal to the cluster and represents a set of resources contained on the cluster physical nodes; a component can also be external to the cluster and represents a set of resources that are external to the cluster physical nodes. Components can be either software or hardware components. Core components are essential to the operation of the cluster. Optional components are used depending on the usage and deployment model of the HAS cluster.

The HAS architecture is flexible and allows administrators of the cluster to add their own software components. The following sub-sections present the components of the HAS architecture, categorize the components as internal or external, discuss their capabilities, functions, input, output, interfaces, and describe how they interact with each other.

3.3.1 Architecture Internal Components

The internal components of the architecture include the master nodes, traffic nodes, storage nodes, routers, and local networks.

Master nodes are members of the HA tier. They provide a key entry into the system through the cluster virtual IP interface, and act as a dispatcher, forwarding incoming traffic from the cluster virtual IP interface to the traffic nodes located in the SSA tier. Moreover, master nodes also provide cluster services to the cluster nodes such as DHCP, NTP, TFTP, and NFS. Section 3.16.1 discusses the characteristics of master nodes.

Traffic nodes reside in the SSA tier of the HAS architecture. Traffic nodes run application servers, such as a web server, and they are responsible for replying to clients requests. Section 3.16.2 discusses the characteristics of traffic nodes.

Storage nodes are located in the storage tier, and provide HA shared storage using multi-node access to redundant mirrored storage. Storage nodes are optional nodes. If the administrators of the HAS cluster choose not to deploy specialized storage nodes, they can instead use a modified implementation of the NFS providing HA capabilities.

The architecture supports two local networks, also called cluster communication paths, to provide connectivity between all cluster nodes. Each of the networks connects to a different router.

3.3.2 Architecture External Components

External components to the cluster represent a set of resources that are external to the cluster physical nodes and include external networks, to which the cluster is connected, the management console through which we administer the cluster, and web users. External networks connect the cluster nodes to the Internet or the outside world. The management console is an external component to the cluster through which the cluster administrator logs in and performs cluster management operations. Web users, also called web clients, are the service requesters. A user can be a human being, an external device, or another computer system. Image servers are optional external components to the HAS architecture. We can use master nodes to provide the functionalities provided by the image servers (DHCP and TFTP services for booting and installation of traffic nodes). However, we recommend dedicating the resources on master nodes to serve incoming traffic.

3.3.3 Architecture Software Modules

The architecture consists of several system software modules. Some of the modules are essential to the operation of the cluster and run by default. However, we expect that across all possible deployments of highly available web clusters, there are trade-offs between available functions, acceptable levels of system complexity, security needs, and administrator preferences.

The HAS system software modules include: traffic client daemon, traffic manager daemon, cluster virtual IP interface, IPv6 router advertisement daemon, DHCP daemon, TFTP daemon, NTP daemon, HA NFS server daemon, heartbeat service, the Linux director daemon, cluster configuration manager, redundancy configuration manager, the connection synchronization manager, and the Ethernet redundancy daemon.

The traffic client daemon is a system software module that runs on all traffic nodes. It computes the `load_index` of traffic nodes, and reports it to the traffic managers running on the master nodes. The traffic client daemon allows the traffic manager running on master nodes to provide efficient traffic distribution based on the load of each traffic node. The metrics collected by the traffic client daemon are the processor load and memory usage. The implementation of the traffic client supports IPv6. Section 3.22.5 discusses the traffic client daemon.

The traffic manager daemon (TM) is a system software module that runs on the cluster master nodes. The traffic manager receives the `load_index` of the traffic nodes from the traffic client daemons, maintains the list of available traffic nodes and their load index, and executes the distribution of traffic to the traffic nodes based on the defined distribution policy in its configuration file. The current traffic manager

implementation supports round robin and the HAS distribution. However, the traffic manager can support more policies. The traffic manager supports IPv6. Section 3.22.3 discusses the traffic manager.

The cluster virtual IP interface (CVIP) is a core system module that runs on the master nodes in the HA tier of the HAS architecture. It masks the HAS architecture internals, and makes it appear as a single server to the external users, who are not aware of the internals of the cluster such as how many nodes exist or where the applications run. The CVIP allows a virtually infinite number of clients to reach a virtually infinite number of servers presented as a single virtual IP address, without impact on client or server applications. The CVIP operates at the IP level, enabling applications that run on top of IP to take advantage of the transparency it provides. The CVIP supports IPv6. Section 3.20 discusses the cluster virtual IP interface.

The IPv6 router advertisement daemon is an optional system software module that is used only when the HAS architecture needs to support IPv6. It offers automatic IPv6 configuration for network interfaces for all cluster server nodes. It ensures that all the HAS cluster nodes can communicate with each other and with network elements outside the HAS architecture over IPv6.

The cluster administrator has the option of using the DHCP daemon (an optional server service) to configure IPv4 addresses to the cluster server nodes.

The TFTP service daemon is an optional software module used in collaboration with the DHCP service to provide the functionalities of an image server. The image server provides an initial kernel and ramdisk image for diskless server nodes within the HAS system. The TFTP daemon supports IPv6 and is capable of receiving requests to download kernel and ramdisk images over IPv6.

The NTP service is a required system service used to synchronize the time on all cluster server nodes. It is essential to the operation of other software modules that rely on time stamps to verify if a node is in service or not. Alternatively, we can use a time synchronization service provided by an external server located on the Internet. However, this poses security risks and it is not recommended.

As for storage, we contribute HA extension to the Linux kernel NFS server implementation that supports NFS redundancy and eliminates the NFS server as a SPOF. Section 3.15 discusses the storage models and the various available possibilities.

The heartbeat service (HBD) runs on master nodes and sends heartbeat packets across the network to the other instances of heartbeat (running on other master nodes) as a keep-alive type message. When the standby master node no longer receives heartbeat packets, it assumes that the active master node is dead, and then the standby node becomes primary. The heartbeat mechanism is a contribution from the Linux-

HA project [66]. We have contributed enhancements to the heartbeat service to accommodate for the HAS architecture requirements. Section 3.18 discusses the heartbeat service and its integration with the HAS architecture.

The Linux director daemon (*LDirectord*) is responsible for monitoring the availability of the web server application running on the traffic nodes by connecting to them, making an HTTP request, and checking the result. If the *LDirectord* module discovers that the web server application is not available on a traffic node, it communicates with the traffic manager to ensure that the traffic manager does not keep the traffic node with a failing application on its list of available traffic nodes. Section 3.19 presents the functionalities of the LDirectord.

The cluster configuration manager (CCM) is a system software that manages all the configuration files that control the operation of the HAS architecture software modules. It provides a centralized single access point for editing and managing all the configuration files. For the purpose of this work, we did not implement the cluster configuration manager. However, it is a high priority future work. At the time of publication, with the HAS architecture prototype, we maintain the configuration files of the various software modules on the network file system.

The redundancy configuration manager (RCM) is responsible for switching the redundancy configuration of each cluster tier from one redundancy configuration to another, such as from the 1+1 active/standby to the 1+1 active/active. It is also responsible for switching service between modules when the cluster tiers follow the N+M redundancy model. Therefore, it should be aware of active nodes in the cluster and their corresponding standby nodes. For the purpose of this work, we did not implement the redundancy configuration manager. Section 5.2.4 discusses the RMC as a future work item.

The connection synchronization manager provides the capabilities to synchronize the ongoing connections between the active master node and the standby master node. As a result, the cluster can minimize and eliminate the situation of lost connections caused by the failure of an active master node. Section 3.21 discusses these capabilities.

The Ethernet redundancy daemon (EthD) is a contributed system software that handles network adapter failures. When a network adapter (Ethernet card) fails, the Ethernet redundancy daemon swaps the roles of the active and standby adapters on that node.

3.4 HAS Architecture Tiers

The following subsections describe the three HAS architecture tiers illustrated in Figure 29: the high availability tier, scalability and service availability tier, and the storage tier.

3.4.1 The High Availability Tier

The HA tier consists of master nodes that act as a dispatcher for the SSA tier. The role of the master node is similar to a connection manager or a dispatcher. The HA tier does not tolerate service downtime. If the master nodes are not available, the traffic nodes in the SSA tier become unreachable and as a result, the HAS cluster cannot accept incoming traffic. The primary functions of the nodes in this tier are to handle incoming traffic and distribute it to traffic nodes located in the SSA tier, and to provide cluster infrastructure services to all cluster nodes.

The HA tier consists of two nodes configured following the 1+1 active/standby redundancy model. The architecture supports the extension redundancy model of this tier to the 1+1 active/active redundancy model. With the 1+1 active/active redundancy model, master nodes share servicing incoming traffic to avoid bottlenecks at the HA tier level. Another possible extension is the support of the N-way and the N+M redundancy models, which allows the HA tier to scale the number of master nodes one at a time. However, it requires a complex implementation and it is not yet supported it. Section 3.7 describes the supported redundancy models.

The HA tier needs to determine the status of traffic nodes and be able to reliably communicate with each traffic node. The HA tier uses traffic managers to receive load information from traffic nodes. Section 3.5.1 presents the characteristics of the HA tier. Section 3.7 discusses the redundancy models supported by this tier.

3.4.2 The Scalability and Service Availability Tier

The scalability and service availability (SSA) tier consists of traffic nodes that run web server applications. The main task of the traffic nodes is to receive, process, and reply to incoming requests. This tier provides a redundant application execution cluster. In the event that the traffic manager daemon running on the master nodes ceases to receive load notification from the traffic client daemon, then the traffic manager remove that specific traffic node from their list of available traffic nodes. Section 3.24.8 discusses this scenario.

If the traffic node becomes unresponsive (after a timeout limit defined in the configuration file), the traffic manager declares the traffic node unavailable. Section 3.24.8 discusses this use case and presents its sequence diagram. The supported redundancy model is the N-way model: all nodes are active and there are no standby nodes. As such, redundancy is at node level. Section 3.5.2 presents the characteristics of the SSA tier. Section 3.9 discusses the redundancy models supported by this tier.

3.4.3 The Storage Tier

The storage tier consists of specialized nodes that provide shared storage for the cluster. This tier differs from the other two tiers in that it is an optional tier, and not required if the master nodes are providing shared storage through a distributed file system. Since storage is not the focus of this dissertation, we do not explore this area in depth. Instead, our interest in this area is limited to providing a highly available storage and access to storage independently from the storage techniques. For instance, to avoid single points of failure, we provide redundant access paths from the cluster nodes to the shared storage. Our efforts in this area include a contribution of a HA extension to the Network File Server (NFS). In the modified version of the NFS, the file system supports two redundant servers, where one server failure is transparent to the users. Master nodes can provide the storage service using the highly available NFS implementation that requires a modified implementation of the mount program. Cluster nodes use the modified mount program to mount simultaneously two network file servers at the same mounting point. Section 3.5.3 presents the characteristics of the storage tier. Section 3.10 presents the redundancy models supported by this tier.

3.5 Characteristics of the HAS Cluster Architecture

The HAS architecture is the combination of the HA, SSA, and storage tiers. The architecture inherits the characteristics of all tiers, discussed in Sections 3.5.1, 3.5.2, and 3.5.3. Furthermore, the HAS architecture posses holistic properties that belong to the HAS cluster as a whole, rather than any single subsystem or software module. These properties emerge from the way the system as a whole is constructed. These properties are not located in one part of the system, but they spread throughout the system, and depend on doing little things right and with many small decisions. The holistic properties of the HAS architecture include:

- *Scalability:* Adding more nodes to the HAS cluster increases the cluster throughput and its capacity. In the following subsection, we discuss the concept of incremental scalability per each tier of the architecture, where we are able to scale each tier independently of the others and based on our needs.
- *Adaptability and Modularity:* The HAS architecture does not require specialized hardware or commercial software modules; instead, the architecture is implementable using COTS hardware and open source software modules. The software modules follow the building block approach. They are independent of each other and talk to each other through defined interfaces. This approach allows us to re-use these software modules in other environments where their functionalities are needed. n addition, the building block approach allows easier testing of the independent software modules, permits the accommodation of new requirements at the software module level, and without architectural changes.
- *Maintainability and Flexibility:* The HAS architecture is very flexible in terms of configuration, upgrades and maintenance, and support for redundancy models. In fact, the HAS architecture support online software and hardware upgrade without any associated downtime.
- *Robustness:* With the HAS architecture, we are able to continue to provide service under heavy load. The benchmarking activities presented in Chapter 4 demonstrate that we are able to maintain close to steady state baseline performance after more than seven hours of traffic generation above the threshold or the capacity of the HAS cluster.

The following sub-sections discuss the characteristics of each architecture tier.

3.5.1 Characteristics of the HA Tier

The HA tier consists of two master nodes that provide cluster services to traffic nodes and direct incoming requests to the SSA tier. The HA tier has the following characteristics:

- *No single point of failure:* If a master node fails and becomes unavailable, the standby master node takes over in a transparent fashion. Section 3.24.7 presents the sequence diagram of this case scenario. The HA tier provides node level redundancy. All software modules are redundant and available on both master nodes. If one master node fails, the other master node handles incoming traffic, and provides cluster services to cluster nodes. Section 3.7 discusses the redundancy model of the HA tier.
- *Sensible repair and replacement model:* The architecture allows the upgrade or replacement of failed modules without affecting the service availability. For instance, if the active master node requires a

processor upgrade, then the administrator gracefully switches control from the active master node to the standby node; the active master becomes standby and the standby master becomes active. The cluster continues to provide services without associated downtime and without performance penalties, with the exception that load sharing among master nodes (1+1 active/active) is not possible since there is only one master node available. With such a model, upgrades do not require a cluster or a service downtime. Instead, only the affected node is involved, and the service continues to be available to end users.

- *Shared storage:* Master nodes require private disk storage to maintain the configuration files for the services they provide to all cluster nodes. Application data on the other hand is stored on shared storage. If a traffic node fails, the application data is still available to the surviving traffic nodes. Therefore, all application data is available exclusively on redundant highly available storage and is not dependent on the serving nodes.
- *Master node failure detection:* The heartbeat mechanism running on master nodes ensures that the standby master node detects the failure of the active master node within a delay of 200 ms. Section 3.18 describes how the architecture handles the failure of a master node.
- *Number of nodes in the high availability tier:* The current implementation of the HA tier supports two master nodes. In this model, the standby node is available to allow a quick transition when the active node fails, or for load-sharing purposes. We can add more master nodes to provide additional protection against multiple failures. The HA tier can scale from two to N master nodes. Although the addition of nodes would provide additional protection and higher capacity, it introduces significant complexity to the implementation of the system software. The current prototype HAS architecture supports the 1+1 redundancy model in both active/standby and active/active modes. Section 3.7 describes the supported redundancy models.
- *Cluster Virtual IP Interface (CVIP):* The cluster virtual IP interface is a transparent layer that presents the cluster as a single entity to the outside world, whether it is users to the cluster, or other systems. Users of the system are not aware that the web system consists of multiple nodes, and they do not know where the applications run. Section 3.20 describes the CVIP interface.
- *Connection synchronization:* Ongoing connections are synchronized. If one of the master nodes fails, the standby node service ongoing connections without a loss in service.

3.5.2 Characteristics of the SSA Tier

The SSA tier consists of multiple independent traffic nodes, each running a copy of the Apache web server software (version 2.0.35), and service incoming requests. Theoretically, there are no limitations on the number of traffic nodes in this tier: the number of nodes can be N, where $N \geq 2$ to insure that traffic nodes do not constitute a SPOF. The HAS architecture prototype consisted of 2 master nodes and 16 traffic nodes. Redundancy in the SSA tier is at the node level. Requests coming from the HA tier are assigned to traffic nodes based on the traffic distribution algorithm (Section 3.22). Traffic nodes reply to requests directly to the web clients eliminating possible bottleneck at master nodes.

The characteristics of the SSA tier are the following:

- *Node availability:* Single node availability is less important in this tier, since there are N traffic nodes available to serve requests. If one traffic node becomes unavailable, the master node removes the failed traffic node from its list of available traffic nodes, and directs traffic to available traffic nodes. Section 3.24.8 presents the case scenario of a failing traffic node.
- *Traffic node failure detection:* In the same way the heartbeat mechanism checks the health of the master nodes in the HA tier, the HAS architecture supports mechanisms to verify that a traffic node is up and running, and that the web software application is providing service to web clients. We use two methods simultaneously to ensure that traffic nodes are healthy and that the web server application is up and running. The first check is a continuous communication between the traffic manager running on the master node and the traffic client running of the traffic node (Section 3.22). The second check is an application check that ensures that the web server application is up and running (Section 3.19).
- *Support for diskless nodes:* The SSA tier supports diskless traffic nodes. Traffic nodes are not required to have local disks and rely on image servers for booting and downloading a kernel and disk image into their memory.
- *Seamless upgrades without service interruption:* With the HAS architecture, it is possible to upgrade the operating system and the application software without disturbing the service availability. We provide the mechanisms to upgrade the kernel and application automatically through an image server, by rebooting the nodes. Upon reboot, the traffic node downloads an updated kernel and the new version of the ramdisk (if the node is diskless) or a new image disk (if the node has disk) from the image server. Section 3.24.5 presents this upgrade scenario with the sequence diagram.

- *Hosting application servers:* Traffic nodes run application servers that can be stateful or stateless. If an application requires state information, then the application saves the state information on the shared storage and makes it available to all cluster nodes.

3.5.3 Characteristics of the Storage Tier

Data oriented applications require that storage and access to storage is available at all times. To ensure reliability and high availability, the architecture does not allow application data to be stored on local node storage devices. Application servers maintain their persistent data on the shared storage nodes located in the storage tier, or on shared storage managed by master nodes through a redundant network file server (Section 3.15.2). Since storage techniques are outside the scope of the thesis, we do not provide a full coverage of this area. However, the architecture is flexible to support multiple ways of providing storage through specialized storage nodes, software RAID and distributed file systems. The supported storage models include:

- *Specialized storage nodes* use techniques such as storage area networks or network-attached storage to provide highly available shared storage over a network to a large network of users.
- *RAID techniques* consist of established techniques to achieve data reliability through redundant copies of the data. Since we are using COTS software, then software RAID is our choice to provide redundant and reliable data storage. In addition, software RAID techniques do not require specialized hardware; therefore, we can use it with master nodes when master nodes are providing shared storage through the NFS.
- *Distributed file systems* allow us to combine all the disk storage on multiple cluster nodes under one virtual file system.

3.6 Availability and Eliminating Single Points of Failures

The HAS architecture aims to increase service availability by offering capabilities to detect errors and faults, and provide correction procedures to recover, when it is possible. The goal with the HAS architecture is to increase the MTBF by improving the quality of the software modules and by using redundancy to eliminate single points of failures, and to decrease MTTR by streamlining and accelerating fail-over, responding quickly to fault conditions, and making faults more granular in time and scope.

The topology of the HAS architecture enables failure tolerance because of the various built-in redundancies within all layers of the HAS architecture.

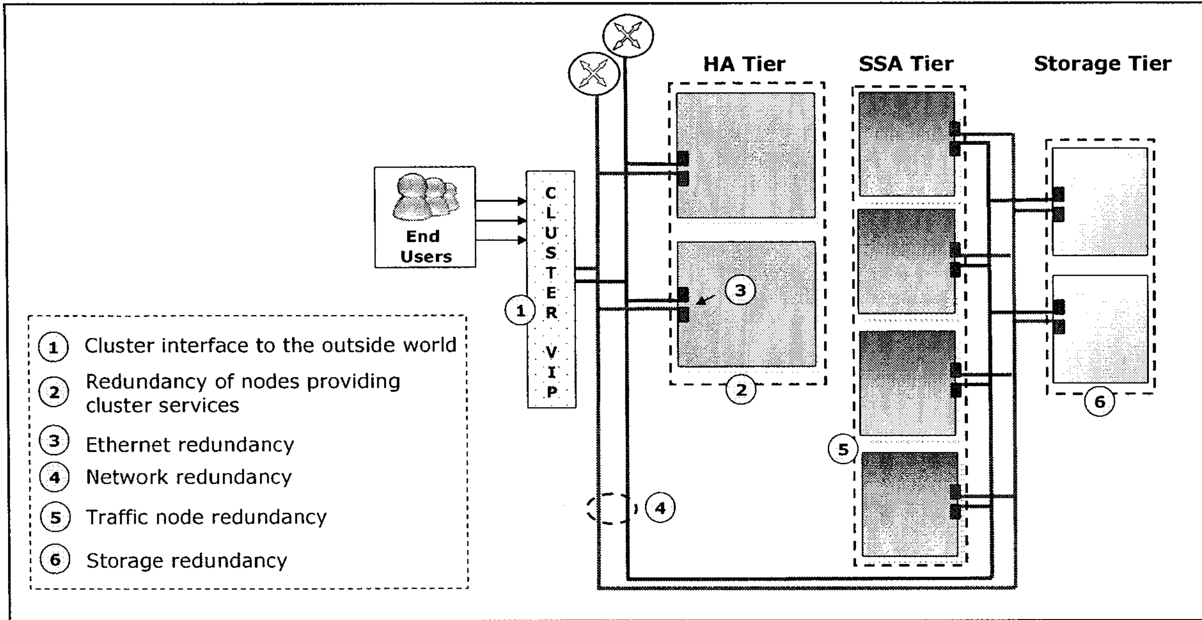


Figure 30: Built-in redundancy at different layers of the HAS architecture

Figure 30 illustrates the supported redundancy at the different layers of the HAS architecture. The cluster virtual IP interface (1) provides a transparent layer that hides the internal of the cluster. We can add or remove master nodes from the cluster without interruptions to the services (2). Each cluster node has two connections to the network (3) ensuring network connectivity (4). Many factors contribute towards achieving network and connection availability such as the availability of redundant routers and switches, redundant network connections and redundant Ethernet cards. We contributed an Ethernet redundancy mechanism to ensure high availably for network connections. As for traffic nodes (5), redundancy is at the node level, allowing us to add and remove traffic nodes transparently and without service interruption. We can guarantee service availability by providing multiple instances of the application running on multiple redundant traffic nodes. The HAS architecture supports storage redundancy (6) through a customized HA implementation of the NFS server; alternatively, we can also use redundant specialized storage nodes.

The following sub-sections discuss eliminating SPOF at each of the HAS architecture layers.

3.6.1 Eliminating Master Nodes as a Single Point of Failure

The HA tier of the HAS architecture consists of two master nodes that follow the 1+1 redundancy model. In the 1+1 active/standby redundancy model, if the active master node is unavailable and does not respond to the heartbeat messages, the standby master node declares it as dead and assumes the active state. Section 3.24.7 discusses this scenario. If the master nodes follow the 1+1 active/active redundancy model, the failures in one master node are transparent to end users, and do not affect the service availability.

3.6.2 Eliminating Applications as a Single Point of Failure

The primary goal of the HAS architecture is to provide a highly available environment for web server applications. Each traffic node runs a copy of the web server. These applications represent a SPOF, since in the event that the application crashes, the service on that traffic node becomes unavailable. To ensure the availability of these applications, the HAS architecture prototype supports two mechanisms. The first mechanism is a health check between the traffic manager and the traffic client. Section 3.22.5 discusses this health check mechanism, and Section 3.24.10 presents the sequence diagram of this scenario. The second mechanism is an application health check to ensure that the application is up and running. If the application is not available, the health check mechanism notifies the traffic manager that removes the traffic node from its list of available node, and as a result, the traffic nodes stops receiving incoming requests. Sections 3.24.8 and 3.24.11 present the sequence diagrams of this scenario.

3.6.3 Eliminating Network Adapters as a Single Point of Failure

The Ethernet redundancy daemon is a system software contribution that handles network adapter failures. Figure 31 illustrates the Ethernet adapter swapping process. When a network adapter (Ethernet card) fails, the Ethernet redundancy daemon swaps the roles of the active and standby adapters on that node. The failure of the active adapter is transparent with a delay of less than 350 ms while the system switches to the standby Ethernet adapter.

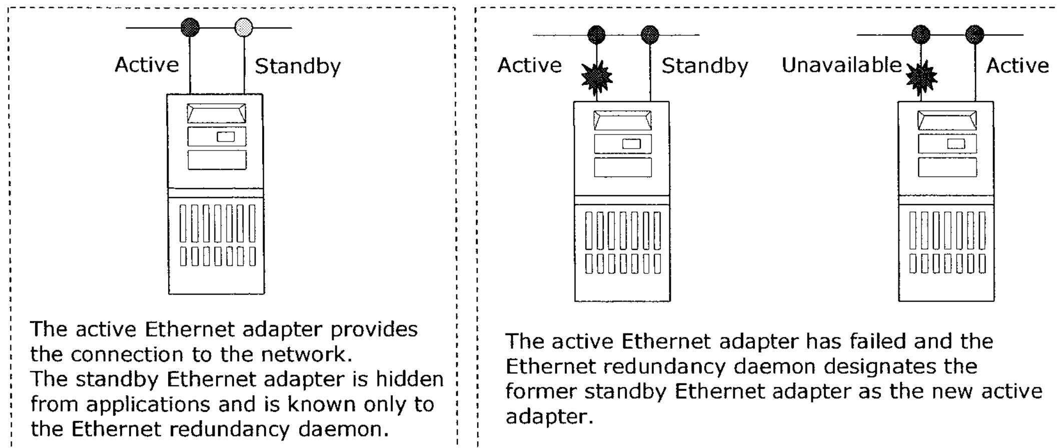


Figure 31: The process of the network adapter swap

3.6.4 Eliminating Storage as a Single Point of Failure

Shared storage is a possible SPOF in the cluster if it relies on a single storage server. Section 3.15 discusses the physical model description which covers eliminating storage as a SPOF using both a custom implementation of a highly available network file system (Section 3.15.2), and redundant specialized storage nodes (Section 3.15.4).

3.7 Overview of Redundancy Models

The following sub-sections examine the various redundancy models: the 1+1 two nodes redundancy model, the N+M and N-way redundancy models, and the *no redundancy* model.

3.7.1 The 1+1 Redundancy Model

There are two types of the 1+1 redundancy model: the active/standby, which is also called the asymmetric model, and the active/active or the symmetric redundancy model [48]. With the 1+1 active/standby redundancy model, one cluster node is active performing work, while the other node is a dedicated standby, ready to take over should the active master node fails. In the 1+1 active/active redundancy model, both nodes are active and doing work. In the event that either node should fail, the survivor node steps in to service the load of the failed node until the first node is back to service.

3.7.2 The N+M Redundancy Model

In the N+M redundancy model, the cluster tier support N active nodes and M standby nodes. If an active node fails, a standby node takes over the active role. If M=1, then the model would be the N+1 redundancy model, where the "+1" node is standby, ready to be active when one of the active nodes fail. The N+1 redundancy model can be applied to services that do not require a high level of availability because an N+1 cluster cannot provide a standby node to all active nodes. N+1 cannot perform hot standby because the standby node never knows which active nodes will fail. One advantage of the N+M over the N+1 is that in the N+M (with $M \geq 2$) provide higher availability should more than one node fails, while not affecting the performance and throughput of the cluster.

3.7.3 The N-way Redundancy Model

In the N-way redundancy model, all N nodes are active. Redundancy is at the node level.

3.7.4 The "No Redundancy" Model

The "no redundancy" model provides no redundancy as its name implies and it is used when the failure of a component does not cause a severe impact on the overall system. Following this model, an active component does not have a standby ready to take over if the active fails. We list this model for completion purposes.

3.8 HA Tier Redundancy Models

The HA tier supports three redundancy models: the 1+1 active/standby, 1+1 active/active and the N-way redundancy model. The current prototype of the HA tier supports the 1+1 redundancy model in both configurations: active/standby and active/active. Support for the N+M and for the N-way redundancy models is a future work item.

3.8.1 The HA Tier Active/Standby Redundancy Model

Figure 32 illustrates the 1+1 active/standby redundancy model supported by the HA tier, which consists of two master nodes, one is active and the second is standby. The active master node hosts active processes and the standby nodes host standby processes that are ready to take over when primary node fails. The active/standby model enables fast recovery upon the failure of the active node.

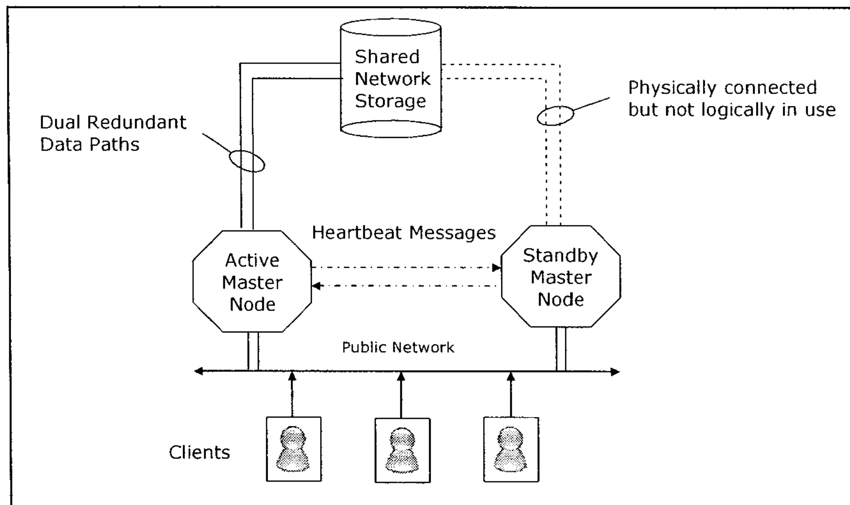


Figure 32: The 1+1 active/standby redundancy model

Figure 33 illustrates the 1+1 active/standby pair after the failover has completed. The active/standby redundancy model supports connection synchronization between the two master nodes. Section 3.21 discusses connection synchronization.

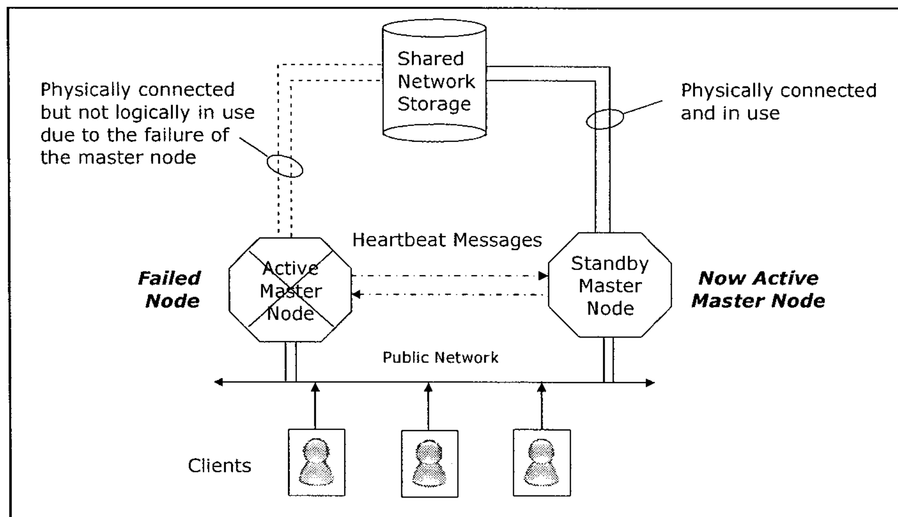


Figure 33: Illustration of the failure of the active node

The HA tier can transition from the 1+1 active/standby to the 1+1 active/active redundancy model through the redundancy configuration manager, which is responsible for switching from one redundancy model to another. The 1+1 active/standby redundancy model provides high availability; however, it

requires a master node to sit idle waiting for the active node to fail so it can take over. The active/standby model leads to a waste of resources and limits the capacity of the HA tier.

The 1+1 active/active redundancy model, discussed in the following section, addresses this problem by allowing the two master nodes to be active and to serve incoming requests for the same virtual service.

3.8.2 The HA Tier Active/Active Redundancy Model

The active/standby redundancy model offers one way of providing high availability. However, we would like to take advantage of the standby node and not have it sitting idle. By moving to the active/active redundancy model, both nodes in this tier are active and resulting in an increased cluster throughput. The active/active model allows two nodes to load balance the incoming connections for the same virtual service at the same time. It provides a solution where the master nodes are servicing incoming traffic and distributing it to the traffic nodes; as a result, we increase the capacity of the whole cluster, while still maintaining the high availability of master nodes.

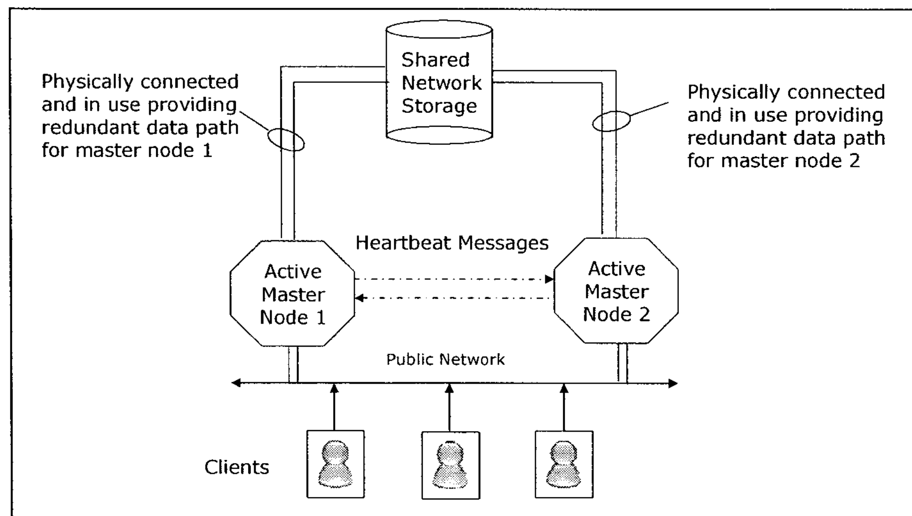


Figure 34: The 1+1 active/active redundancy model

Figure 34 illustrates the 1+1 active/active redundancy model, which supports load sharing between the two master nodes in addition to concurrent access to the shared storage. If the HAS cluster experiences additional traffic, we can extend the number of nodes in this tier and support the N-way redundancy model, where all nodes are active and providing service. However, increasing the number of nodes beyond two nodes increases the complexity of the implementation.

3.8.2.1 The Role of the Saru Module

When the HA tier is in the active/active redundancy model, each master node has the same hardware address (MAC) and IP address and the challenge is how to distribute incoming connections between the two active master nodes. For that purpose, we have used the saru module [31], an existing open source software, to run in coordination with heartbeat on each of the master nodes and be responsible for dividing the incoming connections between the two master nodes. The heartbeat daemon provides a mechanism to determine which master node is available and the *saru* module uses this information to divide the space of all possible incoming connections between the two available master nodes. Incoming requests are forwarded to master nodes according to a hashing function based on source IP address/port. The function is deterministic; a specific source IP address/port always reaches the same IP server.

When the master nodes boot, the saru module starts and elects a master node to do the allocations. The elected master node divides blocks of source or destination ports or addresses between the two active master nodes. The notion of a master and a slave node is only used inside the saru module. Both master nodes are active and continue to service incoming requests distribute it to the traffic nodes.

3.9 SSA Tier Redundancy Models

The SSA tier supports the N+M and the N-way redundancy models. In the N+M model, N is the number of active traffic nodes hosting the active web server application, and M is the number of standby traffic nodes. When M=0, it is the N-way redundancy model where all traffic nodes are active. Following the N-way redundancy model, traffic nodes operate without standby nodes. Upon the failure of an active traffic node, the traffic manager running on the master node removes the failed traffic node from its list of available traffic nodes (Section 3.24.8).

3.10 Storage Tier Redundancy Models

Although storage is outside the scope of our work, the redundancy models of the storage tier depend on the physical storage model described in Section 3.15.

3.11 Redundancy Model Choices

Figure 35 identifies the various redundancy models supported for master, traffic and storage nodes.

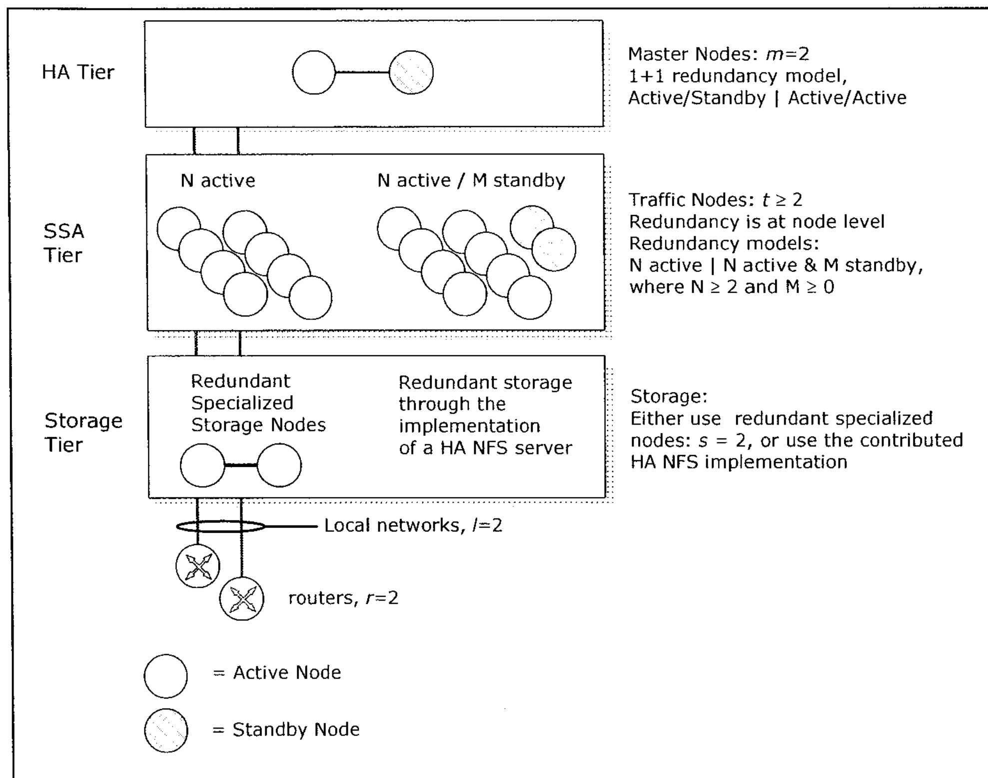


Figure 35: the redundancy models at the physical level of the HAS architecture

Master nodes follow the 1+1 redundancy model. The HA tier hosts two master nodes that interact with each other following the active/standby model or the active/active (load sharing) model. Traffic nodes follow one of two redundancy models: N+M (N active and M standby) or N-way (all nodes are active). In the N+M active/standby redundancy model, N is the number of active traffic nodes available to service requests. We need at least two active traffic nodes, $N \geq 2$. M is the number of standby traffic nodes, available to replace an active traffic node as soon as it becomes unavailable. The N-way redundancy model is the N+M redundancy model with $M = 0$. In the N-way redundancy model, all traffic nodes are in the active mode and servicing requests with no standby traffic nodes. When a traffic node becomes unavailable, the traffic manager stops sending traffic to the unavailable node. Instead, traffic will be forwarded to the remaining available traffic nodes. However, when standby nodes are available, the throughput of the cluster does not suffer from the loss of a traffic node since the standby node takes over the unavailable traffic node.

As for the storage tier, the redundancy model depends on various possibilities ranging from hosting data on the master nodes to having separate and redundant nodes that are responsible for providing storage to

the cluster. The redundancy configuration manager is responsible for switching from one redundancy model to another. For the purpose of the work, we did not implement the redundancy configuration manager (Section 5.2.4). Rather, we relied on re-starting the cluster nodes with modified configuration files when we wanted to experiment with a different redundancy model.

Table 8 provides a summary of the redundancy models per each tier in the HAS architecture. The HAS architecture supports all redundancy models across all the tiers.

	1+1 Active/Active	*1+1 Active/Standby*	*N+M*	*N-Way*	*No Redundancy*
HA Tier	X	X	X	X (N+M, with M=0)	X (one master node)
SSA Tier	X	X	X	X	X (one traffic node)
Storage Tier	X (2 NFS servers)	X	X	X	X (one storage node)

Table 8: Redundancy models per each tier of the HAS architecture

Table 9 illustrates the implemented redundancy models for the HAS architecture proof-of-concept. At the HA tier, both the 1+1 active/standby and the 1+1 active/active redundancy models are supported. At the SSA tier, the HAS architecture supports the N-way redundancy model. The storage tier supports the 1+1 active/active redundancy model.

	1+1 Active/Active	**1+1 Active/Standby**	**N+M**	**N-Way**	**No Redundancy**
HA Tier	X	X			X (one master node)
SSA Tier				X	X (one traffic node)
Storage Tier	X				X (one NFS server)

Table 9: Supported redundancy models per each tier in the HAS architecture prototype

3.12 The States of a HAS Cluster Node

A state transition diagram illustrates how a cluster node transitions from one defined state to another, because of certain events. We represent the states of the nodes as circles, and label the transitions between the states with the events or failures that changed the state of the node from one state to another. A node

starts in an initial state, represented by the closed circle; and can end up in a final state, represented by the bordered circle. A cluster node can be in one of four HA states: active, standby, in-transition, and out-of-cluster.

We identify two types of error conditions: recoverable and unrecoverable errors. Recoverable errors such as the failure of an Ethernet adapter do not qualify as error conditions that require the node to change state because such a failure is recoverable. However, other failures such as a kernel crash are unrecoverable and as a result require a change of state.

Figure 36 presents the HA states of a node, discarding the out-of-cluster state for simplicity purposes.

We assume that the initial state of a node is active (1). When in the active state, the node is ready to service incoming requests or provide cluster services. When the active node is facing some hardware or software problem forcing it to stop service, it transitions (2)(3) to the out-of-cluster state. In this state, the node is not a member of the cluster; it does not receive traffic (traffic node) or provide cluster services to other nodes (master node). When the problem is fixed, the node transitions back to the active state (4)(5) and becomes active. A node is in-transition when it is changing state. If the node is changing to an active state (5), after the completion of the transition, the node starts receiving incoming requests.

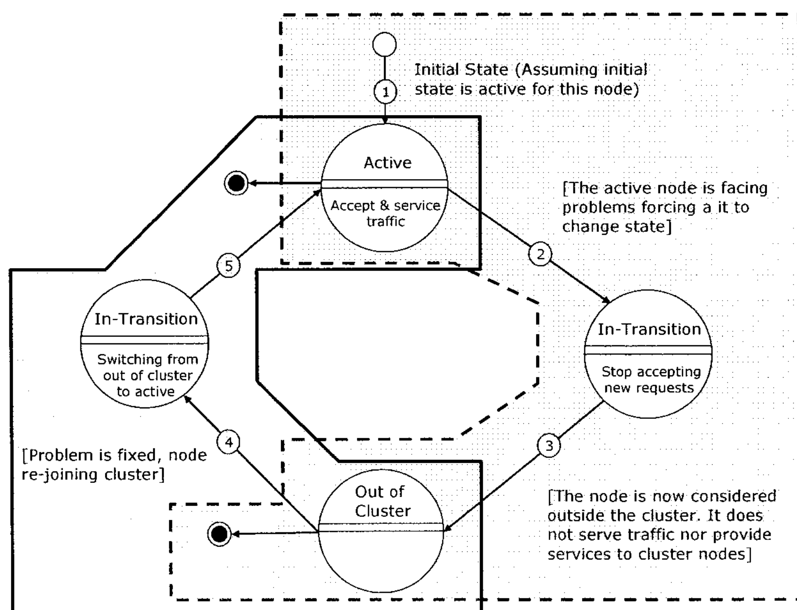


Figure 36: The state diagram of the state of a HAS cluster node

Figure 37 represents the state diagram after we expand it to include the standby state, in which the node is not currently providing service but prepared to take over the active state. This scenario is only applicable

to nodes in the HA tier, which supports the 1+1 active/standby redundancy model. When the node is in the active, in-transition or standby state, and it encounters software or hardware problems, it becomes unstable and it will not be member of the HAS cluster. Its state becomes out-of cluster and it is not anymore available to service traffic. If the transition is from active to standby, the node stops receiving new requests and providing services, but keeps providing service to ongoing requests until their termination, when possible; otherwise, ongoing requests are terminated. The system software that manages the transition of states are the traffic manager and the heartbeat daemon running on the master nodes, and the traffic client and the LDirectord running on the traffic nodes.

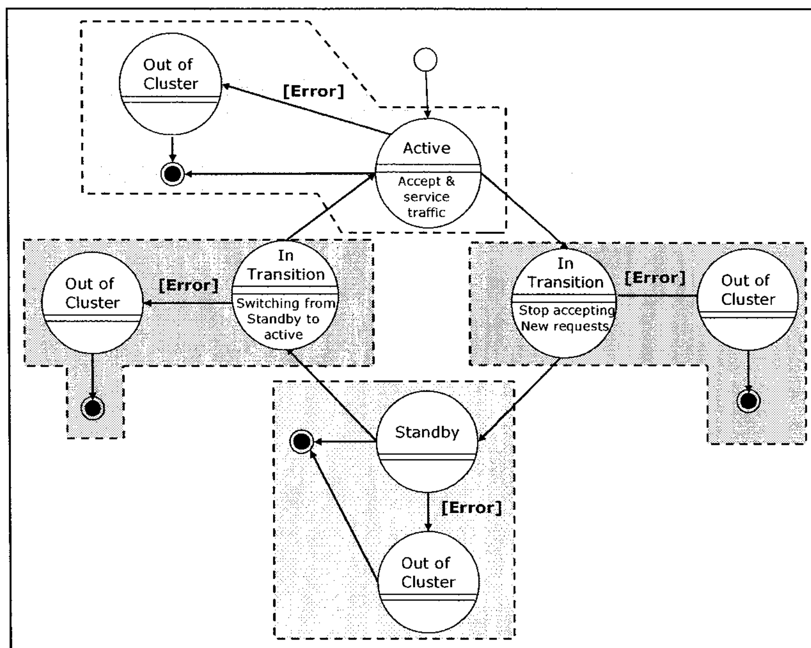


Figure 37: The state diagram including the standby state

3.13 Example Deployment of a HAS Cluster

Figure 38 illustrates an example prototype of the HAS architecture using a distributed file system to provide shared storage among all cluster nodes. In this example, two master nodes (A and B) provide storage using the contributed implementation of the highly available NFS (Section 3.15.2). The HA tier consists of two master nodes in the 1+1 active/standby redundancy model. Node A is the active node and Node B is the standby node. The SSA tier consists of four traffic nodes each with its own local storage. The SSA tier follows the N-way redundancy model, where all traffic nodes are active. There are two redundant local area networks, LAN 1 and LAN 2, connected to two redundant routers. The minimal

prototype of the HAS architecture requires two master nodes and two traffic nodes. In the event that the incoming web traffic increases, we add more traffic nodes into the SSA tier. If we scale the number of traffic nodes, no other support nodes are required and no changes are required for either the master nodes or the current traffic nodes.

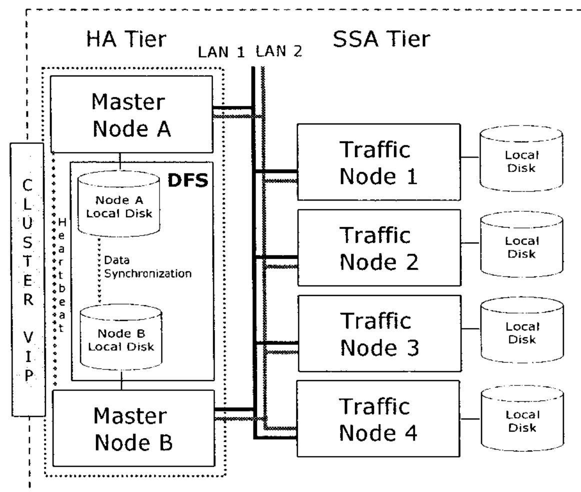


Figure 38: A HAS cluster using the HA NFS implementation

3.14 The Physical View of the HAS Architecture

The physical model specifies the architecture topology in terms of hardware architecture and organization of physical resources (processors, network and storage elements). It also describes the hardware layers and takes into account the system's non-functional requirements such as system connection availability, reliability and fault-tolerance, performance (in terms of throughput) and scalability.

The HAS architecture consists of a collection of inter-connected nodes presented to users as a single, unified computing resource. Nodes are independent compute entities that physically reside on a common network. The software executes on a network of computers. We map the various elements identified in the logical, process, and development views onto the various cluster nodes that constitute the HAS architecture. The mapping of the software modules to the nodes is highly flexible and has minimal impact on the source code itself. We can deploy different physical configurations depending on deployment and testing purposes.

Figure 39 illustrates the physical view of the architecture, which consists of the following resources: master nodes, traffic nodes, redundant storage nodes, redundant LANs, one router per LAN, and two redundant image servers.

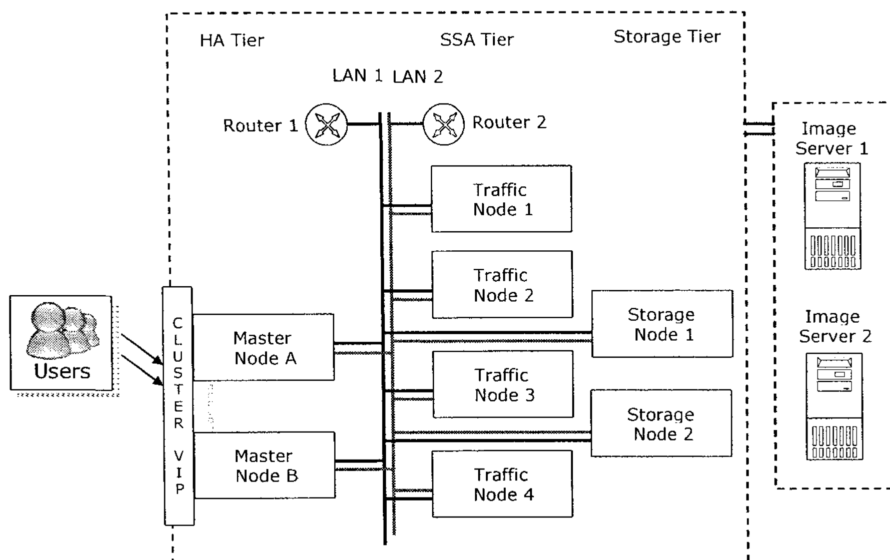


Figure 39: The physical view of the HAS architecture

The HA tier consists of at least two redundant master nodes that accept incoming traffic from the Internet through a virtual network interface that presents the cluster as a single entity and provides cluster infrastructure services to all HAS cluster nodes. The HA tier does not tolerate service downtime because if the master node (acting as dispatcher) goes down, the traffic nodes are unable to receive traffic to service web clients. The HA tier controls the activity in the SSA tier and therefore needs to be able to determine the health of traffic nodes and their load.

The SSA tier consists of a farm of independent and possibly diskless traffic nodes. The number of nodes in this tier scales from two to N nodes. If a node in this tier fails, the dispatcher stops forwarding traffic to the failed node, and forwards incoming traffic to the remaining available traffic nodes.

The storage tier consists of at least two redundant specialized storage nodes.

The HAS architecture requires the availability of two routers (or switches) to provide a highly available and reliable communication path.

An image server is a machine that holds the operating system and ramdisk images of the cluster nodes. This machine, two for redundancy purposes, is responsible for propagating the images over the network to the cluster nodes every time there is an upgrade or a new node joining the cluster. Master nodes can provide the functionalities of the image server; however, for large deployments this might slow down the performance of master nodes. Image servers are external and optional components to the architecture.

We divide the cluster components into the following functional units: master nodes, traffic nodes, storage nodes, local networks, external networks, network paths, and routers.

The m master nodes form the HA tier in a HAS architecture and implement the 1+1 redundancy model. The number of master nodes is $m = 2$. These nodes can be in the active/standby or active/active mode. When $m \geq 2$, then the redundancy model is the N+M model; however, we do not implemented this redundancy model in the HAS prototype. The t traffic nodes are located in the SSA tier, where $t \geq 2$. If $t = 1$, then there is a single traffic node that constitutes a SPOF. Let s indicate the number of storage nodes. If $s = 0$, then the cluster does not include specialized storage nodes; instead master nodes in the HA tier provide shared storage using a highly available distributed file system. The HA file system uses the disk space available on the master nodes to host application data. When $s \geq 2$, it indicates that at least two specialized nodes are providing storage. When $s \geq 2$, we introduce the notion of d shared disks, where $d \geq 2 \times s$. The d shared disks are the total number of shared disks in the cluster. The l local networks provide connectivity between cluster nodes. For redundancy purposes, $l \geq 2$ to provide redundant network paths. However, this is dependent on two parameters: the number of routers r available ($r \geq 2$, one router per network path) and the number of network interfaces *eth* available on each node (one *eth* interface per network path). The HAS architecture requires a minimum of two Ethernet cards per cluster node; therefore $eth \geq 2$. The cluster can be connected to outside networks, identified as e, where $e \geq 1$ to recognize that the cluster is connected to at least one external network.

3.15 The Physical Storage Model of the HAS Architecture

The storage model of the HAS architecture aims to meet two essential requirements. The first requirement is for the storage to be highly available, ensuring that data is always available to cluster users and applications. This requirement enables tolerance of single storage unit failure as mirrored data is hosted on at least two physical units; it also allows node failure tolerance, as mirrored data can be accessed form different nodes. The second requirement is to have high throughput and minimum access delay. Based on these requirements, we propose three possible physical storage access models for the HAS architecture: using the HA implementation of the NFS, using specialized shared storage, or using local node storage. The following sub-sections explore these storage access methods and discuss their architectures and redundancy models.

3.15.1 Without Shared Storage

Figure 40 illustrates the *no-shared* storage model. This model has limited advantages but we list it for completeness. Following this model, cluster nodes use their local storage to store configuration and application data. While the no-shared model offers low cost, it comes at the expense of not being able to use diskless nodes. In addition, application data is not replicated since state information is saved on each node where the transaction took place. If a node becomes unavailable, the data that resides on this node becomes unavailable too. As a result, the no-shared storage model is limited and poses restrictions on cluster nodes and their applications.

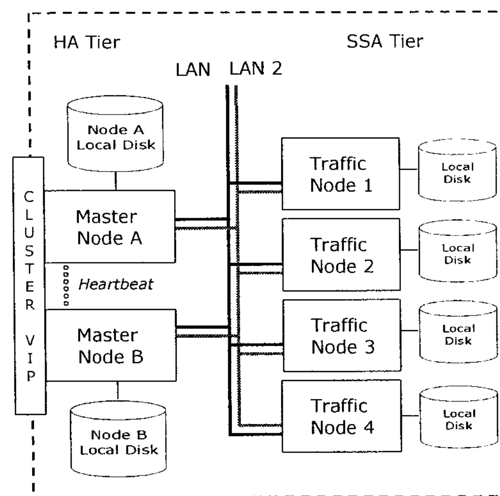


Figure 40: The *no-shared storage* model

However, it is worthy to mention that other research projects (Section 2.15.5) have adopted this model as their preferred way of handling data and dividing it across multiple traffic nodes. Following their architectures, a traffic node receives a connection only if it has the data stored locally.

3.15.2 Shared Storage with Distributed File Systems

Distributed file systems enable file system access from a client through the network. This model enables cluster node location transparency as each cluster node has direct access to the file system data and it is suitable for applications with high storage capacity and access performance requirements.

Figure 41 illustrates the physical model of the HAS architecture using shared storage. In this example, cluster nodes rely on the networked storage available through the master nodes. Master nodes run the modified implementation of the NFS server in which two NFS servers provide redundant shared storage

to cluster nodes. In the event of a failure of one of the NFS servers, the other NFS server continues to provide access to data. We contributed this mechanism with a modified implementation of the mount program that allows us to mount two NFS servers into the same location, where the client data resides.

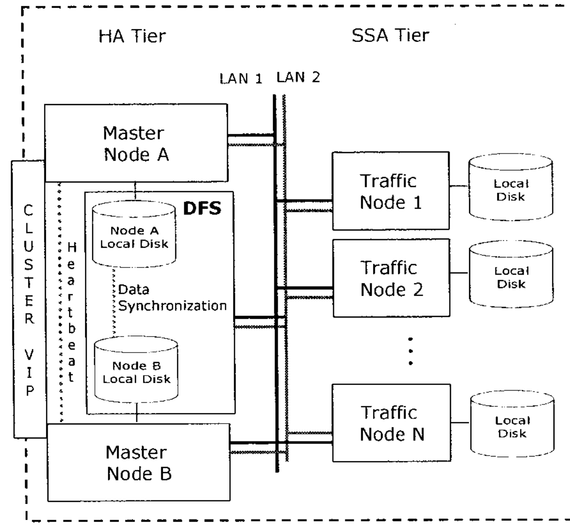


Figure 41: The HAS storage model using a distributed file system

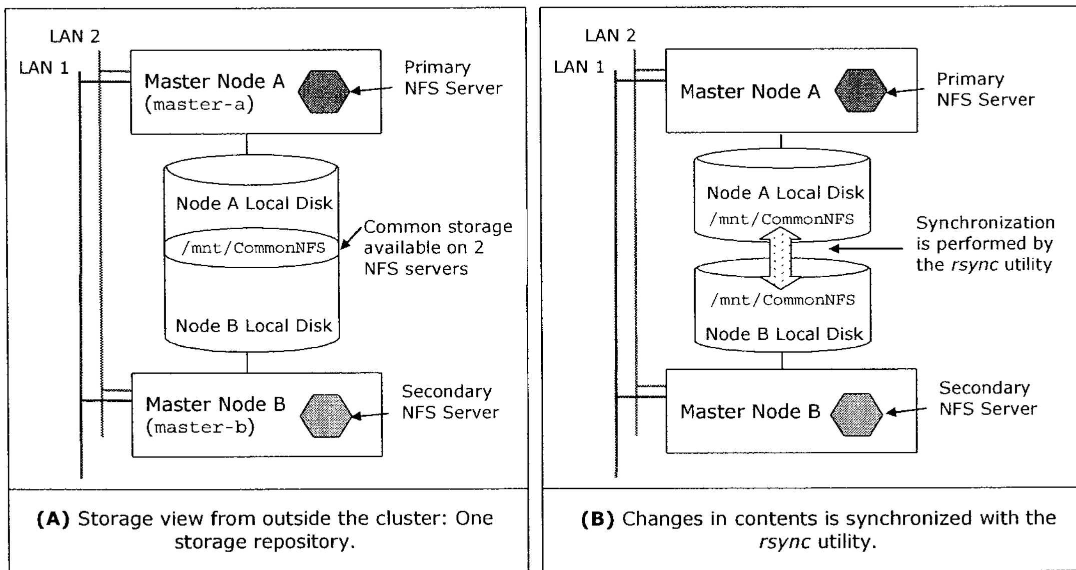


Figure 42: The NFS server redundancy mechanism

Figure 42 illustrates how the HAS architecture achieves NFS server redundancy. In Figure 42-A, `master-a`, is the name of the Master Node A server, and `master-b` is the name of the Master Node B server. Both

master nodes are running the modified HA version of the network file system server. Using the modified *mount* program, we mount a common storage repository on both master nodes:

```
% mount -t nfs master-a,master-b:/mnt/CommonNFS
```

When the rsync utility detects a change in the contents, Figure 42-B, it performs the synchronization to ensure that both repositories are identical. If the NFS server on `master-a` becomes unavailable, data requests to `/mnt/CommonNFS` will not be disturbed because the secondary NFS server on `master-b` is still running and hosting the `/mnt/CommonNFS` network file system.

3.15.2.1 Contribution of New HA Extensions for the NFS Server

It was essential to have this functionality in place to enable highly available shared storage that is accessible to all cluster nodes without a 5SPOF. As a result, in the event that the primary NFS server running on the active master node fails, the secondary NFS server running on the standby node continues to provide storage access without service discontinuity and transparently to end users. The *rsync* utility provides the synchronization between the two NFS servers.

Linux Kernel - Modified Source Files	Description of Changes
`/usr/src/linux/fs/nfs/inode.c`	Added support for NFS redundancy
`/usr/src/linux/net/sunrpc/sched.c`	Raise the timeout flag when there is one
`/usr/src/linux/net/sunrpc/clnt.c`	Raise the timeout flag when there is one

Table 10: The changes made to the Linux kernel to support NFS redundancy

Table 10 lists the Linux kernel files modified to support the NFS redundancy. The implementation of the HA NFS server requires upgrading to the latest stable Linux Kernel release, version 2.6. This contribution is presented and discussed in [23].

3.15.2.2 Contribution of a New Mount Program to Support Mounting Dual NFS Servers

Furthermore, the HA NFS requires a new implementation of the mount program to support mounting multi-host NFS servers, instead mounting a single file server. In the new mount program, the addresses of the two redundancy NFS servers are passed as parameters to the new mount program, and then to the kernel. The new command line for mounting two NFS server looks as follows:

```
% mount -t nfs server1,server2:/nfs_mnt_point
```

3.15.3 Synchronization of Shared Storage

The *rsync* utility is open source software that provides incremental file transfer between two sets of files across the network connection, using an efficient checksum-search algorithm [78]. It provides a method for bringing remote files into synch by sending just the differences in the files across the network link. The *rsync* utility can update whole directory trees and file systems, preserves symbolic links, hard links, file ownership, permissions, devices and times, and uses pipelining of file transfers to minimize latency costs. It uses *ssh* or *rsh* for communication, and can run in daemon mode, listening on a socket, which is used for public file distribution.

We used the *rsync* utility to synchronize data on both NFS servers running on the two master nodes in the HA tier of the HAS architecture.

3.15.3.1 Disk Replication Block Device

DRBD (Disk Replication Block Device) is an alternative to *rsync* to provide highly available storage. DRBD is system software that acts as a block device. It operates by mirroring a whole block device via the network and provides the synchronization between the data storage on both master nodes [73]. DRBD takes over the data, writes it to the local disk, and sends it to the other storage server. DRBD is a distributed replicated block device responsible for carrying the synchronization between the two independently running NFS servers on two separate nodes.

Figure 43 illustrates the data replication with DRDB. If the active node fails, the heartbeat switches the secondary device into primary state and starts the application there. If the failed node becomes available again, it becomes the new secondary node and it synchronizes its NFS content to the primary master node. This synchronization takes place as a background process and does not interrupt the service.

The DRBD utility provides intelligent resynchronization as it only resynchronizes those parts of the device that have changed, which results in less synchronization time. It grants read-write access only to one node at a time, which is sufficient for the usual fail-over HA cluster.

The drawback of the DRBD approach is that it does not work when we have two active nodes because of possibly multiple writes to the same block. If we have more than one node concurrently modifying distributed devices, we have an interesting problem to decide which part of the device is up-to-date on which node, and what blocks we need to resynchronize in which direction.

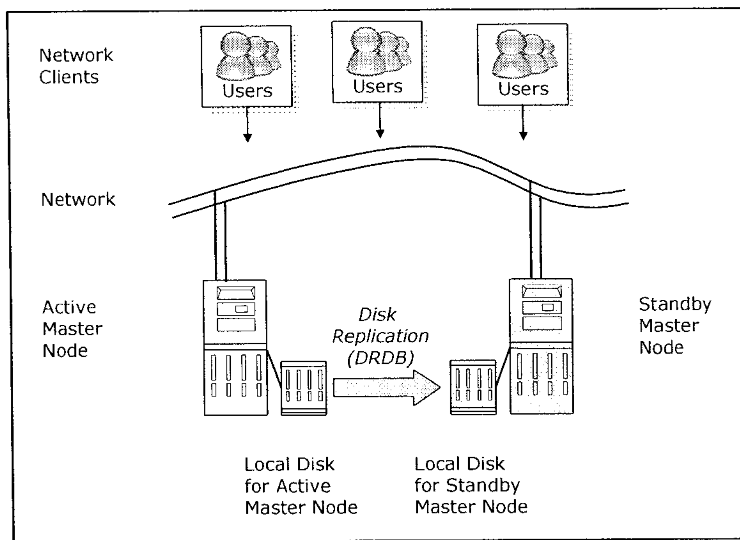


Figure 43: DRDB disk replication for two nodes in the 1+1 active/standby model

3.15.4 Storage with Specialized Storage Nodes

The HAS architecture allows the addition of specialized storage nodes that provide shared storage to the cluster nodes.

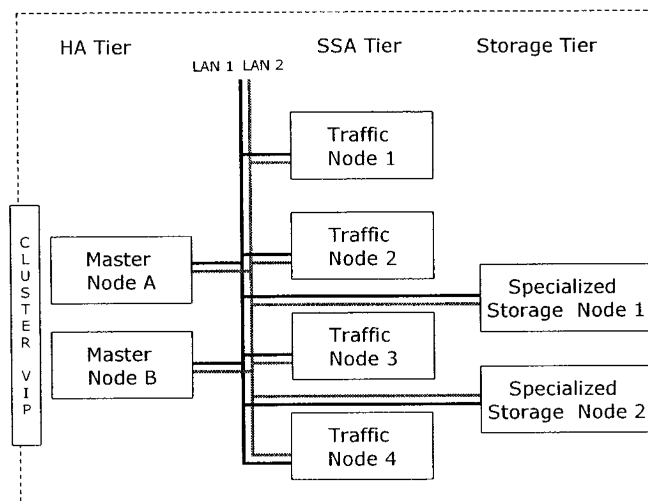


Figure 44: A HAS cluster with two specialized storage nodes

Figure 44 illustrates the physical model of the architecture using a storage area network provided by two redundant specialized storage nodes. Following this model, storage for application data is hosted on both

of the specialized storage nodes. The traffic nodes can use their private disk to maintain configuration and system files. We did not experiment providing storage using specialized storage nodes.

3.16 Types and Characteristics of the HAS Cluster Nodes

A cluster is a dynamic entity consisting of a number of nodes configured to be part of the cluster. Node can join or leave the cluster at anytime. Sections 3.24.3 and 3.24.7 describe the sequence diagrams for a node joining and leaving our HAS architecture prototype. A cluster node is the logical representation of a physical node. A node is an independent compute entity that is a member of a cluster and resides on a common network. It communicates with all the cluster nodes through two local networks, enabling tolerance against single local network failure. A cluster node can be a single or dual processor machine or an SMP machine. An independent copy of the operating system environment usually characterizes each cluster node. However, some cluster nodes share a single boot image from the central, shared disk storage unit or an image server.

There are three types of nodes in the HAS architecture: master, traffic and storage nodes. The following sub-sections present the characteristics of each node type and discuss its responsibilities and roles within the cluster.

3.16.1 Master Nodes

Master nodes reside in the HA tier of the HAS architecture. They have access to persistent storage, and depending on the specific deployment, they provide shared disk access to traffic nodes and host cluster configuration, management and application data. Master nodes have local disk storage, unlike traffic nodes that can be diskless.

Figure 45 illustrates the software and hardware stack of a master node in the HAS architecture. Master nodes provide an IP layer abstraction hiding all cluster nodes and provide transparency towards the end user. Master nodes have a direct connection to external networks. They do not run server applications; instead, they receive incoming traffic through the cluster virtual IP interface and distribute it to the traffic nodes (Section 3.22). Master nodes provide cluster-wide services for the traffic nodes such as DHCP server, IPv6 router advertisement, time synchronization, image server, and network file server. Master nodes run a redundant and synchronized copy of the DHCP server, a communications protocol that allows network administrators to manage centrally and automate the assignment of IP addresses. The configuration files of this service are available on the HA shared storage. The router advertisement

daemon (*radvd*) runs on the master nodes and sends router advertisement messages to the local Ethernet LANs periodically and when requested by a node sending a router solicitation message. These messages are specified by RFC 2461 [56], Neighbor Discovery for IP Version 6, and are required for IPv6 stateless autoconfiguration. The time synchronization server, running on the master nodes, is responsible for maintaining a synchronized system time. In addition, master nodes provide the functionalities of an image server.

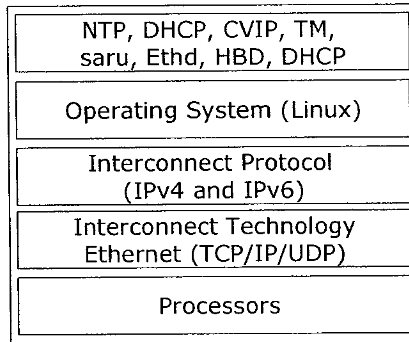


Figure 45: The master node stack

When the SSA tier consists of diskless traffic nodes, there is a need for an image server to provide operating system images, application images, and configuration files. The image server propagates this data to each node in the cluster and solves the problem of coordinating operating system and application patches by putting in place and enforcing policies that allow operating system and software installation and upgrade on multiple machines in a synchronized and coordinated fashion. Master nodes can optionally provide this service. In addition, master nodes provide shared storage via a modified, highly available version of the network file server. We also prototyped a modified mount program to allow master nodes to mount multiple servers over the same mounting point. Master nodes can optionally provide this service.

3.16.2 Traffic Nodes

Traffic nodes reside in SSA tier of the HAS architecture. Traffic nodes can be of two types: either with disk or diskless. Traffic nodes rely on cluster storage facilities for application data. Diskless traffic nodes rely on the image server to provide them with the needed operating system and application images to run and all necessary configuration data.

Figure 46 illustrates the software and hardware stack of a traffic node in the HAS architecture. Traffic nodes run the Apache web server application. They reply to incoming requests. Each traffic node runs a copy of the traffic client, LDirectord, and the Ethernet redundancy daemon.

Traffic nodes rely on cluster storage to access application data and configuration files, as well as for cluster services such as DHCP, FTP, NTP, and NFS services. Traffic nodes have the option to boot from the local disk (available for nodes with disks), the network (two networks for redundancy purposes), flash disk (for CompaqPCI architectures), from CDROM, DVDROM, or floppy. The default booting method is through the network. Traffic nodes also run the NTP client daemon, which continually keeps the system time in step with the master nodes. With the HAS architecture prototype, we experienced booting traffic nodes from the local disk, the network, and from flash disk which we mostly used for troubleshooting purposes.

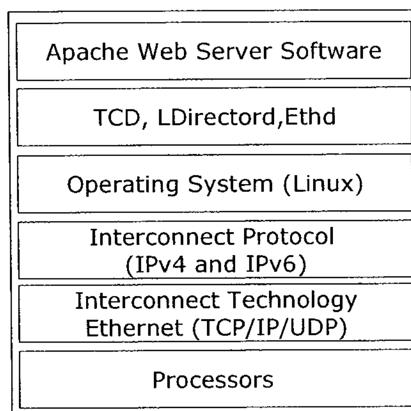


Figure 46: The traffic node stack

3.16.3 Storage Nodes

Cluster storage nodes provide storage that is accessible to all cluster nodes. Section 3.15 presents the physical storage model of the HAS architecture.

3.17 Local Network Access

Cluster nodes are interconnected using redundant dedicated links or local area networks (LAN). All nodes in the cluster are visible to each other. We assume, following the physical model, that all cluster nodes are one routing hop from each other. We achieve redundancy by using two local networks to interconnect all the nodes of the cluster. All nodes have two Ethernet adapters, with each connected to a separate LAN.

Figure 47 illustrates the local network access model where each cluster node connects to both LAN1 and LAN2; LAN1 and LAN2 are separate local area networks with their own subnet or domain.

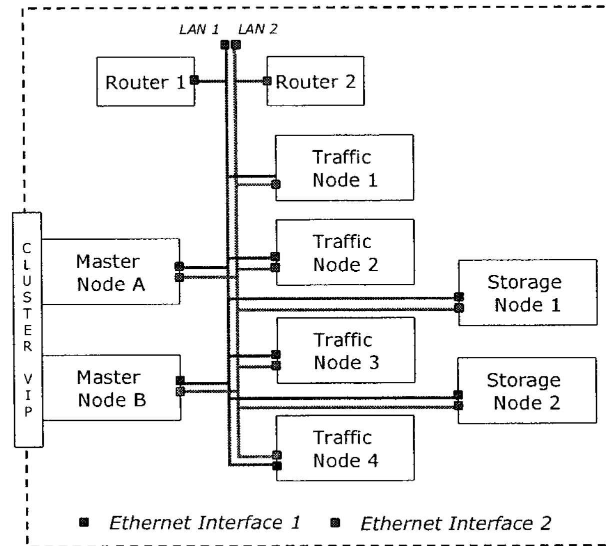


Figure 47: The redundant LAN connections within the HAS architecture

This connectivity model ensures high availability access to the network and prevents the network of being a SPOF. The HAS architecture supports both Internet Protocols IPv4 and IPv6. Supporting IPv4 does not imply additional implementation considerations. However, supporting IPv6 requires the need for a router advertisement daemon that is responsible for automatic configuration of IPv6 Ethernet interfaces. The router advertisement daemon also acts as an IPv6 router: sending router advertisement messages, specified by RFC 2461 [56] to a local Ethernet LAN periodically and when requested by a node sending a router solicitation message. These messages are required for IPv6 stateless autoconfiguration. As a result, in the event we need to reconfigure networking addressing for cluster nodes, this is achievable in a transparent fashion and without disturbance to the service provided to end users.

3.18 Master Nodes Heartbeat

It is necessary to detect when master nodes fail and when they become available again. Heartbeat is the system software that provides this functionality in the HAS architecture prototype. Heartbeat is an open source project that provides system software that runs on both master nodes over multiple network paths for redundancy purposes [65]. Its goal is to ensure that master nodes are *alive* through sending heartbeat packets to each other as defined in its configuration file. The heartbeat software is a low-level component that monitors the presence and health of master nodes in the HA tier of a HAS architecture by sending

heartbeat packets across the network to the other instances of heartbeat running on other master nodes as a sort of *keep-alive* message.

The *heartbeat* program takes the approach that the *keep-alive* messages, which it sends, are a specific case of the more general cluster communications service [49]. In this sense, it treats cluster membership as joining the communication channel, and leaving the cluster communication channel as leaving the cluster. Heartbeat itself acts similarly to a cluster-wide init daemon, making sure each of the services it manages is running at all times. When a master node stops receiving the heartbeat packets, it assumes that the other master node died; as a result, the services the primary master node was providing are failed over to the standby master node. Details on how the failover takes place are presented in [66].

Figure 48 illustrates the heartbeat topology. The HA tier consists of two master nodes following the 1+1 redundancy model. Heartbeat supports both configuration of the 1+1 redundancy model: the active/standby and active/active configurations. Each master node sends heartbeat and administrative messages to the other master node as broadcasts.

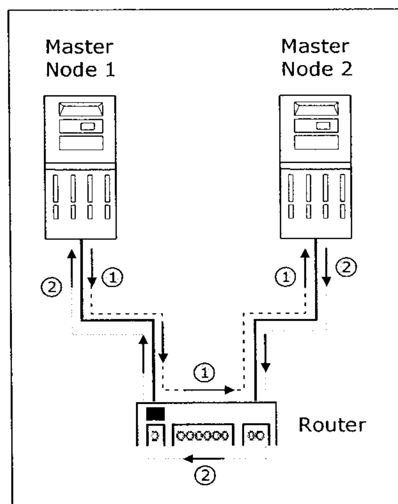


Figure 48: The topology of the heartbeat Ethernet broadcast

With heartbeat, master nodes are able to coordinate their role (active and standby) and track their availability. Heartbeat discussions are presented in [65] and [66].

3.18.1 Contributions to the Heartbeat Mechanism

The heartbeat mechanism is an existing software developed initially by the Linux-HA project. The contributions we made to this software are implementation related and not new ideas. The adaptations and

re-writing of parts of the original source code with the goal of optimizations, resulted in a decrease of the failure detection delay from 200 ms to 150 ms. Furthermore, we can expect additional improvements to the failure detection time down to 100 ms from 150 ms with further optimizations. As future work, we would like to investigate using the heartbeat mechanism with more than two nodes in the HA tier and supporting the N-way redundancy model.

3.19 Traffic Nodes Heartbeat using the LDirectord Module

In the same way, the heartbeat mechanism (Section 3.18) checks the availability of the master nodes in the HA tier, we need to provide a mechanism to verify that traffic nodes in the SSA tier are up and running, and providing service to web clients. We use two methods to ensure that traffic nodes are available and providing services. The first method is the keep-alive mechanism integrated in the traffic distribution scheme discussed in Section 3.22. Each traffic node reports its load to the traffic managers running on the master nodes every x seconds (x is a configuration parameter). This continuous communication ensures that the traffic managers are aware of all available traffic nodes. In the event that this communication is disrupted, after a pre-defined time out the traffic manager removes the traffic node from its list of available traffic nodes. This communication ensures that the traffic client is alive, and that it is reporting the load index of the traffic node to the traffic manager. Section 3.22 discusses this mechanism.

The second method for traffic nodes heartbeat is an application check whose goal is to ensure that the application server running on the traffic node is available and running. The application check relies on the Linux Director daemon (*LDirectord*) to monitor the health of the applications running on the traffic nodes. Each traffic node runs a copy of the *LDirectord* daemon.

The *LDirectord* daemon performs a *connect* check of the services on the traffic nodes by connecting to them and making a HTTP request to the communication port where the service is running. This check ensures that it can open a connection to the web server application. When the application check fails, the *LDirectord* connects to the traffic manager and sets the load index of that specific traffic node to zero. As a result, existing connections to the traffic node may continue, however no new requests are forwarded to this traffic node. Section 3.24.11 discusses this scenario. This method is also useful for gracefully taking a traffic node offline.

3.19.1 Sample LDirectord Configuration

The LDirectord module loads its configuration from the `ldirectord.cf` configuration file, which contains the configuration options. An example configuration file is presented below. It corresponds to a virtual web server available at address `192.68.69.30` on port `80`, with round robin distribution between the two nodes: `142.133.69.33` and `142.133.69.34`.

```
1       # Global Directives
2       Checktimeout    =       10
3       Checkinterval   =       2
4       Autoreload      =       no
5       Logfile         =       "local0"
6       Quiescent       =       yes
7
8       # Virtual Server for HTTP
9       Virtual =       192.68.69.30:80
10              Fallback        =       127.0.0.1:80
11              Real            =       142.133.69.33:80 masq
12              Real            =       142.133.69.34:80 masq
13              Service         =       http
14              Request         =       "index.html"
15              Receive         =       "Home Page"
16              Scheduler       =       rr
17              Protocol        =       tcp
18              Checktype       =       negotiate
```

Once the LDirectord module starts, the virtual server kernel table is populated. The capture below uses the `ipvsadm` command line to capture the output of the kernel. The `ipvsadm` command is used to set up, maintain or inspect the virtual server table in the Linux kernel. The listing illustrates the virtual server session, with the virtual address on port 80, and the two hosts providing this virtual service.

```
% ipvsadm -L -n
IP Virtual Server version 1.0.7 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port          Forward Weight ActiveConn InActConn
TCP  192.68.69.30:80 rr
  -> 142.133.69.33:80            Masq    1      0          0
  -> 142.133.69.34:80            Masq    1      0          0
  -> 127.0.0.1:80                Local   0      0          0
```

By default, the LDirectord module uses the quiescent feature to add and remove traffic nodes. When a traffic node is to be removed from the virtual service, its weight is set to zero and it remains part of the virtual service. As such, exiting connections to the traffic node may continue, but the traffic node is not allocated any new connections. This mechanism is particularly useful for gracefully taking real servers offline. This behavior can be changed to remove the real server from the virtual service by setting the global configuration option `quiescent=no`.

3.19.2 Contributions to the LDirectord Module

LDirectord is a pre-existing software module. The improvements and adaptations to the LDirectord module include new capabilities such as connecting to the traffic manager and the traffic client. These functionalities did not exist because the traffic manager and traffic client are contributions from this dissertation. The need arise for the LDirectord to be able to communicate the failure of the failure of the application to the traffic client and report it to the traffic manager so that the traffic manager removes the specific traffic node from its list of available node. As future work, we would like to minimize the number of sequential steps to improve the performance, and investigate the idea of integrating the LDirectord module with the traffic client into a single software module.

3.20 CVIP: A Cluster Virtual IP Interface for the HAS Architecture

One of the challenges is to present the cluster as a single entity to the end users. To overcome the disadvantage of existing solutions presented in Sections 2.14 and 2.15, the HAS architecture provides a transparent and scalable interface between the internet and the cluster called the cluster virtual IP interface (CVIP). It is fault-tolerant and scalable interface between the Internet and the HAS architecture that provide a single entry point to the HAS cluster. It has not impact on web clients, applications and the existing network infrastructure. It does not present a SPOF, have minimal impact on performance, and have minimal impact in case of failure. In addition, it is scalable to allow a large number of transactions (virtually an unlimited number) without posing a bottleneck, and to allow the increase of the number of master nodes and traffic nodes independently. Furthermore, it supports IPv4 and IPv6, and it is application independent.

3.20.1 Description of CVIP

CVIP is a fault-tolerant and scalable method of interfacing a plurality of application servers running on the HAS cluster. Figure 49 illustrates the generic configuration of the CVIP interface. The HA tier consists of two master nodes that are in the 1+1 active/standby redundancy model. Incoming connections to the HAS cluster arrive at the active master node, owner of the CVIP address. The CVIP interface receives incoming data packets from the Internet or a packet data network and passes packets to the load balancer running on the master node in the HAS architecture. If the packet corresponds to an existing connection, the load balancer forwards the request to an already existing connection. If the packet

corresponds to a new connection, then the load balancer chooses a traffic node from the HAS cluster and forward the request to it.

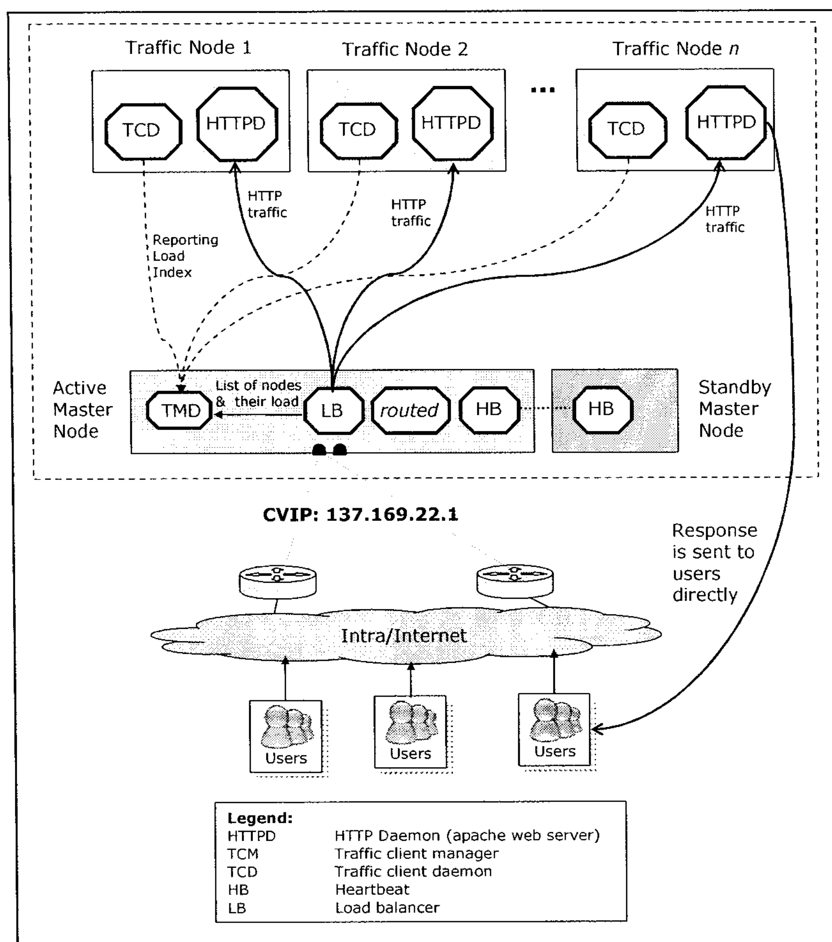


Figure 49: Generic CVIP configuration

The CVIP includes a plurality of network terminations on the master nodes in the HAS architecture that receive incoming data packets from the Internet, and a plurality of forwarding processes that are associated with the network terminations.

Figure 50 illustrates the concept of a network termination. A network termination is the last stop on the network for a given connection (data packet) before it is forwarded for processing inside the HAS architecture. The network termination of a web request packet is the network card interface on the master node in the HA tier of a HAS cluster. Each network termination carries its own IP address and can be addressed directly. These addresses are published via RIP/OSPF/BGP to the routers indicating them as

gateway addresses for the CVIP address; this means that routers see each termination just as another router.

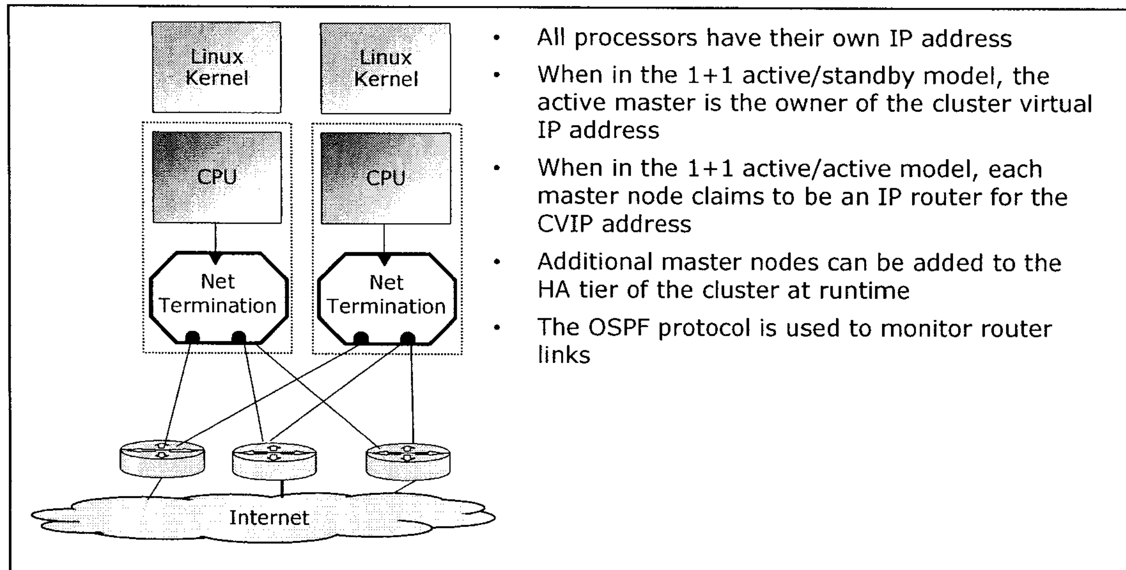


Figure 50: Network terminations

3.20.2 Advantages of CVIP

CVIP is an interfacing that provides fault tolerance and close to linear scalability of the servers and the network interfaces. It is transparent to the web clients and to the HAS cluster servers, and has minimal impact on the surrounding network infrastructure. The following sub-sections present the advantages of CVIP.

Transparency and single entry point to the cluster

CVIP is transparent to the applications running on traffic nodes and web clients are not aware of it. CVIP supports multiple protocols and applications that use TCP, UDP and raw IP sockets, are able to use it transparently. CVIP hides the cluster internals from the users and makes the cluster visible to the outside world as a single entity through a virtual IP address. For the outside world, service is available through a certain web address, which is the virtual cluster IP address that masks behind it the IP addresses of the master nodes. It allows access to a cluster of processors via a single IP address. We can also define a number of CVIP addresses in a system and achieve communication between web clients and applications using different CVIP addresses in the cluster.

Scalable

The CVIP offers a scalability advantage because we can increase network terminations, master nodes, or traffic nodes independently and without any affecting how the cluster is presented to the outside world. We achieve network bandwidth scalability by increasing the number of master nodes and network terminations. We achieve capacity scalability by increasing the number of servers in the SSA tier.

With CVIP, we can cluster multiple servers to use the same virtual IP address and port numbers over a number of processors to share the load. As we add new nodes in the HA and SSA tiers, we increase the capacity of the system and its scalability to handle increased traffic with the same virtual IP address. The number of clients or servers using the virtual IP address is not limited, and we can add more servers to increase the system capacity. In addition, although we have only presented HTTP servers, the applications on top may include server application that runs on IP such as an FTP server for file transfer.

Fault tolerance

The interface hides errors that take place on the HAS nodes and it is always available. Any crash in the application server is transparent to end users. In the event the web server software crashes, only 1/N of the ongoing transactions are lost (only if there is no connection synchronization between master nodes). All ongoing transactions can be saved and the state information can be preserved.

Availability

Since CVIP supports multiple servers, it does not provide a SPOF. In the HAS architecture prototype, CVIP was provided on two master nodes in the HA tier. If one master node crashes, the web clients and web servers are not affected.

Dynamic connection distribution

The framework supports a traffic distribution mechanism discussed in Section 3.22 that provides web clients with access to application servers running on traffic nodes in a transparent way. When the CVIP receives incoming traffic, it is possible to choose between two different distribution algorithms: the round robin distribution mechanism or the HAS distribution mechanism that provides dynamic and high performance distribution (discussed in 3.22).

Support for multiple application servers

CVIP operates at IP level and it is transparent to application servers running on traffic nodes. CVIP is independent from the type of traffic CVIP receives and forwards to traffic node. With CVIP, the HAS architecture supports all types of application servers that work at IP level. It allows transparent access transparently the application servers running on traffic nodes.

3.21 Connection Synchronization

When the HA tier follows the 1+1 active/standby redundancy model, the cluster achieves a higher availability than a single node tier, since there are two master nodes, one is active and the second is standby. When the active master node fails, the standby master node automatically takes over the IP address of the virtual service, and the cluster continues to function. However, when a fail-over occurs at the active master node, ongoing connections that are in progress terminate because the standby master node does not know anything about them. To improve this situation and to prevent the loss of ongoing connections, there is a need to provide the capabilities of synchronizing the ongoing connections information between the active master node and the standby master node. As a result, the cluster can minimize and eliminate the situation of lost connections caused by the failure of an active master node. When the information of ongoing connections is synchronized between master nodes, then if the standby master node becomes the active master node, it retains the information about the currently established and active connections, and as a result, the new active master node continues to forward their packets to the traffic nodes in the SSA tier.

3.21.1 The Challenge of Connection Synchronization

When a master node receives a packet for a new connection, it allocates the connection to a traffic node. This allocation in effected by allocating an `ip_vs_conn` structure in the Linux kernel which stores the source address and port, the address and port of the virtual service, and the traffic node address and port of the connection. Each time a subsequent packet for this connection is received, this structure is looked up, and the packet is forwarded accordingly.

When fail-over occurs, the new master node does not have the `ip_vs_conn` structures for the active connections. Therefore, when a packet is received for one of these connections, the new master node does not know which to which real server it should forward it. As a result, the connection breaks and the web

user need to reconnect. By synchronizing the `ip_vs_conn` structures between the master nodes, this situation ca be avoided, and connections can continue after a master node fails-over.

3.21.2 The Master/Slave Approach to Connection Synchronization

The connection synchronization code relies on a sync-master/sync-slave setup where the sync-master sends synchronization information and the sync-slave listens. The following example, paraphrased from [30], illustrates this scenario. There are two master nodes: Master-A is the sync-master and the active master node, and Master-B is the sync-slave and the standby master node.

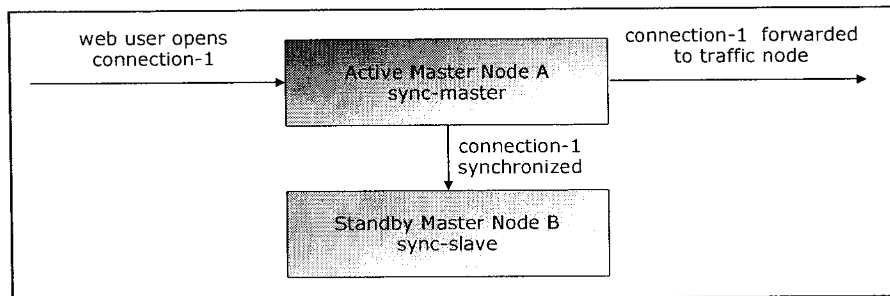


Figure 51: Step 1 - Connection Synchronization

In step 1 (Figure 51), the web user opens connection-1. Master node A receives this connection, forwards it to a traffic node, and synchronizes it to the sync-slave, master node B.

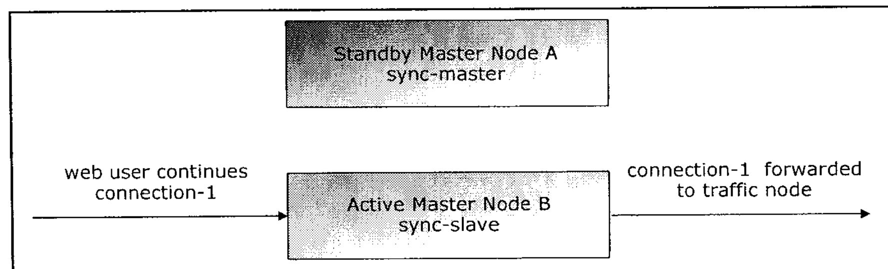


Figure 52: Step 2 - Connection Synchronization

In step 2 (Figure 52), a fail-over occurs and the master node B becomes the active master node. Connection-1 is able to continue because the connection synchronization took place in step 1.

The master/slave implementation of the connection synchronization works with two master nodes: the active master node sends synchronization information for connections to the standby master node, and the standby master node receives the information and updates its connection table accordingly.

The synchronization of a connection takes place when the number of packets passes a predefined threshold and then at a certain configurable frequency of packets. The synchronization information for the connections is added to a queue and periodically flushed. The synchronization information for up to 50 connections can be packed into a single packet that is sent to the standby master node using multicast. A kernel thread, started through an init script, is responsible for sending and receiving synchronization information between the active and standby master nodes.

3.21.3 Drawbacks of the Master/Slave Approach

The master/slave approach for synchronizing connections suffers from a drawback that is demonstrated when the new master node (previously standby) fails and the current standby node (previously active) becomes active again. To illustrate this drawback, we continue discussing the example of connection synchronization from the previous section.

In step 3 (Figure 53), a web user opens connection-2. Master node B receives this connection, and forwards it to a traffic node. Connection synchronization does not take place because master node B is a sync-slave and therefore it is not synchronizing its connections with master node A.

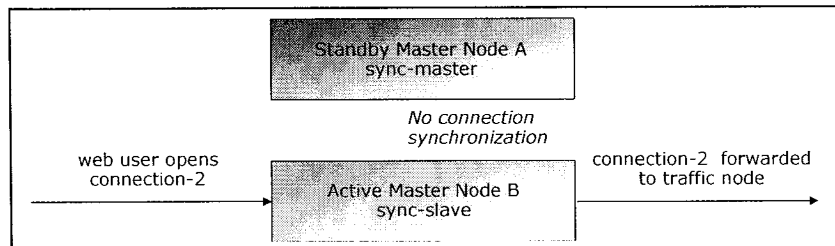


Figure 53: Step 3 - Connection Synchronization

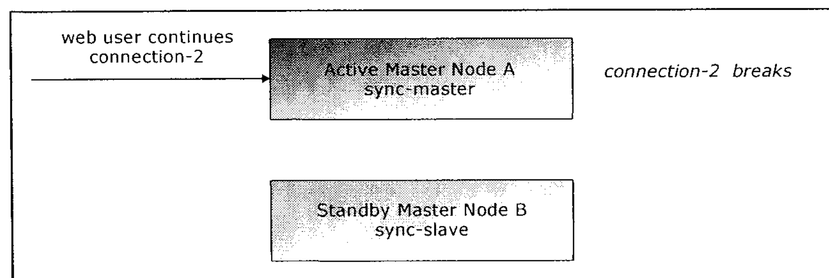


Figure 54: Step 4 - Connection Synchronization

In step 4 (Figure 54), another fail-over takes place and master node A is again the active master node. Connection-2 is unable to continue because it was not synchronized.

3.21.4 Alternative Approach: Peer-to-Peer Connection Synchronization

The master/slave approach to connection synchronization has synchronization limitation and therefore there is a need for a different approach that eliminates the drawback in the existing implementation.

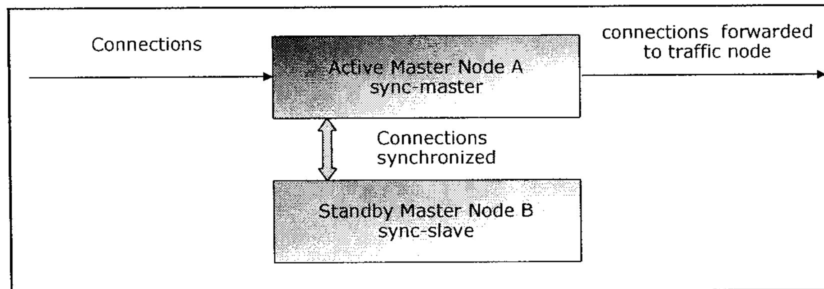


Figure 55: Peer-to-peer approach

Figure 55 illustrates the peer-to-peer approach. In this approach, each master node sends synchronization information for connections that it is handling to the other master node. Therefore, in the scenario illustrated in Figure 54, connections are synchronized from master node B to master node A; connection-2 are able to continue after the second fail-over when master node A becomes the active master node again. The implementation of this approach is a future work item.

3.22 Traffic Management

Traffic management is an important aspect of the HAS architecture that contributes to building a highly available and scalable web cluster. This section presents the requirements for a flexible and scalable traffic distribution inside a web cluster. It discusses the needs for a traffic management scheme and presents the HAS architecture solution that consists of a load balancer, a traffic manager, a traffic client, and a distribution policy.

3.22.1 Background and Requirements

Traffic distribution is the process of distributing network traffic across a set of server nodes to achieve better resource utilization, greater scalability, and high availability. Scalability is an important factor because it ensures a rapid response to each network request regardless of the load. Availability ensures the service continues to run despite failure of individual server nodes. Traffic distribution can be passive

or active technology depending on the specific implementation. Some traffic distribution schemes do not modify network requests, but pass them verbatim to one of the cluster nodes and returns the response verbatim to the client. Other schemes, such as NAT, change the request headers and force the request to go through an intermediate server before it reaches the end user.

There exist many interesting aspects of a traffic distribution implementation that we considered when designing and prototyping the HAS architecture traffic distribution scheme. These aspects include optimizing of response time, coping with failures of individual traffic nodes, supporting heterogeneous traffic nodes, ensuring the distribution service remains available, supporting session persistence, and ensuring transparency so that users are not aware where the application is hosted and if the server is a cluster or a single server.

Our survey of identical works (Sections 2.14 and 2.15) has identified that added performance from complex algorithms is negligible. The recommendations were to focus on a distribution algorithm that is uncomplicated, has a low overhead, and that minimizes serialized computing steps to allow for faster execution.

Scalable web server clusters require three core components: a scheduling mechanism, a scheduling algorithm, and an executor. The scheduling mechanism directs clients' requests to the best web server. The scheduling algorithm defines the best web server to handle the specific request. The executor carries out the scheduling algorithm using the scheduling mechanism. The following sub-section present these three core components in the HAS architecture.

3.22.2 Traffic Management in the HAS Architecture

Traffic management is a core technology that enable better scalability in the HAS architecture. It consists of four elements that work together to achieve efficient and dynamic traffic distribution among cluster traffic nodes. These elements are the load balancer and the traffic manager running on the master nodes, the traffic client daemons running on traffic nodes, and the traffic distribution policy. Traffic nodes continuously report to the traffic managers their availability and their load index. The traffic manager maintains the list of available nodes and their load index and makes it available for the load balancer that distributes incoming traffic to the traffic nodes.

When a user accesses a virtual service provided by the HAS cluster, the packet destined for the virtual IP address arrives to the master nodes, and it is forwarded to a traffic node based on the distribution policy in place. The traffic node processes the request and replies directly to the user. This scheme follows the

direct routing method, where the master nodes and traffic nodes have one of their interfaces physically linked by a hub or a switch.

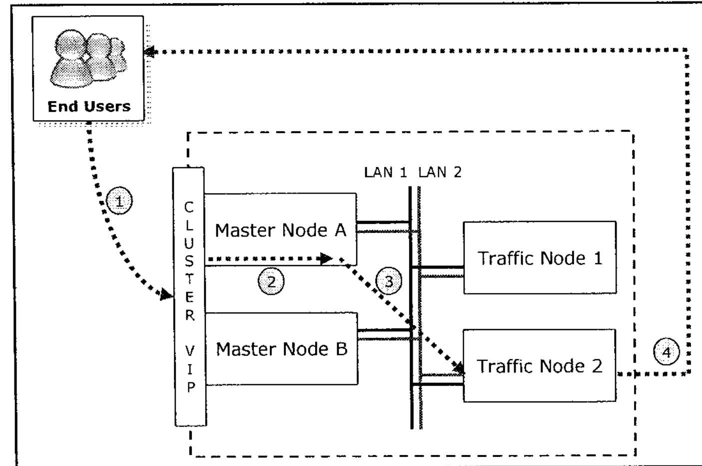


Figure 56: The direct routing approach – traffic nodes reply directly to web clients

Figure 56 illustrates the scenario of direct access. When a request arrives to the cluster (1), the master node examines it, decides where the request should be forwarded, and forwards it (3) to the appropriate traffic node. The request reaches the traffic node that treats it, and replies directly (4) to the web client. Based on the survey of related work (Sections 2.14 and 2.15), the direct access routing approach helps improve the system scalability. This model is supported by the HAS cluster prototype and with which we have performed our benchmarking tests.

3.22.3 Traffic Manager

The traffic manager runs on master nodes. It receives load announcements from the traffic client daemons running on traffic nodes, and updates its internal list of traffic nodes and their load. It maintains a list of available traffic nodes and their load index. The traffic manager maintains the list of available traffic nodes and makes it available to the load balancer. The traffic manager requires a configuration file that lists the addresses of all traffic nodes, the port of communication, the timeout limit, and the addresses of the master nodes.

3.22.3.1 Sample Traffic Manager Configuration File

This section presents sample configuration file of a traffic manager daemon.

```
1     # TMD Configuration File
2     # List of master nodes
3     master1       <IP Address of Master Node 1>
4     master2       <IP Address of Master Node 2>
5     # List the IP addresses of active traffic nodes
6     traffic1      <IP Address of Traffic Node 1>
7     traffic2      <IP Address of Traffic Node 2>
8     traffic3      <IP Address of Traffic Node 2>
9     traffic4      <IP Address of Traffic Node 2>
10    # List of IP addresses of Standby traffic nodes - Used _only for NxM
11    <IP Address of Traffic Node 1>
12    <IP Address of Traffic Node 2>
13    <IP Address of Traffic Node M>
14    # Port number
15    port          <port_number>
16    # Reporting errors - for troubleshooting
17    ErrorLog      <full_path_to_error_log_file>
18    # The Timeout option specifies the amount of time in ms the TMD
19    # waits to receive load info from TCD.
20    # Timeout should be > than the update frequency of the TCD
21    timeout       <timeout_value>
```

3.22.3.2 Improvements to the Traffic Manager

The current implementation of the traffic manager can use several improvements that include further testing and stabilizing of the source code, optimizing insertion, and updates of the list of traffic nodes and their load index. As future work, we would like to investigate the possibility of merging the functionalities of the *saru* module (connection synchronization) with the traffic manager.

3.22.4 The Proc File System

The /proc file system is a real time, memory resident file system that tracks the processes running on the machine and the state of the system, and maintains highly dynamic data on the state of the operating system. The information in the /proc file system is continuously updated to match the current state of the operating system. The contents of the /proc files system are used by many utilities which read the data from the particular /proc directory and display it.

The traffic client uses two parameters from /proc to compute the load_index of the traffic node: the processor speed and free memory. The /proc/cpuinfo file provides information about the processor, such as its type, make, model, cache size, and processor speed in BogoMIPS. The BogoMIPS parameter is an internal representation of the processor speed in the Linux kernel.

Figure 57 illustrates the contents of the /proc/cpuinfo file at a given moment in time and highlights the BogoMIPS parameter used to compute the load_index of the traffic node. The processor speed is a constant parameter; therefore, we only read the /proc/cpuinfo file once when the TC starts. The /proc/meminfo file reports a large amount of valuable information about the RAM usage. The /proc/meminfo file contains information about the system's memory usage such as current state of physical RAM in the system, including a full breakdown of total, used, free, shared, buffered, and cached memory utilization in bytes, in addition to information on swap space.

```
% more /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 13
model name      : Intel(R) Pentium(R) M processor 1.70GHz
stepping        : 6
cpu MHz         : 598.186
cache size      : 2048 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr mce cx8 sep mtrr pge mca
cmov pat clflush dts acpi mmx fxsr sse sse2 ss tm pbe est tm2
bogomips        : 1185.43
```

Figure 57: The CPU information available in /proc/cpuinfo

Figure 58 illustrates the contents of the /proc/meminfo file at a given moment in time and highlights the MemFree parameter used to compute the load_index of the traffic node. Since the MemFree is a dynamic parameter, it is read from the /proc/meminfo file every time the TC calculates the load_index.

```
% more /proc/meminfo
MemTotal:          775116 kB
MemFree:             6880 kB
Buffers:            98748 kB
Cached:            305572 kB
SwapCached:          2780 kB
Active:            300348 kB
Inactive:          286064 kB
HighTotal:              0 kB
HighFree:               0 kB
LowTotal:          775116 kB
LowFree:             6880 kB
SwapTotal:        1044184 kB
SwapFree:         1040300 kB
Dirty:                 16 kB
Writeback:              0 kB
Mapped:            237756 kB
Slab:              171064 kB
Committed_AS:      403120 kB
PageTables:          1768 kB
VmallocTotal:      245752 kB
VmallocUsed:        11892 kB
VmallocChunk:      232352 kB
  HugePages_Total:      0
  HugePages_Free:       0
```

Figure 58: The memory information available in `/proc/meminfo`

3.22.5 Traffic Client

The traffic client is a daemon process that runs on each traffic node. It collects processor and memory information from the `/proc` file system and reports it to the traffic managers running on master nodes. The implementation of the traffic client requires a configuration file (Section 3.22.5.1) that lists the addresses of master nodes, port of communication, timeout limits, and the logging directive.

In the event of failure of the traffic client daemon, the traffic manager does not receive a load notification and after a timeout, it removes the traffic node from its list of available traffic nodes.

3.22.5.1 Sample Traffic Client Configuration File

This section presents sample configuration file of a traffic client daemon.

```
1    # TC Configuration File
2    # List of master nodes to which the TC daemon reports load
3    master1     <IP Address of Master Node 1>
4    master2     <IP Address of Master Node 2>
```

```
5       # Port number to connect to at master - this port number can be
6       # between 1024 and 49151, and between 49152 through 65535
7       port          <port_number>
8       # Frequency of load updates in ms
9       updates       <frequency_of_updates>
10      # Reporting errors -- needed for troubleshooting purposes
11      ErrorLog      <full_path_to_error_log_file>
12      # Number of RAM in the node with the least RAM in the cluster
13      RAM           <num_of_ram>
```

3.22.5.2 Improvements to the Traffic Client

As future work, we plan to investigate how to have a better representation of the load index, by adding for instance the network bandwidth parameter into the load index computation. In addition, we would like to investigate the possibility of merging the functionalities of the LDirectord module with the traffic client resulting in one system software that reports the node load index and monitors the health of the application server running on the traffic node.

3.22.6 Characteristics of the Traffic Management Scheme

The traffic management scheme supports static distribution with round robin without taking into consideration the load of traffic nodes. It also supports dynamic distribution that is configurable to support various intervals of load updates. The HAS cluster can include nodes with heterogeneous hardware such as with different processor speeds and memory capacity. The traffic distribution mechanism is aware of the variation in computing power since the processor speed, reported in BogoMIPS, and memory reported in MB, are factored in the load index formula. As a result, the load balancer assigns traffic to the traffic nodes based on their load index. The traffic client uses the following contributed new formula to calculate the load index of each traffic node:

$$Load_a = \left(\frac{CPU_a * RAM_a}{RAM_x} \right)$$

The formula consists of the following variables:

- CPU_a is the BogoMIPS representation of the processor speed of traffic node *a*
- RAM_a is the number of free RAM in MB of traffic node *a*, and

- RAM_x is the number of total RAM in MB of traffic node *x*, where traffic node *x* is the node with the least amount of RAM among all traffic nodes. The configuration file of the traffic client specifies this parameter (Section 3.22.5.1).

The result of the computation is the relative load of traffic node a.

We can expand the `load_index` formula to support other parameters such as available bandwidth. This is a future work item.

Let us consider an example of how the formula is applied. Consider a traffic node with a Pentium Mobile processor running at a speed of 1.7 GHz with 748 MB of RAM. The BogoMIPS processor speed is 3358. We assume that the node has 512 MB of free RAM. Let us consider that the cluster node with the lowest amount of RAM has 256 MB of RAM. Therefore, the load of that machine as reported by the traffic client daemon to the traffic manager is:

$$Load_a = \left(\frac{3358{,}72 * 524{,}288}{262{,}144} \right) \approx 6{,}717$$

Table 11 illustrates an example of a cluster that consists of eight traffic nodes.

Traffic Node IP Address	Load Index
192.168.1.100	4484
192.168.1.101	4397
192.168.1.102	4353
192.168.1.103	4295
192.168.1.104	4235
192.168.1.105	4190
192.168.1.106	4097
192.168.1.107	3973

Table 11: Example list of traffic nodes and their load index

When the load balancer receives an incoming request, it examines the list of nodes and their load index and forwards the request to the least busy node on the list. The list of traffic nodes is a sorted linked list that allows us to maintain an ordered list of nodes without having to know ahead of time how many nodes we will be adding. To build this data structure, we used two class modules: one for the list head and another for the items in the list. The list is a sorted linked list; as we add nodes into the list, the code finds the correct place to insert them and adjusts the links around the new nodes accordingly. The traffic

management scheme is lightweight and configurable. The traffic management scheme is able to cope with failures of individual traffic nodes. It supports a cluster that consists of heterogeneous cluster nodes with different processor speed and memory capacity. The scheme is transparent to web clients who are unaware that the applications run on a cluster of nodes.

3.22.7 Example of Operation

Figure 59 illustrates the interaction between the traffic client and traffic manager, when the traffic client is reporting the load index of the traffic node to the traffic manager. The traffic client reads the `/proc` file system and retrieves the BogoMIPS processor speed from `/proc/cpuinfo`, and then retrieves the amount of free memory from `/proc/meminfo` (1). Next, the traffic client computes the node load index based on the formula presented in Section 3.22.6 (2). The traffic client then reports two parameters to the traffic manager: the traffic node IP address and the load index (`traffic_node_IP, load_index`) (3). The traffic manager receives the load index and updates its internal list of traffic nodes to reflect the new `load_index` of the `traffic_node_IP` (4).

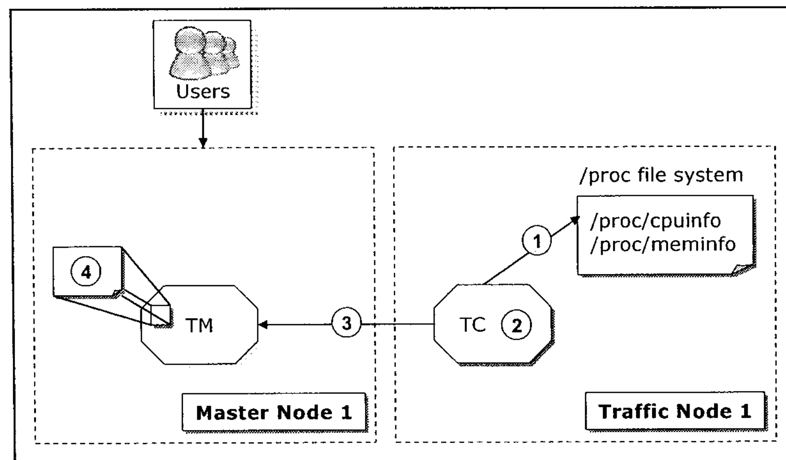


Figure 59: The interaction between the traffic client and the traffic manager

3.22.8 Dependencies of Software Modules

The HAS architecture consists of many software modules that depend on each other to provide service. Figure 60 illustrates the dependencies and interconnections of the various software module within the HAS architecture in relation to traffic distribution.

The following subsections discuss the dependencies among these software modules, and present how they interact with each other. As a future work item, we plan to consolidate the functionalities of these system

software modules resulting in fewer modules and lesser interactions in order to eliminate pre-processing steps.

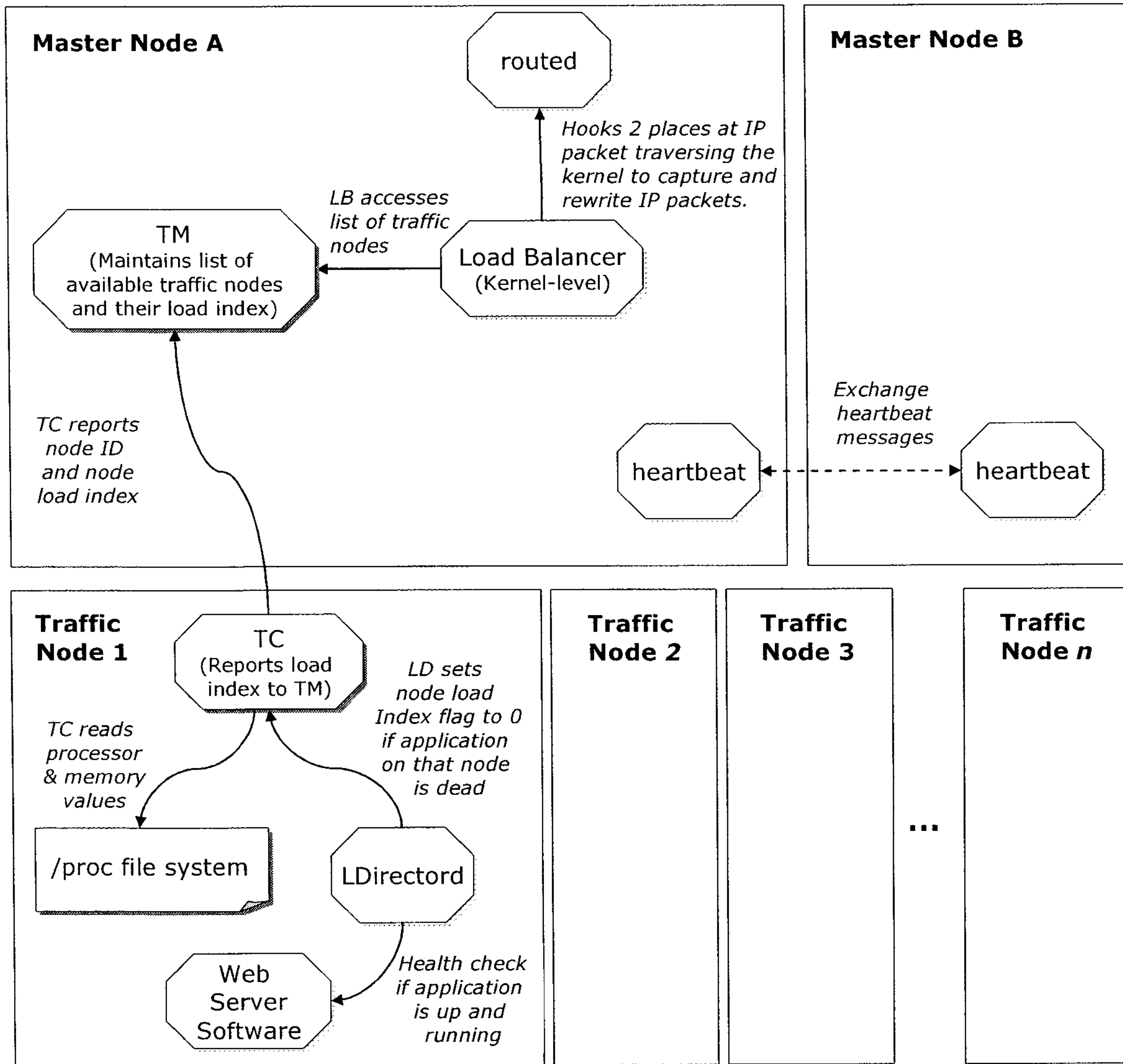


Figure 60: Interaction and dependencies of software modules

Load Balancer and Traffic Manager

The load balancer hooks in two places at IP packet traversing the kernel to capture and rewrite IP packets. It looks up the virtual server rules hash table for new connections, and checks the connection hash table for established connections.

The traffic manager maintains the list of available traffic nodes and their load index. The load balancer uses this list to determine to which traffic nodes it will forward incoming traffic.

Traffic Manager and Traffic Client

The traffic managers, running on master nodes, depend on the traffic clients running on traffic nodes to receive the load index used by the traffic distribution algorithm to forward traffic to incoming requests. If a traffic client daemon does not send the load index to the traffic managers, traffic managers assume that that specific traffic node is unavailable and as a result stop sanding traffic to it. Therefore, we need to ensure that traffic managers are aware when a traffic node becomes unavailable, when a traffic client daemon fails, and when the application fails on the traffic node.

The LDirectord Module and the Traffic Client

The traffic client reports the load of the traffic node to the traffic manager. This interaction is used as a health check between the traffic client and the traffic manager. If the traffic client fails to report within a specific time, and the timeout is exceeded, then the traffic manager consider the traffic node as unavailable and removes it from its list of active nodes. However, there is a case scenario where the traffic client is reporting the load index to the traffic manager but it is unaware that the application is unavailable. The role of the LDirectord module is to ensure that the traffic client does not update the `load_index` while the application is not responsive. Therefore, the LDirectord sets the traffic client `load_index_report_flag` to 0. When the `load_index_report_flag` = 0, the traffic client stops reporting its load to the traffic manager. When the application check performed by LDirectord returns positive (the application is up and running), the LDirectord sets the `load_index_report_flag` to 1. As a result, the traffic client restarts to report its load index to the traffic manager and then the traffic manager adds the traffic node to its list of available nodes.

Traffic Client and the /proc File System

The traffic client daemon retrieves the memory and processor usage metrics from the `/proc` file system and computes the `load_index` of a traffic node.

3.23 Ethernet Redundancy Daemon Contribution

The Ethernet redundancy daemon is an original contribution of this work. The goal with the Ethernet Redundancy daemon is to maintain Ethernet connectivity at the server node level. It monitors the link

status of the primary Ethernet port. On link down, the route of the first port is deleted, and incoming traffic is directed to the second Ethernet port. When the link goes up again, the daemon waits to make sure the connection does not drop again, and then switches back to the primary Ethernet port.

The Ethernet redundancy daemon (*erd*) runs after the "`route`" configuration data has been installed, following boot up. In order to have link redundancy, is it necessary to configure the paired ports (`eth0 and eth1`) with the same IP and MAC addresses.

The "*erd*" program starts by fetching the configuration of all routes and then storing them into a table (loaded from `/proc/net/route`). The table is used to compose *route deletion* commands for all routes bound to the secondary link (`eth1`). The route configuration for the primary link (`eth0`) is then copied to the secondary link (`eth1`) with a higher metric (meaning a lower priority). The primary link (`eth0`) is then polled every *x* milliseconds (*x* is a configurable parameter in the configuration file). When a primary link fails, all routing information is deleted for the link, causing traffic to be switched to the secondary link. Upon link restoration, the cached routing information for the primary link is updated, forcing a traffic switchback. This contribution is presented and discussed in [23].

3.23.1 Command Line Usage

The command has specific usage syntax: `% erd [eth0 eth1]`

The example shown above indicates that `eth0` has `eth1` as its backup link. If we do not specify parameters in the command, it defaults to the equivalent of "`erd eth0 eth1`". We automated this command on system startup.

3.23.2 Encountered Issues

The servers in our prototype use Tulip Ethernet cards. We patched the `tulip.c` driver to make the MAC addresses for ports 0 and 1 identical. Alternatively, we also were able to get the same result by issuing the following commands (on Linux) to set the MAC address for an Ethernet port:

```
% ifconfig eth[X] down
% iconfig eth[X] hw ether AA:BB:CC:DD:EE:FF
% ifconfig eth[X] up
```

We also modified the source code of the Ethernet device driver `tulip.c` to toggle the `RUNNING` bit in the `dev→flags` variable, which allows `ifconfig` to present the state of an Ethernet link. The state of the `RUNNING` bit for the primary link is accessed by *erd* via the system call `ioctl`.

3.23.3 Improvements to the Ethernet Redundancy Daemon Contribution

Further improvements to the current implementation include stabilizing the source and optimizing the performance of the Ethernet daemon, which include optimizing the failure detection time of the Ethernet driver. In addition, the source code of the Ethernet redundancy daemon is to be upgraded to run on the latest release of the Linux kernel, version 2.6.

3.24 Scenario View of the Architecture

The scenario view consists of a small subset of important scenarios – instances of use cases – that illustrate how the elements of the HAS architecture work together. For each scenario, we describe the corresponding sequences of interactions between the various objects and processes. The following sub-sections examine these scenarios:

- *Normal scenario:* This scenario describes how the HAS cluster receives and processes a web request.
- *Traffic client daemon reporting its load:* This scenario describes how a traffic node computes and reports its load index to the traffic managers running on master nodes.
- *Addition of a traffic node:* This scenario describes the steps involved in adding a traffic node to the SSA tier.
- *Boot process of a traffic node:* This scenario describes the boot process of a traffic node. There are two variations of this scenario depending on whether the traffic node is diskless or it has local disk.
- *Upgrading the operating system and application server software:* This scenario describes upgrading the kernel and applications on a HAS cluster node.
- *Hardware upgrade on master node:* This scenario describes the process of upgrading a hardware component on a master node.
- *Master node becomes unavailable:* This scenario describes the events that occur when a master node becomes unavailable.
- *Traffic node becomes unavailable:* A traffic node can become unavailable because of hardware or software error. This scenario describes how the cluster reacts to the unresponsiveness of a traffic node.
- *Ethernet port becomes unavailable:* A cluster node can face networking problems because of Ethernet card or Ethernet drivers issues. This scenario examines how a HAS cluster node reacts when it faces Ethernet problems.

- *Traffic node leaving the cluster:* When a traffic node is not available to serve traffic, the traffic manager disconnects it from the cluster. This scenario illustrates how a traffic node leaves the cluster.
- *Application server process dies on a traffic node:* When the application becomes unresponsive, it stops serving traffic. This scenario examines how to recover from such a situation.
- *Network becomes unavailable:* This scenario presents the chain of events that take place when the network to which the cluster is connected, becomes unavailable.

3.24.1 Normal Scenario

The normal scenario describes how the HAS cluster processes an incoming web request. Figure 61 illustrates the sequence diagram of a successful request. This scenario assumes that the nodes in the HA tier follow the 1+1 active/standby redundancy model. The web user issues a request for a specific service that the cluster provides through its virtual interface (1). The load balancer examines the destination address and the port. If they are matched for a virtual service, then the load balancer picks a traffic nodes from the list of available nodes (2). The load balancer then changes the MAC address of the data frame to be the same of the traffic node (3) and retransmits it to the chose traffic node (4). The traffic node receives the requests, processes it, and returns the reply directly to the web user (5).

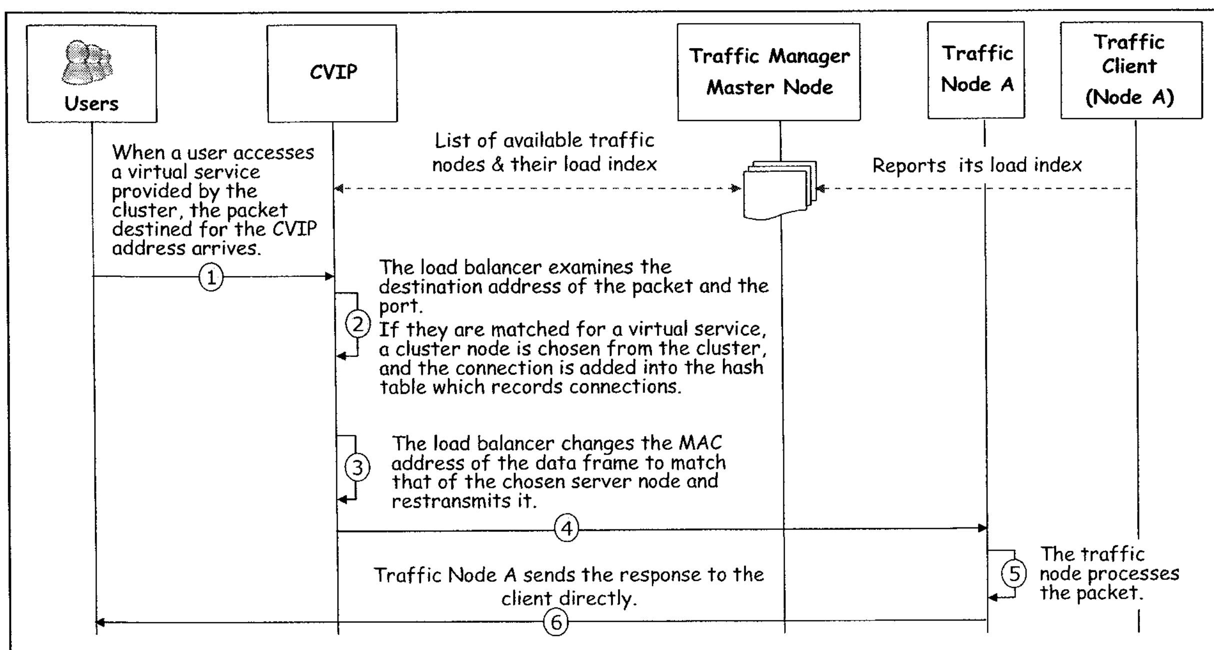


Figure 61: The sequence diagram of a successful request

3.24.2 Traffic Node Reporting Load Information

Figure 62 illustrates the sequence diagram of a traffic node reporting its load information to the traffic managers running on master nodes. This scenario assumes that each traffic node runs a copy of the traffic client daemon. When the traffic client daemon starts, it loads its configuration from the `/etc/tcd.conf` configuration file. As a result, it obtains the IP addresses of the master nodes to which it reports its load, the port number, the frequency of the load index update, and the location where it logs errors. The traffic client daemon then reads the processor and memory information from the `/proc` file system (1), computes the load index (Section 3.22.5), and reports it (2) to traffic managers running on the master nodes. The traffic managers receive load index of the traffic node, and update their list of available traffic nodes and their load indexes (4)(5). The frequency of reporting the load of the traffic node is defined in the configuration file of the traffic client. Due to the repetitive nature of this activity, and since the processor information is static, unlike memory information, the processor information is read from the `/proc` file system on the first time the traffic client is started.

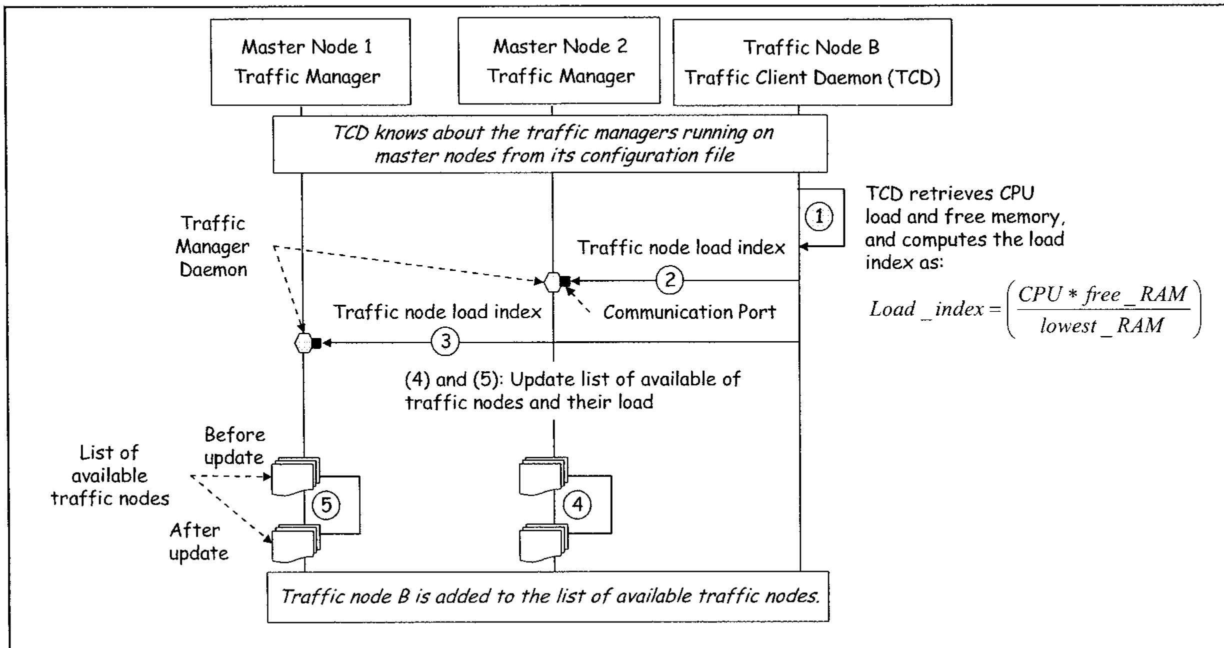


Figure 62: A traffic node reporting its load index to the traffic manager

3.24.3 Adding a Traffic Node to the HAS Cluster

The HAS architecture allows the addition of traffic nodes transparently and dynamically to the cluster in response to increased traffic. The HAS cluster administrator configures the traffic node to boot from

LAN. The traffic node boots and downloads a kernel image and a ramdisk (Section 3.24.4.1). The ramdisk includes three software modules that are started automatically: the Ethernet redundancy daemon, the LDirectord, and the traffic client.

Figure 63 illustrates the scenario of a traffic node joining the HAS cluster. When the traffic client starts, it reads the processor speed and available memory from the `/proc` file system and computes the load index (1). The traffic client then reports its load index to the traffic managers running on the master nodes in the HA tier as a pair of traffic node IP address and the load index (2)(3). The traffic managers receive the notification and update their list of available traffic nodes (4)(5).

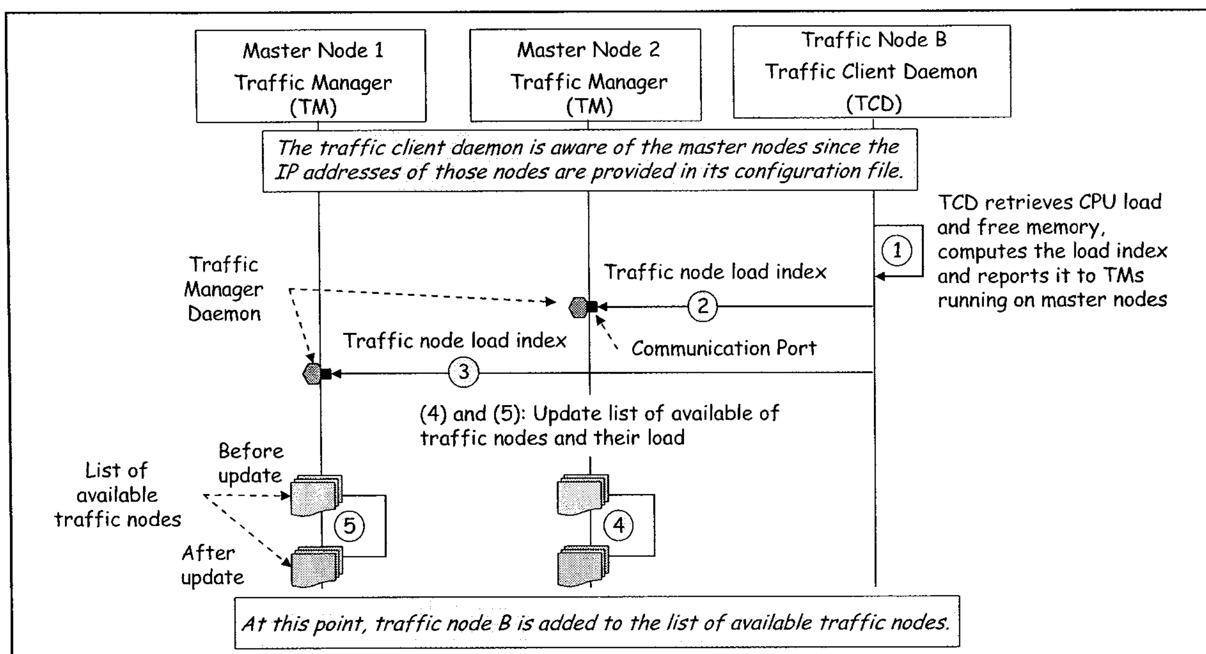


Figure 63: A traffic node joining the HAS cluster

3.24.4 Boot Process of a Traffic Node

There are two types of traffic nodes, diskless nodes and nodes with local disk, and each boots differently. The following sub-sections describe both booting methods, but first, we examine how the cluster operates. The nodes in the cluster boot from the network. When the nodes boot, they broadcast their Ethernet MAC address looking for a DHCP server. The DHCP server, running on the master nodes and configured to listen for specific MAC addresses, responds with the correct IP address for the nodes. Alternately, the DHCP server responds to any broadcast on its physical network with IP information from a designated range of IP addresses. The nodes receive the network information they need to configure

their interfaces: IP addresses, gateway, netmask, domain name, the IP address of the image and boot server, and the name of the boot file. Next, the nodes download a kernel image from the boot server, which can also be the master node. The boot server responds by sending a network loader to the client node, which loads the network boot kernel. The network boot loader mounts the root file system as read-only; the network loader reads the network boot kernel sent from the boot server into local memory and transfers control to it. The kernel mounts root as read/write, mounts other file systems, and starts the init process. The init process brings up the customized Linux services for the node, and the node is now fully booted and all initial processes are started.

3.24.4.1 Boot Process of a Diskless Traffic Node

Figure 64 illustrates the process of adding a diskless node to a running HAS cluster. The starting assumption is that the MAC address of the NIC on that diskless node is associated with a traffic node and configured on the master nodes as a diskless traffic node. The notion of diskless is important since the traffic node will download from the imager server a kernel and ramdisk image.

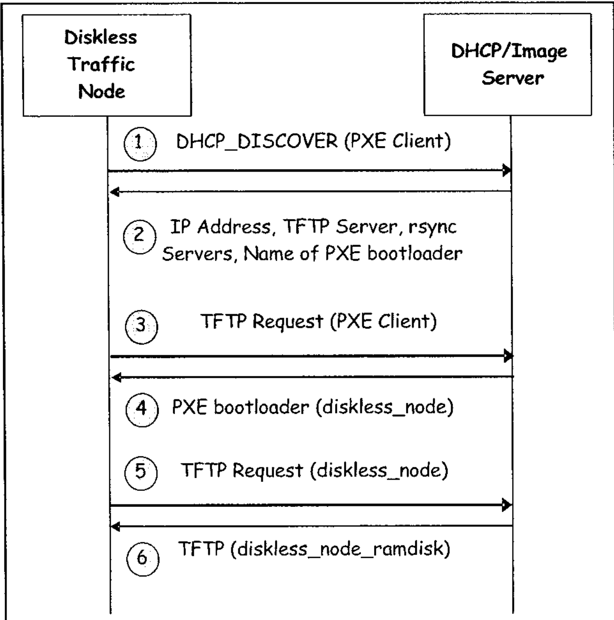


Figure 64: The boot process of a diskless node

Traffic nodes have their BIOS configured to do a network boot. When the administrator starts traffic nodes, the PXE client that resides in the NIC ROM, sends a DHCP_DISCOVER message (1). The DHCP

server, running on the master node, sends the IP address for the node with the address of the TFTP server and the name of the PXE bootloader file that the diskless traffic node should download (2). The NIC PXE client then uses TFTP to download the PXE bootloader (3). The diskless traffic node receives the kernel image (`diskless_node`) and boots with it (4). Next, the diskless traffic node sends a TFTP request to download a ramdisk (4). The image server sends the ramdisk to the diskless traffic node (5). The diskless traffic node downloads the ramdisk and executes it (6).

When the diskless traffic node executes the ramdisk, the traffic client daemon starts and reports the load of the node periodically to the master nodes. The traffic manager, running on the master nodes, adds the traffic nodes to its list of available traffic nodes.

3.24.4.2 Boot Process of a Traffic Node with Disk

There are two scenarios to add a node with disk to the HAS cluster. The scenarios differ if the traffic node has the latest versions of the operating system and the application server or if the traffic node requires the upgrade of either the operating system or the application server.

Figure 65 illustrates the sequence diagram of a traffic node with disk that is booting from the network. If the traffic node requires the upgrade of either the operating system or the application server, the traffic node downloads the updated operating system image and then a ramdisk image including the newer version of the web server application.

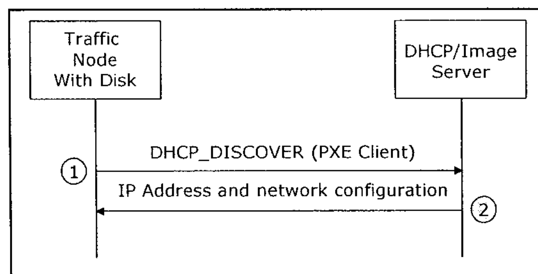


Figure 65: The boot process of a traffic node with disk

Figure 66 illustrates the process of upgrading the ramdisk on a traffic node. To rebuild a traffic node or upgrade the operating system and/or the ramdisk image, we re-point the symbolic link in the DHCP configuration to execute a specific script, which results in the desired upgrade. At boot time, the DHCP server checks if the traffic node requires an upgrade and if so, it executes the corresponding script.

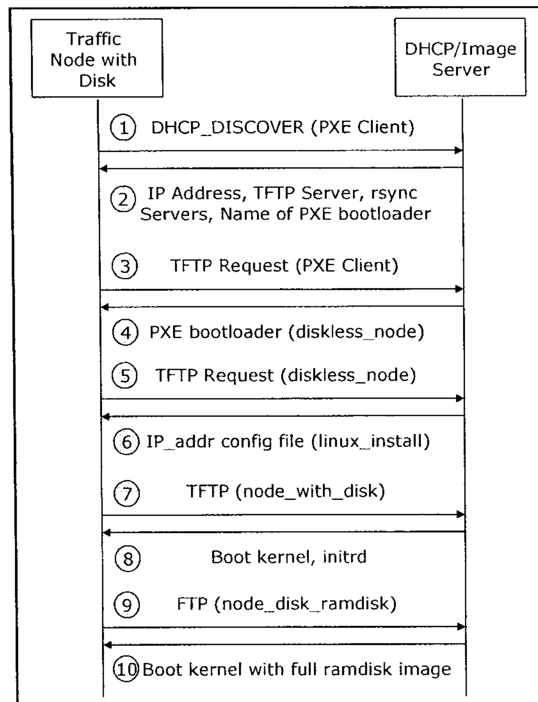


Figure 66: The process of rebuilding a node with disk

3.24.5 Upgrading Operating System and Application Server

The architecture promises service continuity even in the event of upgrading the operating system and/or the application servers running of the traffic nodes.

Figure 67 illustrates the events that take place when we initiate an upgrade of both the kernel and the application servers on a specific traffic node. While a traffic node is undergoing a software upgrade, traffic arrives to the cluster and the load balancer forwards it to the available traffic nodes. Following this model, we upgrade the software stack on the cluster nodes without having a service downtime.

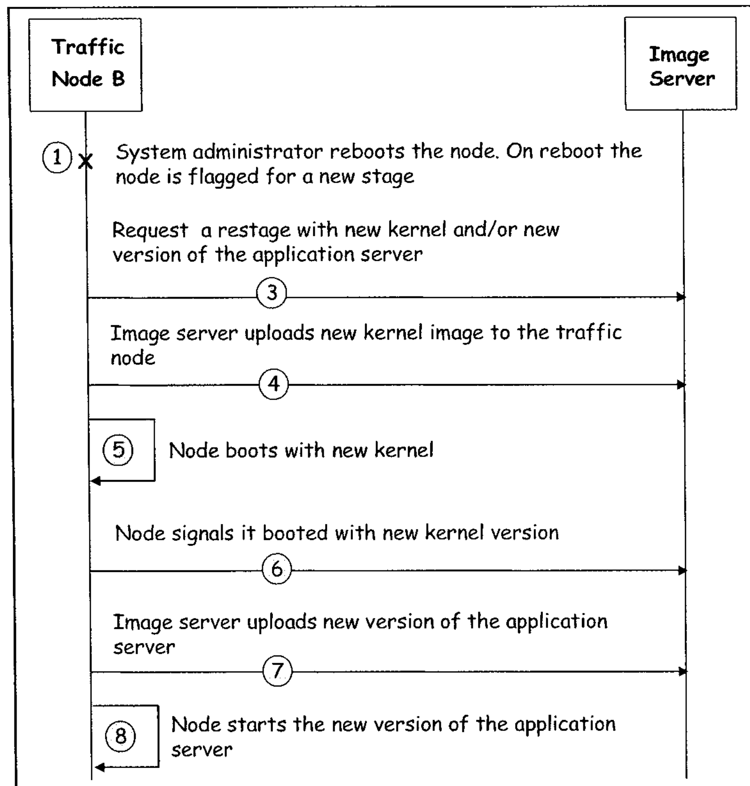


Figure 67: The process of upgrading the kernel and application server on a traffic node

3.24.6 Upgrading Hardware on Master Node

Figure 68 illustrates the events that take place when a master node is undergoing a hardware upgrade. The administrator of the system shuts down (1) the master node 1, which becomes offline (2). The heartbeat service on master node 1 is not available anymore (3) to reply to heartbeat messages sent to it from the heartbeat instance running on master node 2. After a timeout (specified in the heartbeat configuration file), master node 2 becomes the active master node (4) and the owner of the virtual services offered by the cluster. As a result, all incoming traffic arrives to the only available master node (5).

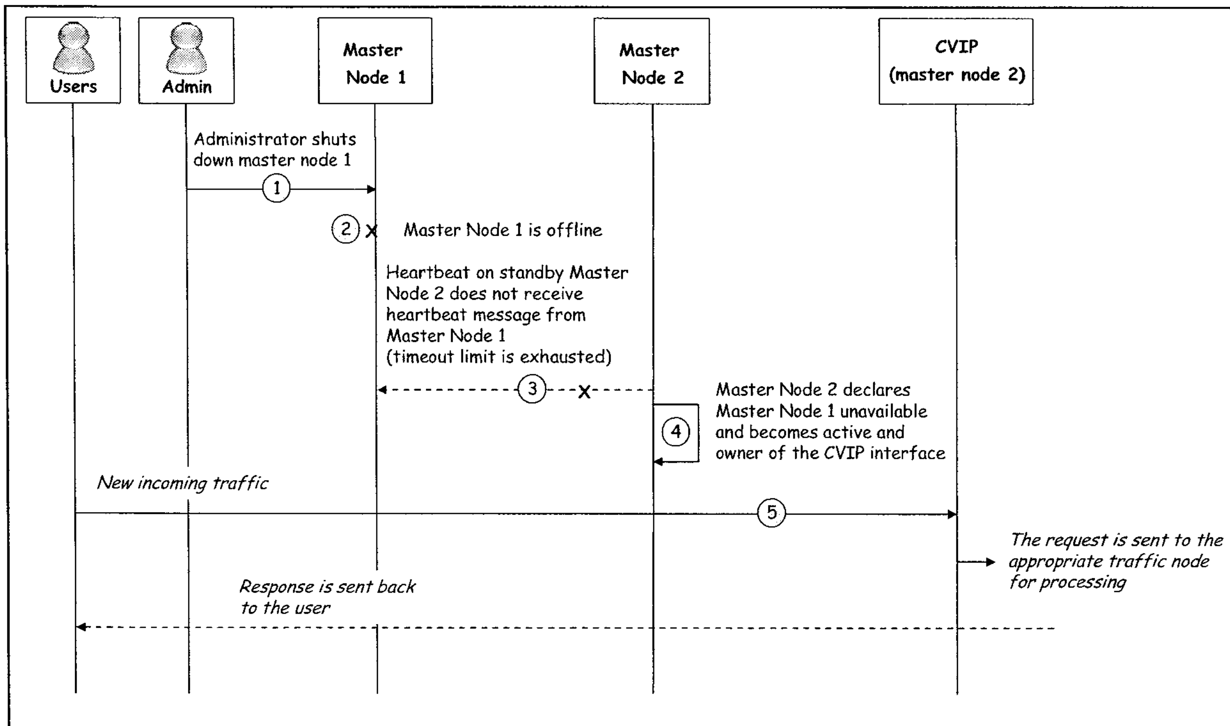


Figure 68: The sequence diagram of upgrading the hardware on a master node

3.24.7 Master Node Becomes Unavailable

Master nodes supervise the availability of each other using heartbeat, which allows the detection of a master node failure within a delay of 200 ms. One common scenario of a failure is when a master node becomes unavailable because of a hardware problem or an operating system crash. It is critical to have a mechanism in place to deal with such a challenge.

Figure 69 illustrates the sequence of events when master node 1 becomes unavailable. When the master node 1 becomes unavailable (1), the heartbeat instance running on the master node 2 does not receive the keep-alive message from the master node 1 and after a configurable timeout of 100 ms (2), master node 2 become the active master node (3) and owner of the cluster virtual interface (4)(5).

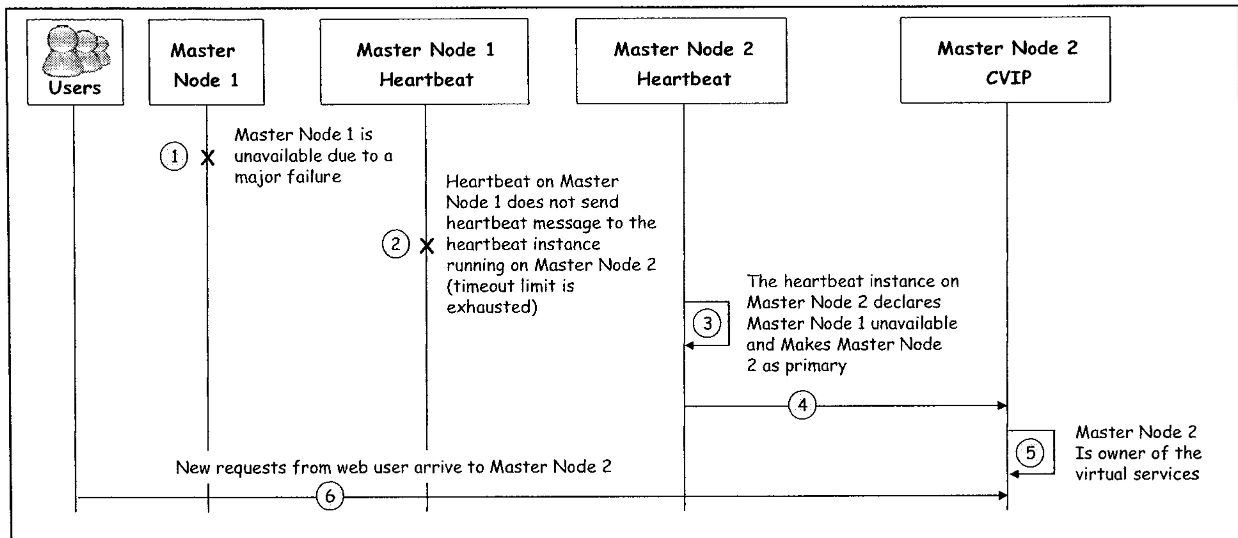


Figure 69: The sequence diagram of a master node becoming unavailable

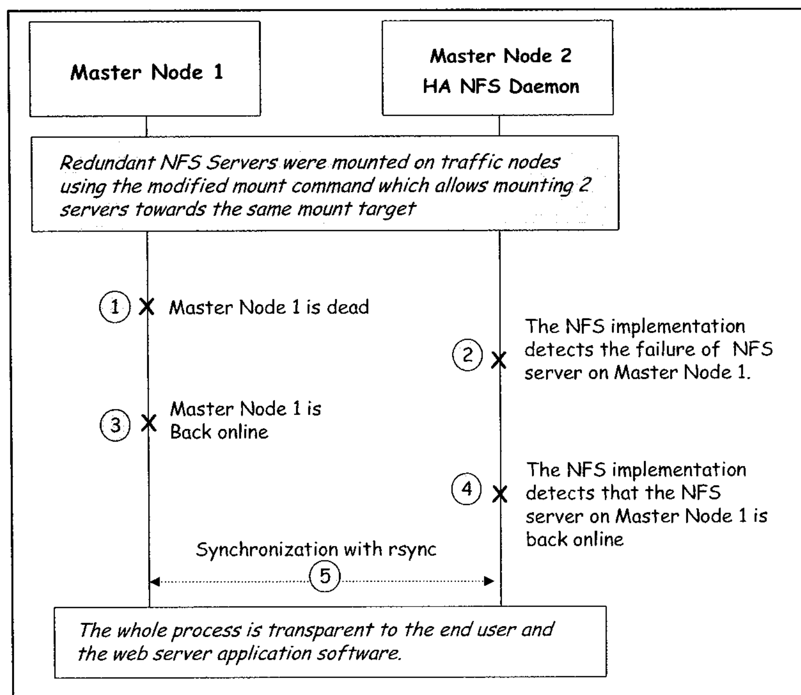


Figure 70: The NFS synchronization occurs when a master node becomes unavailable

3.24.8 Traffic Node becomes Unavailable

Traffic nodes face problems and can become unavailable to serve incoming traffic. The HAS cluster architecture overcomes this challenge by supporting node level redundancy.

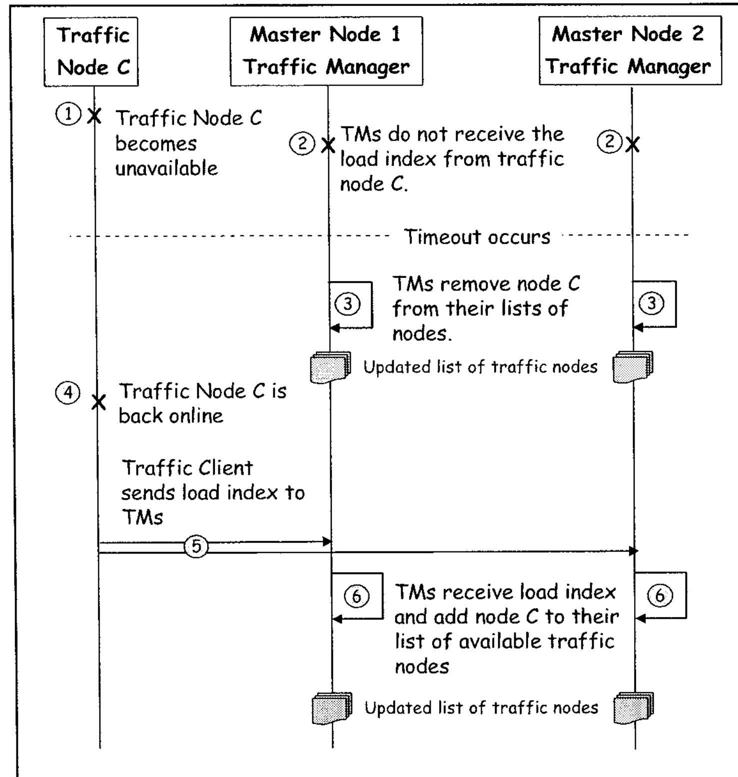


Figure 71: The sequence diagram of a traffic node becoming unavailable

Figure 71 illustrates the scenario when a traffic node becomes unavailable. When a traffic node becomes unavailable (1), the traffic client daemon (running on that node) becomes unavailable and does not report the load index to the master nodes. As a result, the traffic manager daemons do not receive the load index from the traffic node (2). After a timeout, the traffic managers remove the traffic node from their list of available traffic nodes (3). However, if the traffic nodes becomes available again (4), the traffic client daemon reports the load index to the traffic manager running on the master node (5). When the traffic manager receives the load index from the traffic node, it is an indication that the node is up and ready to provide service. The traffic manager then adds the traffic node (6) to the list of available traffic nodes. A traffic node is declared unavailable if it does not send its load statistics to the master nodes within a specific configurable time.

Figure 72 illustrates a traffic node losing network connectivity. The scenario assumes that traffic node C lost network connectivity, and as a result, it is not a member of the HAS cluster.

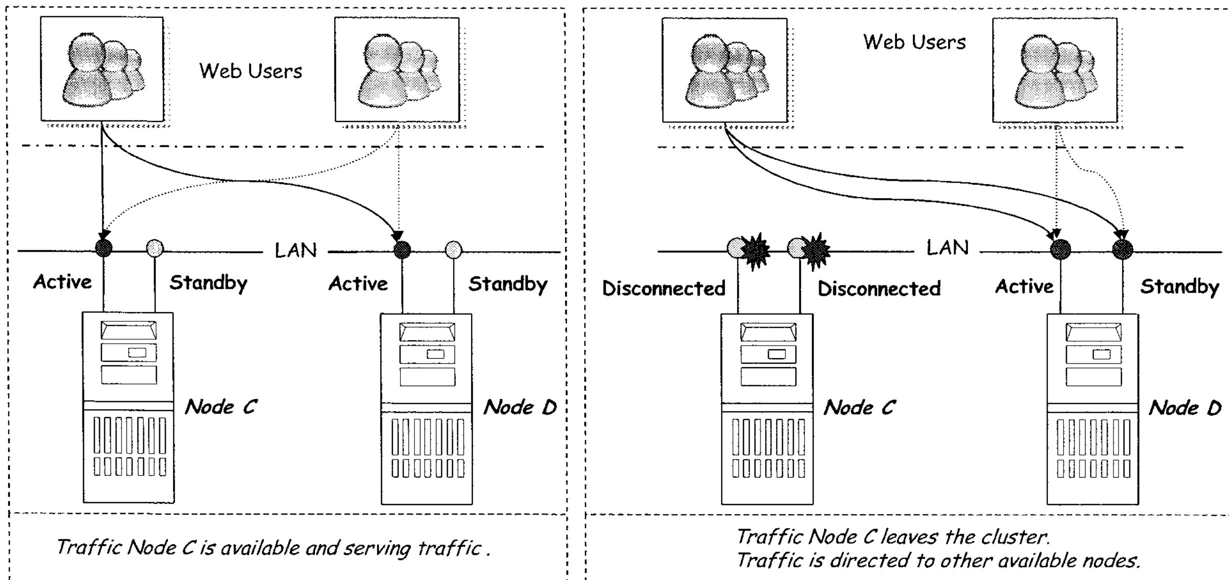


Figure 72: The scenario assumes that node C has lost network connectivity

3.24.9 Ethernet Port becomes Unavailable

All cluster nodes are equipped with two Ethernet cards. The Ethernet redundancy daemon, running on each cluster node, is responsible for detecting when an Ethernet interface becomes unavailable and activating the second Ethernet interface.

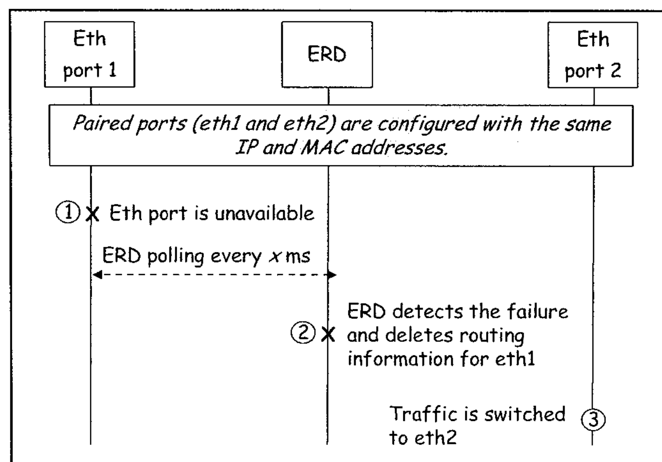


Figure 73: The scenario of an Ethernet port becoming unavailable

Figure 73 illustrates this scenario. When the Ethernet port 1 becomes unavailable (1), the Ethernet redundancy daemon detects the failure after a timeout (2) and activates Ethernet port 2 with the same MAC address and IP address as Ethernet port 1 (3). From this point further, all communication goes through Ethernet port 2.

3.24.10 Traffic Node Leaving the Cluster

When a traffic node does not report its load index to the traffic managers, the later remove the traffic node from their list of available traffic nodes, and as a result, the node stops receiving traffic.

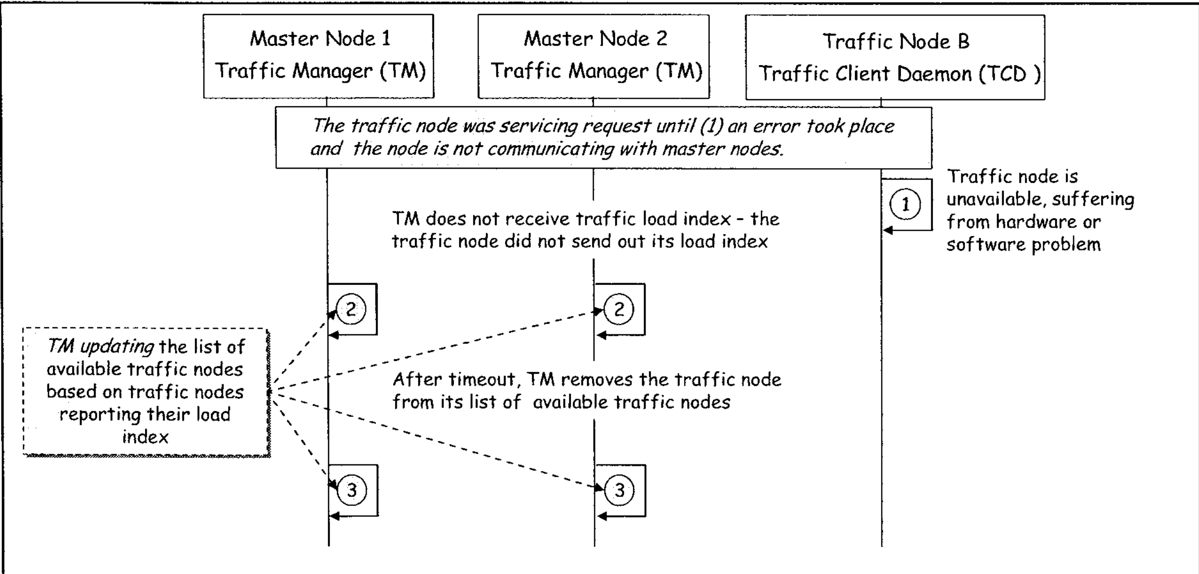


Figure 74: The sequence diagram of a traffic node leaving the HAS cluster

Figure 74 illustrates how a traffic node leaves the cluster. The traffic managers stop receiving messages from the traffic node who must report its load index (1)(2). After a defined timeout, the traffic managers remove the node from the list of available traffic nodes (3). The scenario of a traffic node leaving the HAS cluster is similar to the scenario *Traffic Node Becomes Unavailable* presented in Section 3.24.8.

3.24.11 Application Server Process Dies

The availability of service depends on the availability of the application process (in our cases, the Apache web server). If the application crashes and becomes unavailable, we need a mechanism to detect the failure and restart the application. This is important because a master node cannot detect such a failure

and as a result, the load balancer continues to forward incoming traffic to a traffic node that hosts a failing application.

Our approach to overcome this challenge requires two software modules: the cron operating system facility and the LDirectord module. The *cron* utility is a UNIX system daemon that executes commands or scripts as scheduled by the system administrator. As such, the operating system monitors the application and it re-starts it when it crashes. However, if the application crashes in between cron checks, then the load balancer continues to forward requests to the web server running on the traffic node. New and ongoing requests fail because the web server is down.

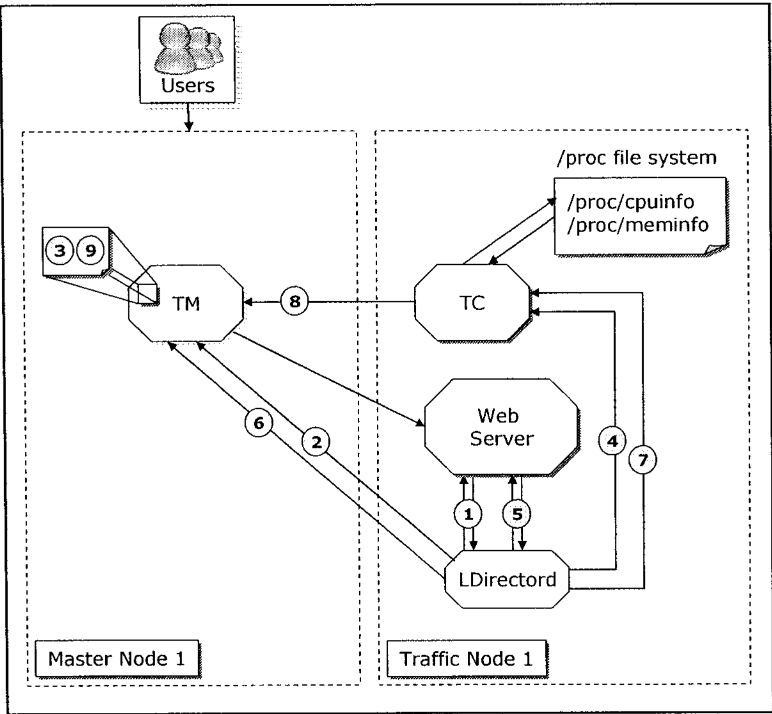


Figure 75: The LDirectord restarting an application process

Figure 75 illustrates our initial approach into addressing this challenge. The LDirectord performs an application check by making an HTTP request to the application and checking the result (1). The check is performed every x milliseconds (x is a configurable parameter), and ensures that we can open a HTTP connection to the web server running on the traffic node. If the result is positive, then the application is up and running and no action is required. However, if the result of the check is negative, then the web server did not respond as expected. In this case, the LDirectord connects to the traffic manager and reports that the application running on the traffic node is not available. Otherwise, the traffic manager continues to

have the traffic node among its list of available traffic nodes. The LDirectord delivers the pair of (traffic_node_IP, load_index) to the traffic manager, with the load_index = 0 (2). A load_index = 0 indicates to the traffic manager that the node or application is not responsive. Next, the traffic manager removes the specific traffic nodes from its of available traffic nodes (3). The LDirectord also ensures that the traffic client does not update the load_index on the traffic manager while the application is still unresponsive. Therefore, the LDirectord sets the load_index_report_flag to 0 (4). When the load_index_report_flag = 0, the traffic client stops reporting its load to the traffic managers. On the next loop cycle (5), the LDirectord checks if the application is still unresponsive. If the application is still not available, then no action is required from LDirectord. If the application check returns positive, the LDirectord connects to the traffic manager and delivers the pair of (traffic_node_IP, load_index), with the load_index = 1. The LDirectord also resets the load_index_report_flag to 1 (7). When the load_index_report_flag = 1, the traffic client resumes reporting its load to the traffic manager. The traffic client reports the new load_index (8). The traffic manager adds the traffic node to its list of available traffic nodes (9).

3.24.12 Network Becomes Unavailable

In the event that one network becomes unavailable, the HAS cluster needs to survive such a failure and switch traffic to the redundant available network.

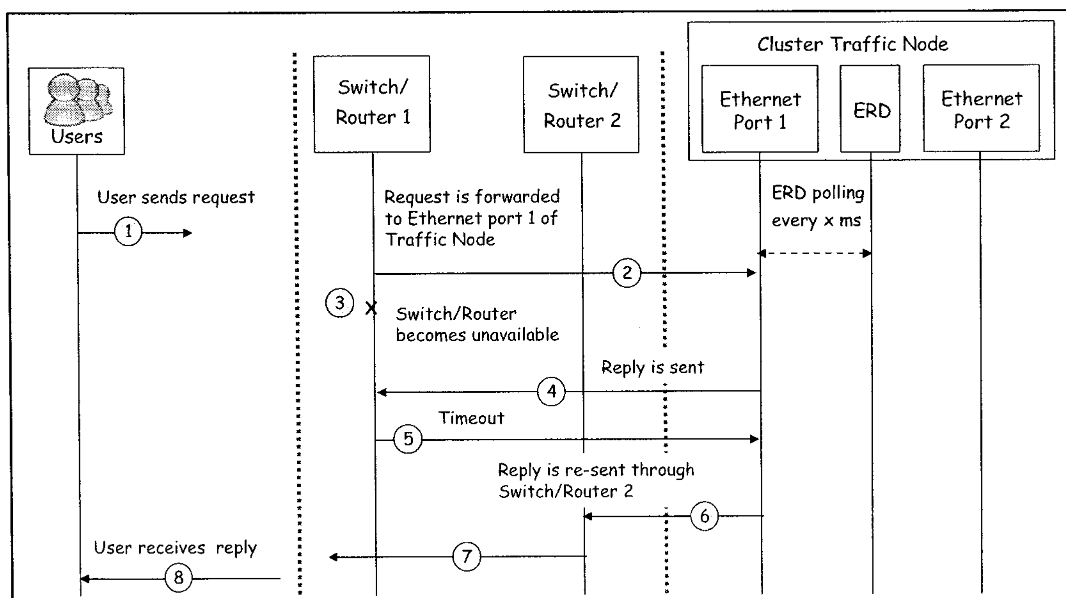


Figure 76: The network becomes unavailable

Figure 76 illustrates this scenario. When the router becomes unavailable (3), Ethernet port 1 gets a reply with a timeout (5). At this point, the Ethernet port 1 uses its secondary route through router 2. We can use the heartbeat mechanism to monitor the availability of routers. However, since routers are outside our scope, we do not pursue how to use heartbeat to discover and recover from router failures.

3.25 Network Configuration with IPv6

By default, cluster administrators configure the network setting on all cluster nodes with IPv4 through either static configuration or using a DHCP server. However, with clusters consisting of tens and hundreds of nodes, these methods are labor intensive and prone to many errors. Designers of network protocols recognize the difficulty of installing and configuring TCP/IP networks. Over the years, they have come up with solutions to overcome these pitfalls. Their latest outcome is a newly designed IP protocol version, IPv6. One of IPv6's useful features is its auto-configuration ability. It does not require a stateful configuration protocol such as DHCP. Hosts, in our case cluster nodes, can use router discovery to determine the addresses of routers and other configuration parameters. The router advertisement message also includes an indication of whether the host should use a stateful address configuration protocol.

There are two types of auto-configuration. *Stateless configuration* requires the receipt of router advertisement messages. These messages include stateless address prefixes and preclude the use of a stateful address configuration protocol. *Stateful configuration* uses a stateful address configuration protocol, such as DHCPv6, to obtain addresses and other configuration options. A host uses stateful address configuration when it receives router advertisement messages that do not include address prefixes and require that the host use a stateful address configuration protocol. A host also uses a stateful address configuration protocol when there are no routers present on the local link. By default, an IPv6 host can configure a link-local address for each interface. The main idea behind IPv6 autoconfiguration is the ability of a host to auto-configure its network setting without manual intervention.

Autoconfiguration requires routers of the local network to run a program that answers the autoconfiguration requests of the hosts. The radvd (Router ADVertisement Daemon) provides these functionalities. This daemon listens to router solicitations and answers with router advertisement.

Figure 77 illustrates the process of auto-configuration. This scenario assumes that the router advertisement daemon is started on at least one master node, and that cluster nodes support the IPv6 protocol at the operating system level, including its auto-configuration feature. The node starts (1). As the

node is booting, it generates its link local address (2). The node sends a router solicitation message (3). The router advertisement daemon receives the router solicitation message from the cluster node (4); it replies with the router advertisement, specifying subnet prefix, lifetimes, default router, and all other configuration parameters. Based on the received information, the cluster node generates its IP address (5). The last step is when the cluster node verifies the usability of the address by performing the Duplicate Address Detection process. As a result, the cluster node has now fully configured its Ethernet interfaces for IPv6.

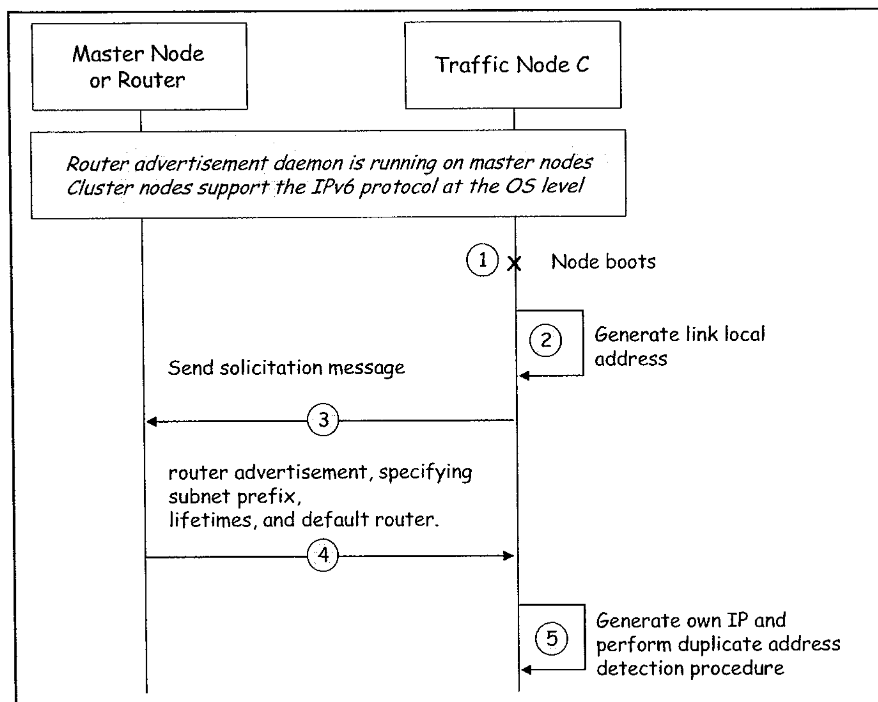


Figure 77: The sequence diagram of the IPv6 autoconfiguration process

Chapter 4
Evaluation of the HAS Architecture

This chapter presents the evaluation of the HAS architecture, which includes the demonstration of performance and scalability, the testing of the failover mechanisms, and the modeling and simulation of the architecture availability.

4.1 Benchmarking Hardware Environment

We used web server benchmarking to evaluate the performance and scalability of the HAS architecture proof-of-concept. We used 31 Intel PIII machines, called web client machines, to generate web traffic to the HAS architecture proof-of-concept. Each of the web client machines is equipped with 512 MB of RAM and runs Windows NT. In addition, the benchmarking environment requires a controller machine that is responsible for collecting and compiling the test results from all the web client machines. The controller machine has a Pentium IV processors running at 500 MHz with 512 MB of RAM.

4.2 Benchmarking Tool and Workload

The web client machines run a copy of the WebBench [77] software, a freeware benchmarking tool that simulates web browsers. When a web client receives a response from the web server, WebBench records the information associated with the response and immediately sends another request to the server.

WebBench uses PC clients to send requests for standardized workloads to the web server. The workload is a combination of static files and dynamic executables that run in order to produce the data the server returns to the client. These client machines simulate web browsers. When the server replies to a client request, the client records information such as how long the server took and how much data it returned and then sends a new request. When the test ends, WebBench calculates two overall server scores, requests per second and throughput in bytes per second, as well as individual client scores. WebBench maintains at run-time all the transaction information and uses this information to compute the final metrics presented when the tests are completed.

Figure 78 presents the WebBench architecture. WebBench runs on a server running the controller program and one or more servers, each running the client program. The controller provides means to set up, start, stop, and monitor the WebBench tests. It is also responsible for gathering and analyzing the data

reported from the clients. The web client machines execute the WebBench tests and send requests to the web server.

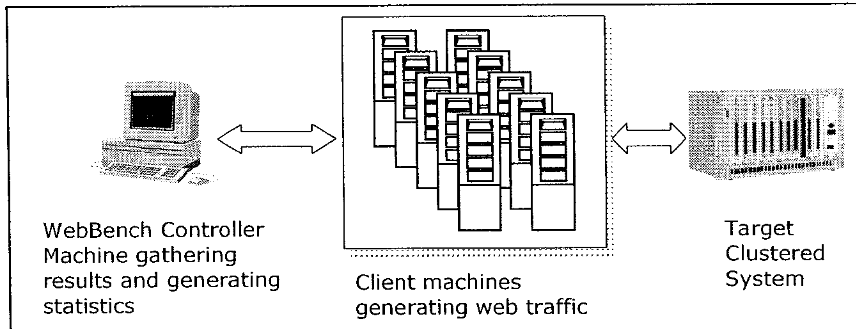


Figure 78: The architecture of the WebBench benchmarking tool

WebBench supports up to 1,000 web clients generating traffic to a target web server. WebBench stresses a web server by using a number of test systems to request URLs. We can configure each WebBench test system to use multiple threads to make simultaneous web server requests. By using multiple threads per test system, it is possible to generate a large load on a web server to stress it to its limit with a reasonable number of test systems. Each WebBench test thread sends an HTTP request to the web server and waits for the reply. When the reply comes back, the test thread immediately makes a new HTTP request. WebBench makes peak performance measurements that illustrate the limitations of a web server platform. Figure 79 illustrates the configuration window of the test suites and client mixes. In this specific test case, we are using the static test suite – `static.tst`. The workload tree provided by WebBench contains the test files the WebBench client access when we execute a test suite. WebBench workload tree is the result of studying real-world sites such as Microsoft, USA Today, and the Internet Movie Database. The tree uses multiple directories and different directory depths. It contains over 6,200 static pages and executable test files. The WebBench provides static (`static.tst`) and dynamic test suites (`wbssl.tst`).

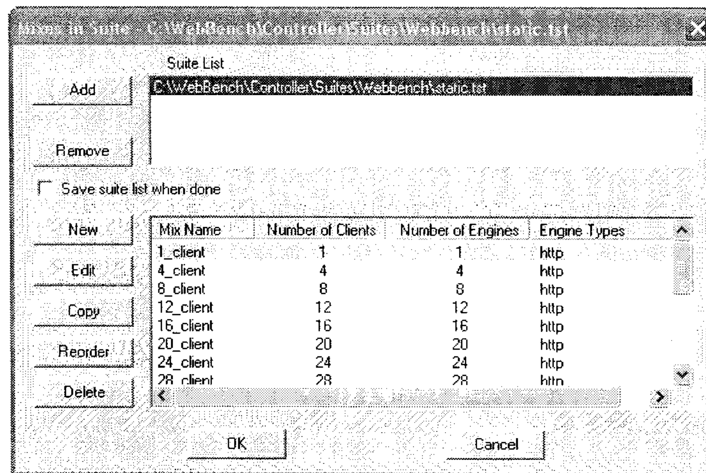


Figure 79: Adding mixes to the WebBench test

WebBench keeps at run-time all the transaction information and uses this information to compute the final metrics presented when the tests are completed. The standard test suites of WebBench begin with one client and add clients incrementally until they reach a maximum of 60 clients (per single client machine). WebBench provides numerous standardized test suites. For our testing purposes, we executed a mix of the `static.tst` (90%) and `wbssl.tst` (10%) tests.

The duration of each test is a configuration parameter. In our testing, to ensure that we are receiving the steady state performance, we benchmarked the cluster starting from two hours and a half with 2 traffic nodes up to over seven hours for the 16 traffic nodes cluster.

The distribution of the WebBench requests to the cluster follows a well-defined procedure. The WebBench tool maintains a file that contains all the documents available for serving from the web server. This list is the workload tree and contains the test files the WebBench clients request when executing a test suite. The tree uses multiple directories and different directory depths. Each WebBench client generates requests following this list sequentially. The list contains over 6200 entries to documents stored on the web server.

Figure 80 illustrates the various configuration parameters that can be tuned when performing a benchmarking test with WebBench. After setting all the parameters and choosing the workload, we start the WebBench controller and the WebBench clients.

Figure 81 is a screen capture from the WebBench controller PC that shows 379 connected WebBench clients from the client machines that are ready to generate traffic to the web cluster.

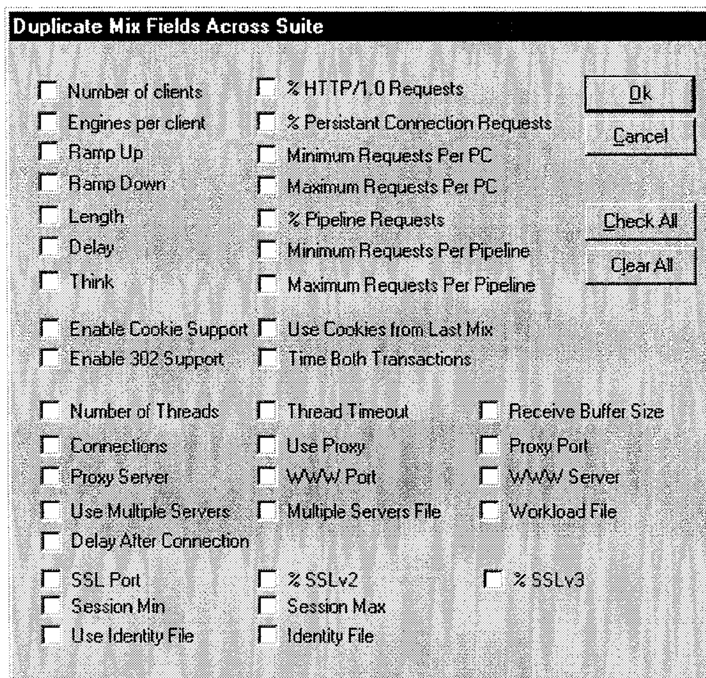


Figure 80: Configurable WebBench parameters

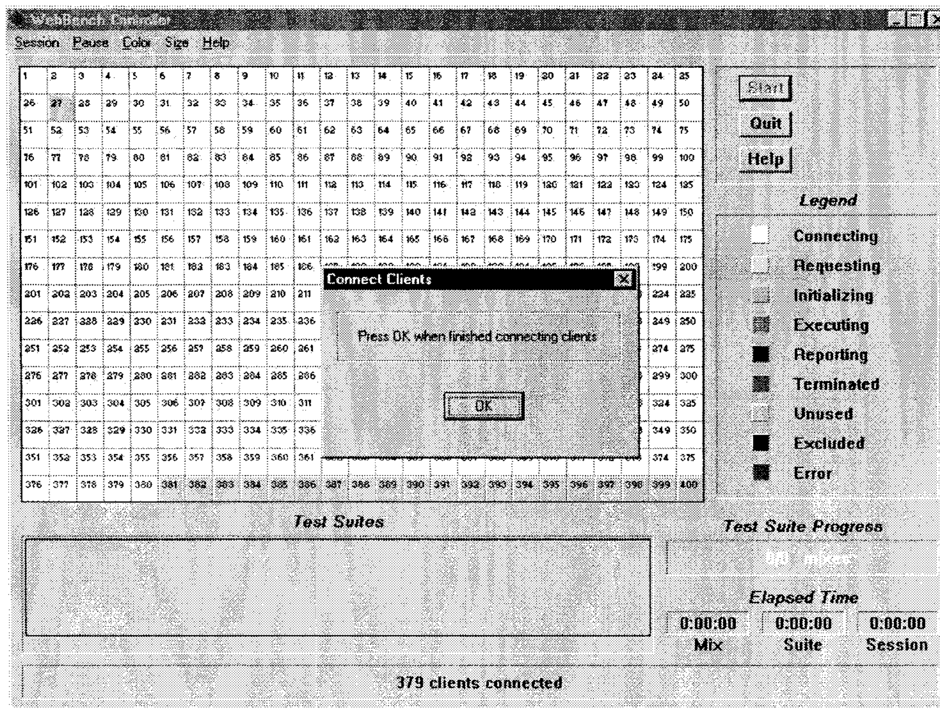


Figure 81: A screen capture of the WebBench software showing 379 connected clients

In [24], we present the details of setting up the software and hardware benchmarking environment.

4.3 Benchmarking Network Environment

The benchmarking tests took place at the Ericsson Research lab in Montréal, Canada. Although the lab connects to the Ericsson Intranet, our LAN segment is isolated from the rest of the Ericsson network and therefore our measurement conditions are under well-defined control.

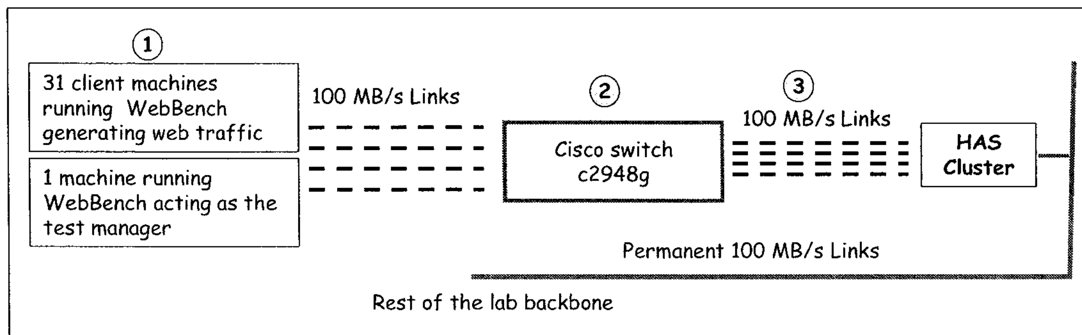


Figure 82: The network setup inside the benchmarking lab

Figure 82 illustrates the network setup in the lab. The client computers run WebBench to generate web traffic with one computer running WebBench as the test manager. These computers connect to a fiber capable Cisco switch (2) through 100 MB/s links. All the nodes in the HAS cluster connect to the Cisco switch (3) through a 100 MB/s links.

4.4 Determining Baseline Performance of a Standalone Traffic Node

The goal of this test is to establish the baseline performance of a single web server node, and to allow us to determine the performance limitation of a single node. It consists of generating web traffic to a single standalone server running the Apache web server software (version 2.0.35), and requesting documents from the NFS server running on the network segment.

Table 12 presents the results of the benchmark with a single server node. It illustrates the number of WebBench clients generating web traffic, the number of requests per second the servers has successfully completed, the throughput, and the number of unsuccessful requests due to errors. WebBench generates this table automatically as it collects the results of the benchmarking test.

The main lesson from this benchmark is that the average capacity for a standalone server is 1,032 requests per second. If the server receives requests over its baseline capacity, it becomes unable to respond to all of them. Hence, the number of failed requests (Table 12) as soon as WebBench is generating traffic with the

16_clients mix. Apache stops responding to incoming requests when we reach 16 simultaneous WebBench clients generating over 1,300 requests per second.

Number of Clients	Requests Per Second	Throughput (Bytes/Sec)	Throughput (KBytes/Sec)	Errors (Connection + Transfer)
1_client	143	825609	806	0
4_clients	567	3249304	3173	0
8_clients	917	5240358	5118	0
12_clients	1012	5769305	5634	0
16_clients	1036	5924397	5786	212
20_clients	1036	6044917	5903	456
24_clients	1038	6058103	5916	692
28_clients	1040	6063940	5922	932
32_clients	1037	6046431	5905	1012
36_clients	1037	6052267	5910	1252
40_clients	1037	6049699	5908	1484
44_clients	1029	6008823	5868	1736
48_clients	1032	6020502	5879	1983
52_clients	1033	6032181	5891	2237

Table 12: The performance results of one standalone processor

In Figure 83, we plot the results from Table 12, the number of clients versus the number of requests per second. We notice that as we reach 16 clients, Apache is unable to process additional incoming requests and the scalability curve levels-off.

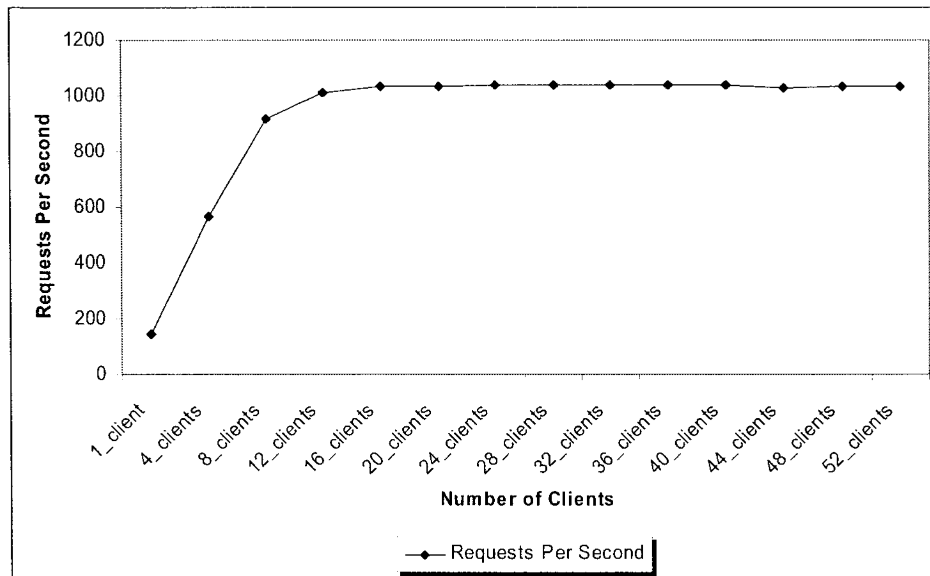


Figure 83: The results of benchmarking a standalone processor (requests per second)

With this exercise, we conclude that the maximum number of requests per second we can achieve with a single process is 1,032. We use this number to measure how our cluster scales as we add more processors. Figure 84 presents the throughput achieved with one processor. The maximum throughput possible with a single processor averages around 5,800 KB/s.

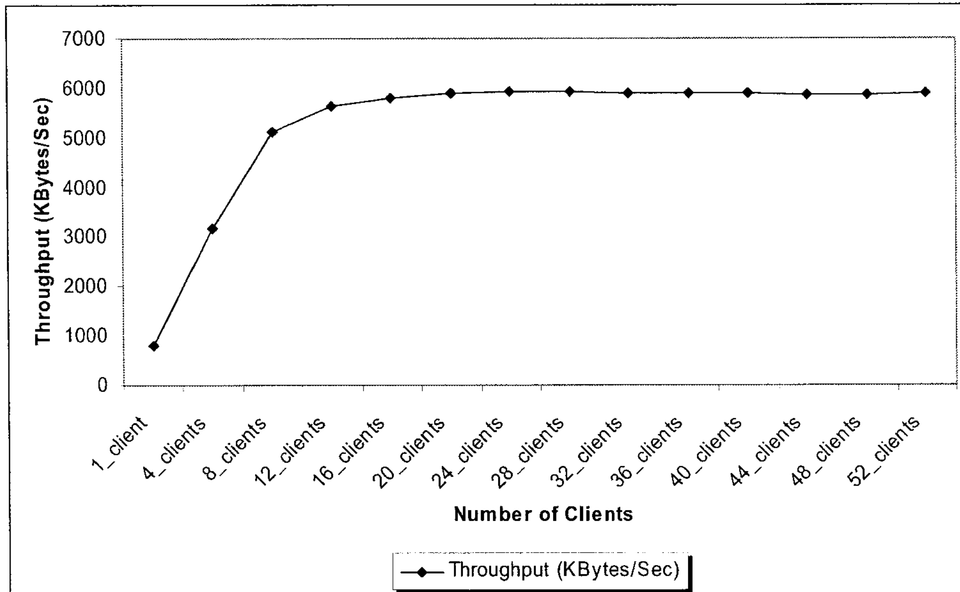


Figure 84: The throughput of a standalone processor expressed in KB/s

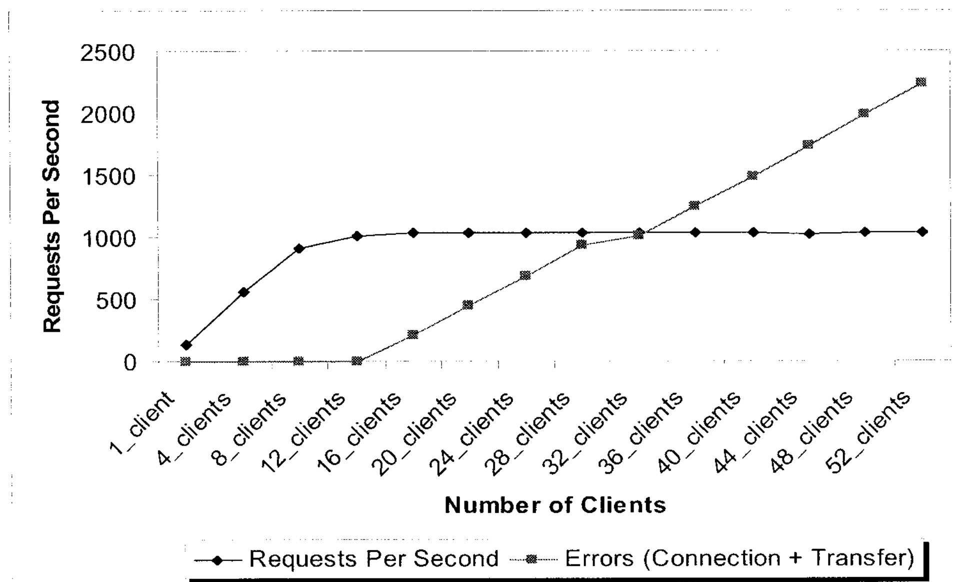


Figure 85: The number of failed requests per second on a standalone processor

In addition to the presented results, WebBench also provides statistics about failed requests. Figure 85 illustrates the curve of successful requests per second combined and the curve of failed requests per second. As we increase the number of clients generating traffic to the processor, the number of failed requests increases. Based on the benchmarks with a single node, we can draw two main conclusions. The first is that a single processor can process up to one thousand requests per second before it reaches its threshold. The second conclusion is that after reaching the threshold, the application server stops responding to incoming requests.

4.5 Benchmarking the HAS Architecture Proof-of-Concept

Figure 86 illustrates the four benchmarked configurations of the HAS architecture proof-of-concept: Test-1, Test-2, Test-3, and Test-4.

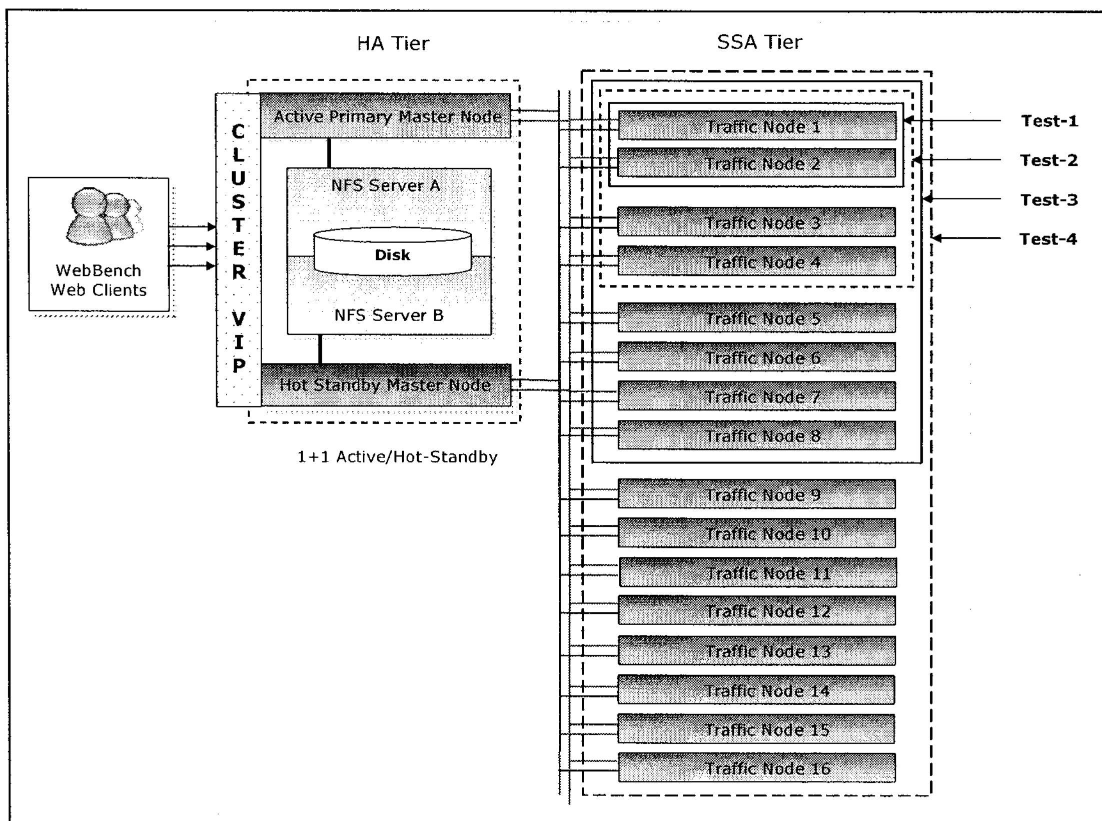


Figure 86: The four benchmarked configurations

The duration of the tests varies, starting from a three hours and a half for **Test-1** up to over seven hours for **Test-4.**

- **Test-1:** In this configuration, the HAS cluster consists of two master nodes and two traffic nodes. The master nodes follow the active/standby redundancy model. All traffic nodes are active. This is the minimal deployment of HAS cluster.
- **Test-2:** In this configuration, the HAS cluster consists of two master nodes and four traffic nodes. The master nodes follow the active/standby redundancy model. All traffic nodes are active. This configuration has double the number of traffic nodes than Test-1.
- **Test-3:** In this configuration, the HAS cluster consists of two master nodes and eight traffic nodes. The master nodes follow the active/standby redundancy model. All traffic nodes are active. This configuration has double the number of traffic nodes than Test-2.
- **Test-4:** In this configuration, the HAS cluster consists of two master nodes and 16 traffic nodes. The master nodes follow the active/standby redundancy model. All traffic nodes are active. This configuration has double the number of traffic nodes than Test-3.

4.6 Test-4: Experiments with an 18-node HAS Cluster

In this test, the HAS cluster consists of 18 nodes; each node has a Pentium III processor and 512 MB of RAM. The traffic distribution mechanism was the HAS mechanism. We configured the HAS cluster prototype as follows:

- Two master nodes, running in active/standby mode, provide an entry point to the cluster through the CVIP, and provide storage service through the HA NFS implementation.
- Sixteen traffic nodes each run a copy of the Apache web server version 2.0.35.

This test is the largest we conducted with 32 machines in the benchmarking environment, 31 of which generate traffic, in addition to one machine that administers the test, collects and compile the results.

Figure 87 presents the number of successful transactions per second achieved with the 18 processor HAS cluster. In this test, the HAS cluster achieved an average of 16,001 requests per second, an average of 1,000 requests per second per traffic node in the HAS cluster.

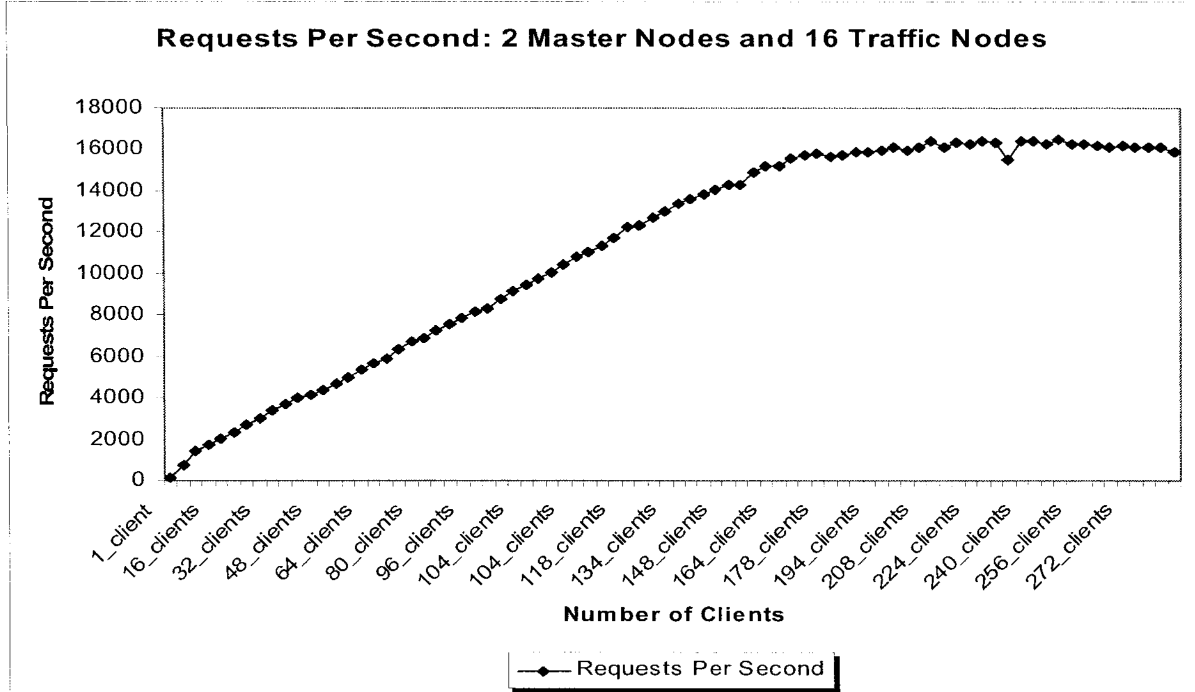


Figure 87: The results of benchmarking an 18 nodes HAS cluster

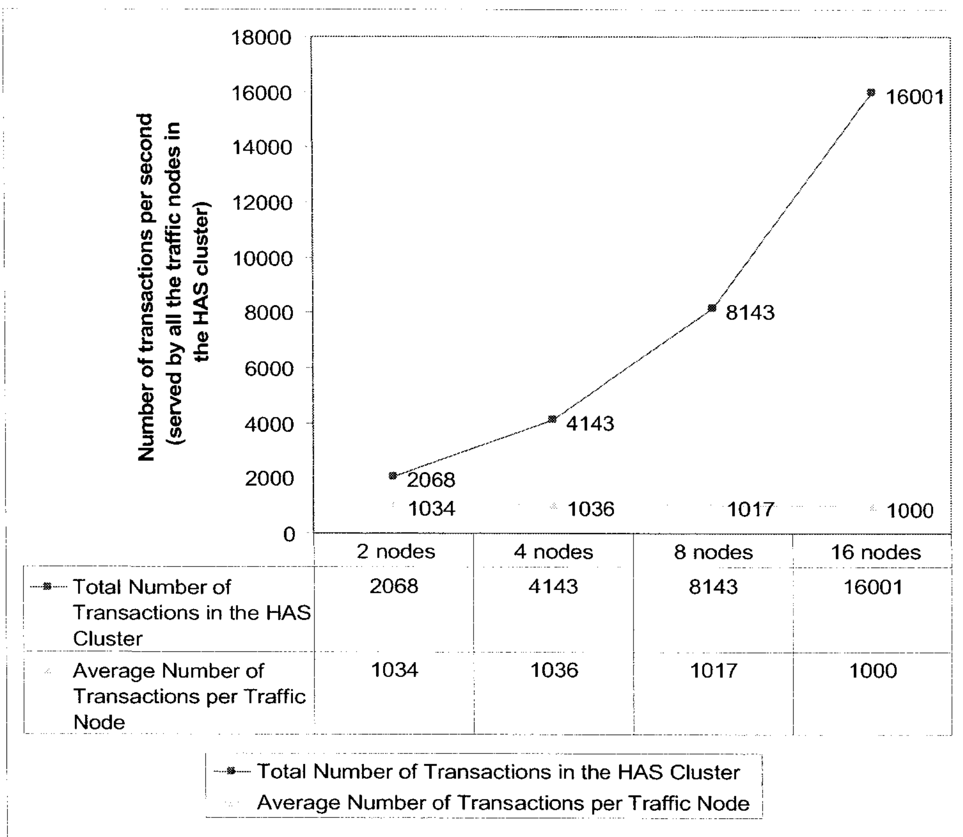
4.7 Overall Results and Discussion

Table 13 presents the summary of the benchmarking results collected from Test-1, Test-2, Test-3, and Test-4. For each test, we recorded the average number of requests per second that each HAS cluster configuration supported. When we divide this number by the number of traffic processors, we get the average number of requests per second that each traffic node can process in each configuration.

Test-ID	Number of Traffic Nodes	Average Successful Transactions per Second	Average Successful Transactions per Traffic Node per Second
Test-1	2	2068	1034
Test-2	4	4143	1036
Test-3	8	8143	1017
Test-4	16	16001	1000

Table 13: The summary of the benchmarking results of the HAS architecture prototype

Figure 88 illustrates the scalability of the prototyped HAS cluster architectures starting with two traffic processors in Test-1 and up to 16 traffic processors in Test-4. The HAS cluster maintained a 1,000 requests per second per traffic node. As we scaled by adding more traffic nodes to the SSA tier, we lost 3.1% of the baseline performance per traffic node as defined in Section 4.4.



	2 nodes	4 nodes	8 nodes	16 nodes
Total Number of Transactions in the HAS Cluster	2068	4143	8143	16001
Average Number of Transactions per Traffic Node	1034	1036	1017	1000

Figure 88: The results of benchmarking the HAS architecture prototype

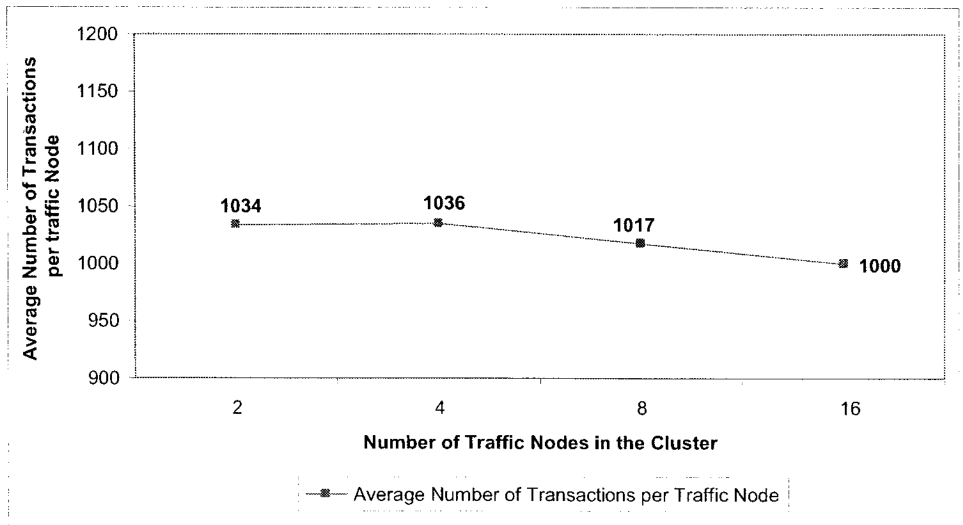


Figure 89: The scalability chart of the HAS architecture prototype

Figure 89 provides a close look at the scalability chart of the HAS cluster. The results demonstrate close-to-linear scalability as we increased the number of traffic nodes in the HAS clusters. The HAS architecture scaled for up to 16 nodes with a 3.1% decrease in the baseline performance.

4.8 Testing of Failover Mechanisms

The goal of testing the failover mechanism is to ensure that the system software works properly and to try to discover every conceivable fault or weakness. To test the failover mechanisms in the HAS architecture proof-of-concept, we provoked several failure scenarios. Our testing strategy relies on provoking common faults in the HAS architecture proof-of-concept, examining how the failures are observed and repaired by the underlying mechanisms, and monitoring their effect on the service provided. The following sub-sections discuss the testing of failover mechanisms in the HAS architecture proof-of-concept and cover the testing the connectivity (Ethernet connection and routers), data availability (redundant NFS server), master node, and traffic node availability.

4.8.1 Experiments with Connectivity Availability

Figure 90 illustrates the experiments to test the failover mechanisms of the HAS architecture.

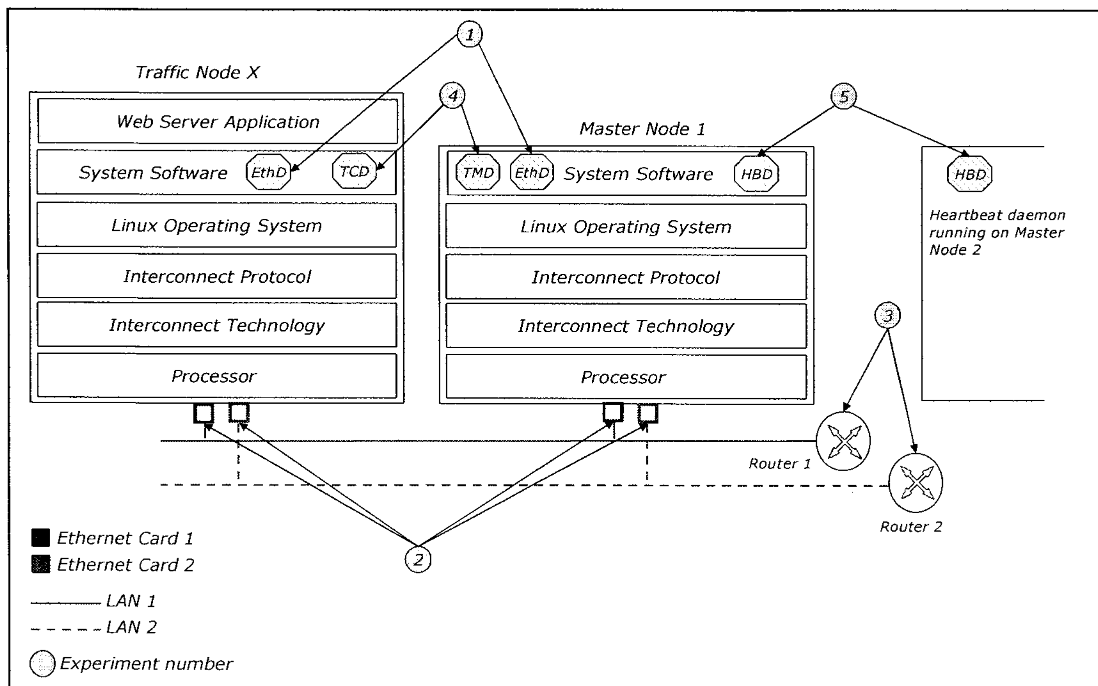


Figure 90: The possible connectivity failure points

The failures tested include Ethernet daemon failure, failure of Ethernet adapter, router failure, discontinued communication between the traffic manager and the traffic client daemon and discontinued communication between the heartbeat instances running on the master nodes. The experiments included provoking a failure to monitor how the HAS cluster reacts to the failure, how the failure affects the service provided, and how the cluster recovers from the failure.

4.8.1.1 Ethernet Daemon Becomes Unavailable

For this type of failure, there is no differentiation if the failure takes place on a master node or a traffic node. If we shutdown the Ethernet daemon on a cluster node (Figure 90 – experiment 1), there is no direct effect on the service provided

The one negative impact that can take place is when the primary Ethernet card fails (Ethernet Card 1), the Ethernet redundancy daemon is not running to detect the failure and to switch to the backup Ethernet card (Ethernet Card 2). As a result, the node becomes isolated from the cluster and the client traffic daemon is not able to connect to the traffic manager. The traffic manager declares the traffic node unavailable and removes it from its list of available traffic nodes.

4.8.1.2 Ethernet Card is Unresponsive

Similar to the use case above, there is no difference for type of failure if the failure occurs on a master node or a traffic node. If the Ethernet card becomes unresponsive and unavailable (Figure 90 – experiment 2), the Ethernet redundancy daemon detects the failure (within a range of 350 ms to 400 ms), considers the Ethernet card out of order, and starts the process of the network adapter swap (Section 3.6.3). As a result, the Ethernet redundancy daemon designates the standby Ethernet card (Ethernet Card 2) as the primary adapter. Section 3.24.9 covers the workings of this scenario. If the unresponsive Ethernet card receives traffic while it is down, or while the transition to the second active Ethernet card is not yet complete, new incoming connections will stall, and ongoing connections are lost.

4.8.1.3 Router Failure

If the router fails (Figure 90 – experiment 3), we consider this failure a network problem and it is beyond the scope of the HAS cluster architecture.

4.8.1.4 Discontinued Communication between the TM and the TC

This test case examines the scenario where we disrupt the communication between the TM and the TC (Figure 90 – experiment 4). As a result, the TM stops receiving load alerts from the TC. After a predefined timeout, the TM removes the traffic node from its list of available traffic nodes and the load balancer stops forwarding traffic to it. When we restore communication between the TM and the TC, the TC starts sending its load messages to the TM. The TM then adds the traffic node to its list of available nodes and the load balancer starts forwarding traffic to it. Section 3.24.8 examines a similar scenario.

4.8.1.5 Discontinued Communication between the Heartbeat Instances

This test case examines what happens when we disconnect the communication between the heartbeat instances running on the master nodes (Figure 90 – experiment 5). This failure scenario depends on whether the disconnected node is the active master node or the standby master node.

If the heartbeat instance running on the active master node becomes unavailable, then the heartbeat instance running on the standby node does not receive the keep-alive message. As a result, the heartbeat instance on the standby node declares the primary master node unavailable, thereby making the standby master node the active master node and owner of the virtual service. The new master node starts receiving traffic within a delay of 200 ms.

If the heartbeat instance running on the standby node becomes unavailable, then the heartbeat instance running on the master node does not receive the keep-alive message. The cluster does not undergo a reconfiguration. However, the HA tier of the HAS cluster becomes vulnerable to SPOF since there is only one master node that is available, and it is in the active state.

4.8.2 Experiments with Data Availability

The availability of data is essential in a clustered environment where data resides on shared storage and multiple clients access it. In the HAS architecture prototype, the data is duplicated and available on two NFS servers through a special implementation of the NFS server code and an updated implementation of the mount program.

Our testing methodology uses a direct approach. Based on Figure 91, two components can affect data availability: the availability of the NFS server daemons and the availability of the master nodes. Master nodes are redundant and the failure of one master node does not affect the availability of data or access to the provided service. The only scenario that could lead to data unavailability is if the NFS server daemons

running on master nodes were to crash. For this purpose, we have implemented redundancy in the NFS server code.

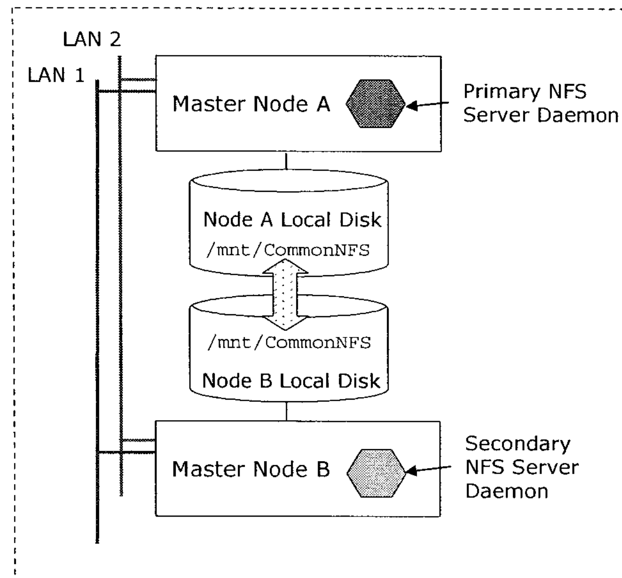


Figure 91: The tested setup for data redundancy

The two test cases we experimented with are shutting down the NFS server daemon on a master node and disconnecting the master node from the network. In both scenarios, there was no interruption to the service provided. Instead, there was a delay ranging between 450 ms and 700 ms to receive the requested document. The delays are captured by the WebBench results.

4.8.3 Experiments with Traffic Node Availability

Traffic nodes follow the N redundancy model. If one node fails and becomes unavailable, the traffic manager removes the node from its list of available nodes and the load balancer forwards traffic to other available traffic nodes. The failures tested include connectivity problems, hardware and software problems, and TC problems.

Connectivity problems: Section 4.8.1 discusses the testing of the connectivity.

Unknown hardware or software problems leading to abnormal node unavailability: If the traffic node goes to a halt state (power off), the traffic manager declares that node as unavailable (by taking it off its list of available traffic nodes) since the traffic client daemon becomes unreachable and unable to report node status to the traffic manager. Section 3.24.8 presents this case scenario.

TC problem: If the traffic client daemon becomes unavailable, because of software or a hardware problem, the daemon does not report the node availability and status to the traffic managers. The traffic node stops receiving traffic. Section 3.24.8 discusses a similar case scenario.

4.8.4 Experiments with Connection Synchronization

To test the connection synchronization mechanism, we open a connection to the virtual service while the master node is active. Then we cause a fail-over to occur by powering down the master node. At this point, the connection stalls. Once the CVIP address is failed over to the standby master node, the connections continues.

Streaming is a useful way to test this, as streaming connections by their nature are open for a long time. Furthermore, it provides intuitive feedback as the video pauses and then continues. It is of note that by increasing the buffer size of the streaming client software, the pause is eliminated.

4.9 Architecture Modeling and Availability Prediction

In 2001, I participated in the Open Cluster Group and proposed the creation of a new working group whose goal is to design and prototype a highly available clustering stack for clusters that run mission critical application. The Open Cluster Group initiated the HA-OSCAR working group, High Availability Open Source Cluster Application Resource, to enhance a Beowulf cluster system for mission critical applications, achieve high availability, and incorporate self-healing mechanisms, failure detection and recovery, automatic failover and failback mechanisms [26].

The author of this thesis is a co-founder of the HA-OSCAR project. The HA-OSCAR architecture is exclusively based on the HAS architecture. In addition to the architecture contribution, several software modules such as the Ethernet Redundancy Daemon and the HA NFS are contributed and used within the HA-OSCAR project.

The HA-OSCAR team at Louisiana Technical University conducted the modeling and simulation activity. The goal is to model and simulate the architecture, its system failure, and recovery, and calculate and predict the availability of the cluster based on a range of predefined assumptions and parameters using the Stochastic Reward Nets (SRN) [54]. My involvement in this activity was as a consultant, and secondary author of the some of the resulting papers [41]. The HA-OSCAR team did all the modeling and simulation work at Louisiana Technical University and they were the primary authors of the resulting publications.

The following sub-sections present an overview of the HA-OSCAR in comparison to a Beowulf cluster, the Stochastic Reward Nets modeling approach, present on the HA-OSCAR modeling and simulation, and the results, and discuss the applicability of the results to the HAS architecture.

4.9.1 HA-OSCAR versus Beowulf Architecture

Beowulf is one approach to clustering COTS components to form a supercomputer. Figure 92 illustrates the architecture of a Beowulf cluster. A Beowulf cluster is a collection of COTS computers that are networked together to run high performance computing application. A Beowulf cluster consists of two node types: head node servers and multiple identical compute nodes. The head node is the single point of entry to the cluster, and is responsible for receiving and distributing user requests to compute nodes via scheduling and queuing software. Compute nodes are dedicated to computation. The Beowulf architecture has several points of failure such as the head node and the communication, and does not offer a highly available solution.

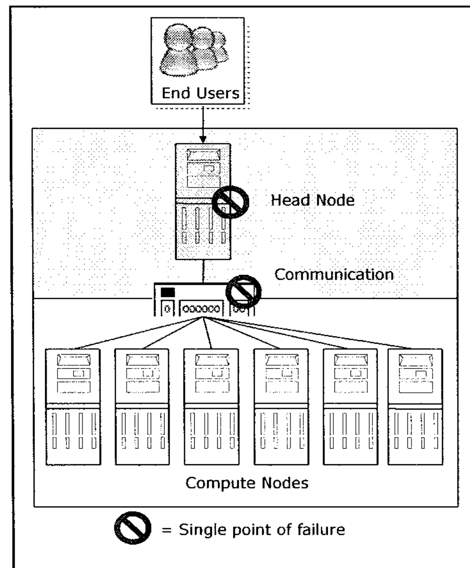


Figure 92: The architecture of a Beowulf cluster

Figure 93 illustrates the HA-OSCAR architecture, which is based on the HAS architecture. The HA-OSCAR architecture deploys duplicate master nodes to offer server redundancy, following the active/standby approach. Furthermore, each node in the HA-OSCAR architecture has two redundant Ethernet cards, eliminating communication as a single point of failure [25]. The HA-OSCAR 1.0 release supports high availability capabilities for Linux Beowulf clusters, and supports the active/standby

redundancy model for the head nodes. It provides a graphical installation wizard and a web-based administration tool to allow the administrator of the HA-OSCAR cluster to create and configure a multi-head Beowulf cluster. In addition, HA-OSCAR includes a default set of monitoring services to ensure that critical services, hardware components, and certain resources are always available at the master node [40].

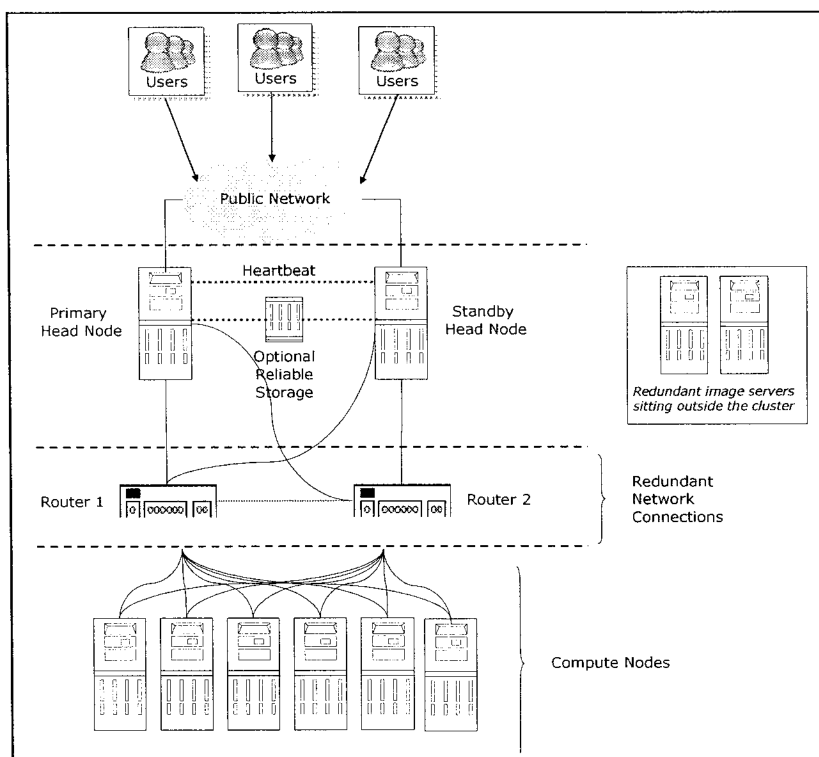


Figure 93: The HA-OSCAR architecture

4.9.2 Stochastic Reward Nets and Stochastic Petri Net Package

Stochastic Reward Nets is a formal method in computer science that is used in the availability evaluation and prediction for complicated systems when the time-dependent behavior is of interest. The Stochastic Petri Net Package (SPNP) [16] is the modeling tool designed for SPN and SRN models [15], used as a modeling tool for performance, dependability (reliability, availability, safety), and performability analysis of complex systems. It allows the specification of SRN models and the computation of steady and transient-state [29]. It uses efficient and numerically stable algorithms to solve input models based on the theory of the SRN.

The SRN models are described in the input language for SPNP called CSPL (C-based SPN Language) which is an extension of the C programming language with additional constructs that facilitate easy description of SPN models. Additionally, if the user does not want to describe his model in CSPL, a graphical user interface is available to specify all the characteristics as well as the parameters of the solution method chosen to solve the model [28].

4.9.3 The SRN Model, Parameters, and Assumptions

The HA-OSCAR project team used the SRN to develop a failure-repair behavior model for the system, and to determine the high availability of the cluster [43][44][45][46].

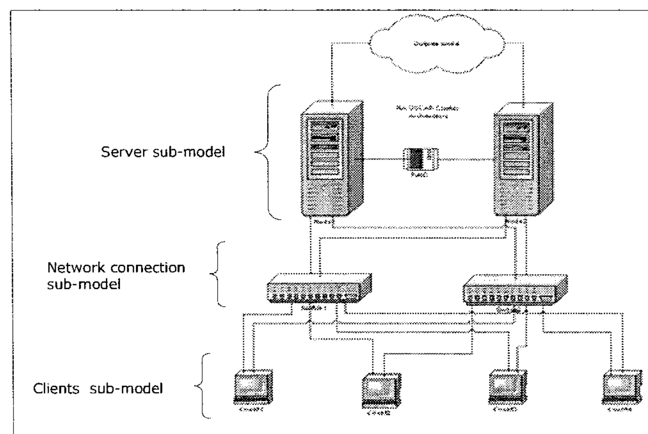


Figure 94: The modeled HA-OSCAR architecture, showing the three sub-models

Figure 94 illustrates the behavior of the HA-OSCAR cluster divided into three sub-models: server, network connection, and client sub-models. The model was input into the SPNP with a set of parameters and assumptions to build and solve the HA-OSCAR SRN model. The SRN model of the HA-OSCAR architecture makes four assumptions. First, there has to be at least one active master node that is functioning properly. Second, at least one LAN that is available for cluster nodes. Third, the "Quorum Value Q", the minimum number of nodes required for the system to keep functioning, must be valid. The last assumption is that the cluster is not undergoing any upgrades or reconfiguration.

Table 14, from [43], presents the input parameter values that are based on studying the overall cluster uptime and the impact of different polling interval sizes in the fault monitoring mechanism.

Input Parameter	Numerical Value
Mean time to primary server failure, $1/\lambda_{ps}$	5,000 hrs.
Mean time to primary server repair, $1/\mu_{psr}$	4 hrs.
Mean time to takeover primary server, $1/\mu_{st}$	30 sec.
Mean time to standby server failure, $1/\lambda_{ss}$	5,000 hrs.
Mean time to standby server repair, $1/\mu_{ssr}$	4 hrs.
Mean time to primary LAN failure, $1/\lambda_{pl}$	10,000 hrs.
Mean time to primary LAN repair, $1/\mu_{plr}$	1 hr.
Mean time to takeover primary LAN, $1/\mu_{lt}$	30 sec.
Mean time to standby LAN failure, $1/\lambda_{sl}$	10,000 hrs.
Mean time to standby LAN repair, $1/\mu_{slr}$	1 hr.
Mean time to client permanent failure, $1/\lambda_p$	2,000 hrs.
Mean time to client intermittent failure, $1/\lambda_i$	1,000 hrs.
Mean time to system reboot, $1/\mu_{srb}$	15 min.
Mean time to client reboot, $1/\mu_{crb}$	5 min.
Mean time to client reconfiguration, $1/\mu_{rc}$	1 min.
Mean time to client repair, $1/\mu_{rp}$	4 hrs.
Permanent failure coverage factor, c_p	0.95
Intermittent failure coverage factor, c_i	0.95

Table 14: Input parameters for the HA-OSCAR model [43]

The availability of the cluster at time *t* is then computed as the expected instantaneous reward rate *E[X(t)]* at time *t* and its general expression is:

$$E[X(t)] = \sum_{k \in \tau} r_k \pi_k(t)$$

where r_k represents the reward rate assigned to state *k* of the SRN, τ is the set of tangible marking, and $\pi_k(t)$ is the probability of being in marking *k* at time *t* [43].

4.9.4 The Results

The first experiment was to compare the availability HA-OSCAR versus Beowulf cluster. The HA-OSCAR experimental and analysis results suggested a significant improvement in availability from the single head and single router Beowulf architecture.

Figure 95, from [41], illustrates the total availability, including planned and unplanned downtime, improvement analysis of the HA-OSCAR architecture versus the Beowulf architecture. The results

demonstrate an availability of 99.9968% for HA-OSCAR compared to 92.387% availability for a Beowulf cluster with a single head node [41]. The results demonstrate that the component redundancy in HA-OSCAR is efficient in improving the cluster system availability. Next, the experimentation focused on the HA-OSCAR architecture. The model parameters were modified to reflect addition nodes in the cluster and the goal is to observe how availability is affected as we add more nodes to the cluster.

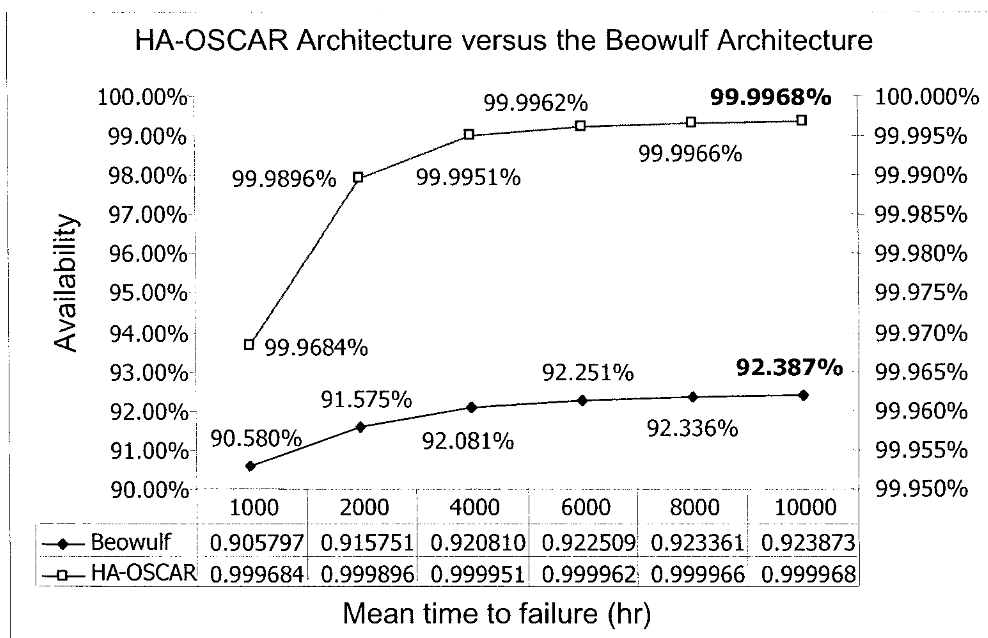


Figure 95: Availability improvement analysis of HA-OSCAR [41]

Table 15, from [43], presents the steady-state availability and the mean cluster down time per year for the different HA-OSCAR configurations. The "System Configuration" is the number of compute nodes in the cluster. The "Quorum Value" is the minimum number of compute nodes that must be present so that the cluster functions properly. The outputs are the "System Availability", and the mean cluster down time.

Table 15 illustrates that the system availabilities for the various configurations, starting from four up to 16 nodes in the cluster, fall within a small range of difference. After introducing the quorum voting mechanism in the client sub-model, the system availability is not sensitive to the change in the number of compute nodes. When we add more compute nodes to the cluster to improve the performance, the availability of the system almost remains unchanged.

System Configuration	Quorum Value	System Availability	Mean cluster down time
4	3	0.999933475091	34.9654921704
6	4	0.999933335485	35.0388690840
8	5	0.999933335205	35.0390162520
16	9	0.999933335204	35.0390167776

Table 15: System availability for different configurations [43]

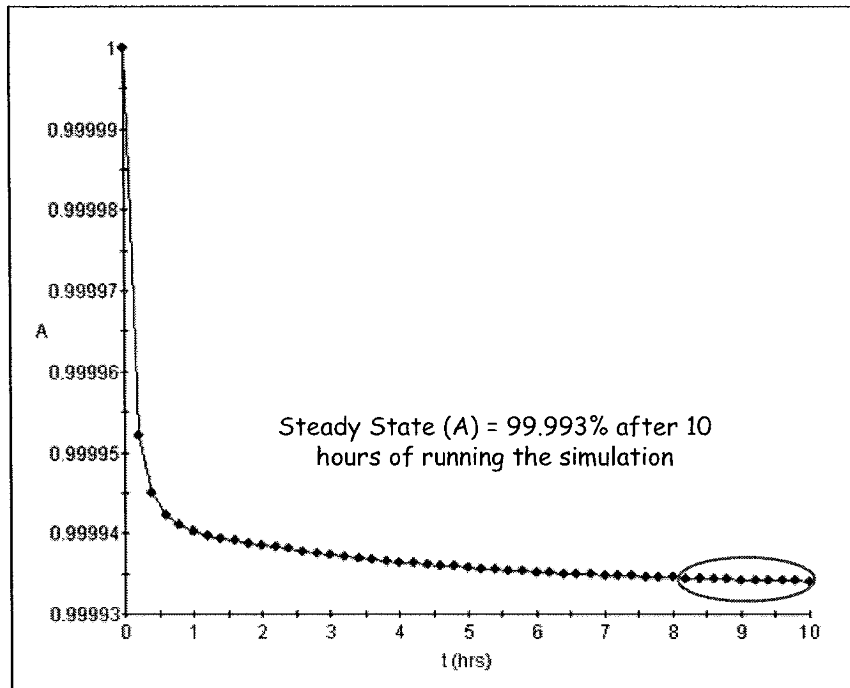


Figure 96: Steady State Availability of 99.993% [43]

Figure 96, from [43], illustrates the steady state availability of the system after 10 hours of running the simulation. The steady state availability is the stabilizing point where the system's availability becomes a constant value.

4.9.5 Discussion

The modeling and simulation of the HA-OSCAR architecture demonstrate that the architecture offers over four nines availability. The results of the HA-OSCAR availability modeling and simulation are directly applicable to the HAS architecture. HA-OSCAR and the HAS share the same architecture. In fact, the HA-OSCAR uses the HAS architecture as the base and contributes specialized software modules targeted for HPC applications. Furthermore, the modeling and simulation of the architecture

are independent from the workload of applications running on these architectures. In the modeling and simulation, the SPN model parameters describe the parameters of the architecture which are shared between HAS and HA-OSCAR, and they are independent of the applications running on these architecture.

Chapter 5
Contributions, Future Work, and Conclusion

5.1 Contributions – The HAS Architecture

The initial goal of this dissertation was to demonstrate a highly available and scalable web cluster architecture. The three initial set goals for the HAS architecture, scalability, performance, and availability, have been demonstrated. To the best of our knowledge, this work contributes the first highly available and scalable architecture for web server clusters that demonstrates close to linear scaling for up to 16 nodes, maintains over 96% of baseline performance, and supports over 99.99% availability.

The HAS architecture follows the building block approach where software modules can be re-used in other environments, they can evolve independently without architecture redesign, and they are easier to test and experiment with. The HAS architecture is based on loosely coupled nodes that are interconnected through high-speed network. The loosely coupled cluster model is suitable for web servers and similar types of applications that follow the client/server model, and characterized by short and high frequency transactions. The architecture does not exclude specializations, however, it can handle them and the HA-OSCAR project is one example on how to extend the HAS architecture to support specialized applications. The HAS architecture provides the infrastructure for cluster membership, cluster storage, fault management, recovery mechanisms, and traffic distribution. It supports various redundancy models for each tier of the architecture and provides a seamless software and hardware upgrade without interruption of service.

To the best of our knowledge, this work appears to be the first to propose and demonstrate a highly available and scalable architecture for web server clusters that meets the following criteria:

1. *The HAS architecture supports high availability at different layers of the cluster: traffic nodes, master nodes, communication, ongoing connections, and data:* The HAS architecture supports additional protection against errors and failures because of the embedded redundancy layers across all the architecture tiers, software modules, and hardware components.
2. *The HAS architecture proof-of-concept demonstrates close to linear scaling for up to 16 nodes:* We are able to add server nodes transparently to the HAS cluster without affecting the servicability or the uptime. When we experience an increase in traffic, we can add more traffic nodes into the SSA tier and experience additional capacity. The HAS architecture proof-of-concept was benchmarked using a

standardized tool and workload and the results demonstrated that the architecture is able to close-to-linearly scale for up to 16 nodes.

3. *The HAS architecture is able to maintain over 96% of base line performance as we increase the number nodes:* The benchmarking results of the HAS architecture proof-of-concept demonstrate that the HAS architecture is able to sustain close the baseline performance per processor as we increase the number of processors in the cluster for up to 16 nodes. The impact on the baseline performance was minimal at a loss of 3.1%, dropping from 1032 requests per second to 1000 requests per second.
4. *The HAS architecture supports online operating system and software upgrade:* With the HAS architecture, we demonstrated the ability to perform maintenance activities such as upgrading the kernel and the system software of the cluster nodes without any associated downtime.
5. *The HAS architecture support multiple redundancy models:* The HAS architecture is characterized by its three tiers, HA, SSA and storage. Each of these tiers supports multiple redundancy models independently from the redundancy model adopted by the other tiers. The HA tier supports the 1+1 (active/active and active/standby), N-way, and N+M redundancy models. The SSA tier supports the N-way and the N+M redundancy modes. Because the SSA tier of the HAS architecture support the N-way redundancy model, the architecture does not force us to deploy traffic nodes in pairs. As a result, we can deploy exactly the right number of traffic nodes to meet our traffic demands without having traffic nodes sitting idle. The storage tier supports the 1+1 (active/active and active/standby), N-way, and N+M redundancy models and it is not restricted to usage of specialized storage nodes.
6. *The HAS architecture uses common-off-the-shelf hardware and software:* Unlike some of the surveyed work, the HAS architecture does not require any specialized software or hardware, and can be built using COTS hardware and software.
7. *The HAS architecture supports dynamic traffic distribution:* The HAS traffic distribution scheme monitors the load of the traffic nodes using multiple metrics, and uses this information to distribute incoming traffic among the traffic nodes. This scheme enables the cluster to operate at or near full capacity in overload situations, which is in contrast to conventional cluster architectures that are subject to congestion and collapse under overload. The traffic management scheme is lightweight and configurable. It is able to cope with failures of individual traffic nodes. It supports a cluster that consists of heterogeneous cluster nodes with different processor speed and memory capacity. The scheme is transparent to web clients who are unaware that the applications run on a cluster of nodes.

8. *The HAS architecture is a reliable architecture:* In our benchmarking experiments, the 100 MB/s connections to the nodes were over 80% saturated and the HAS proof-of-concept was able to withstand such high traffic and continue to provide close to linear scalability.
9. *The HAS architecture supports detection of failures and recovery:* The HAS architecture proof-of-concept is capable to detect failures in traffic nodes, master nodes, file system, Ethernet cards, traffic client, web server software, and ongoing connections, and provide correction action when the failures are detected. The HAS architecture proof-of-concept contributed a modified version of the NFS server with HA extension to provide storage to the HAS cluster nodes, and a specialized mount program that allows mounting of two redundant NFS servers over the same mount point. Furthermore, the HAS architecture proof-of-concept contributes the Ethernet Redundancy Daemon that that monitors the link status of the primary Ethernet port and switches control to the second Ethernet port upon the failure of the first port.
10. *The HAS architecture supports heterogeneous cluster nodes hardware:* The HAS architecture does not assume that all nodes in the cluster have the same hardware configuration. A HAS cluster can consist of traffic nodes with varying processor speed and RAM capacity and still achieve efficient resource utilization taking into consideration the nodes hardware configuration when forwarding traffic.
11. *The HAS architecture supports keep-alive mechanism between traffic nodes and master nodes:* The HAS traffic distribution scheme integrates a keep-alive mechanism, which allows the master node to know when a traffic node is available for service and when it is not available because of either software or hardware problems.
12. *The HAS architecture follows the building block approach:* Since the HAS architecture prototype follows the building block approach, the software modules can be reused in different environments outside of the HAS architecture and can function completely independently outside of a cluster environment. The HAS architecture relies on the integration of many system components into a well-defined, and generic cluster platform.
13. *The HAS architecture supports a cluster single IP interface towards the outside world:* The interface is transparent, scalable, and fault-tolerant.
14. *The HAS architecture supports continuous service:* With the ability to synchronize connections at the master node level, the HAS architecture is capable of providing continuous service to the web clients even in the event of software or hardware failures.

15. *Industry contribution:* The author of this thesis is a contributor to the Carrier Grade specification that defines an architectural model for telecommunication platforms providing voice and data communication services. The Carrier Grade architecture cluster model [62] is an industry standard that is largely based on the HAS architecture with minor modifications to accommodate gateways, signaling, and management communication servers.

Table 16 presents the status of web clusters after the contributions of the HAS architecture.

Area	Status after contributions
Scalable Architecture	The HAS architecture supports incremental scalability at all tiers We are capable of adding nodes to increase capacity in each tier independently The architecture demonstrated close to linear scalability for up to 16 nodes
Availability	HA is supported across all layers of the architecture: nodes, connectivity, network, and data The architecture supports all redundancy models 1+1, N+M, and N-way across all tiers
Performance	The HAS architecture is capable of maintaining 96% of traffic nodes baseline performance
Master Node Availability	Fast failure detection contributing to decreasing the overall MTTR
Traffic Distribution	The mechanism uses traffic nodes load information to perform dynamic distribution through two contributions: the traffic client and the traffic manager. The formula to calculate the load index of the traffic node can be modified to accommodate additional parameters, aside the processor speed, and available free memory.
Heterogeneous Nodes	The architecture supports nodes with different hardware configurations while maximizing resources on each node
Traffic Node Availability	Master nodes are aware when traffic nodes are not available
Application Availability	The application is monitored locally and the traffic node does not receive incoming traffic if the application is not responsive.
Ethernet Redundancy	The Ethernet redundancy daemon polls the primary ports and on failures, deletes all routes to primary port
Data Availability	This dissertation contributed HA extensions to the NFS and a supporting mount program to mount two NFS servers on the same mounting point supporting HA of data.
Automated Installation	Fully automated installation for traffic nodes with disks and diskless traffic nodes
Maintenance and Upgrades	The architecture supports transparent upgrades of software and operating system without any manual interference

Table 16: Status of web clusters

5.2 Future Work

There are several interesting challenges that are still unresolved. The following sub-sections propose several areas that would extend and improve the current work.

5.2.1 Support Linear Scalability Beyond 16 Nodes

With the HAS architecture, we were able to reach a near linear scalability for up to 16 traffic nodes. The next goal is to investigate how to maintain linear scalability beyond 16 nodes. One of the important further investigations in this area relates to minimizing the scaling overhead and the requests pre-processing steps.

5.2.2 Traffic Client Implementation

We plan to allow the traffic client to receive notification that a master node is not available directly from the heartbeat mechanism. With such information, the traffic client stops reporting its load index to a master node that is unavailable, minimizing its communication overhead and resulting in less network traffic.

5.2.3 Additional Benchmarking Tests

Because of limitations such as timing and access to lab hardware, we were not able to conduct more benchmarking in the lab. As future work, we would to establish a larger benchmarking environment that consists of more test machines to generate additional traffic into the HAS cluster with both master nodes in the HA tier are in load sharing mode, 1+1 active/active.

Furthermore, we would like to benchmark the HAS architecture prototype using specialized storage nodes and compare the results to when using the HA NFS implementation to provide storage. These tests will give us insights on the most efficient storage solution.

5.2.4 Redundancy Configuration Manager

The current prototype of the HAS architecture does not support dynamic changes to the redundancy configurations nor transitioning from one redundancy configuration to another. This feature is very useful when the nodes reach a certain pre-defined threshold, then the redundancy configuration manager would for example transition the HA tier from the 1+1 active/standby to the 1+1 active/active, allowing both master nodes to share and service incoming traffic. Such a transition in the current HAS architecture

prototype requires stopping all services on master nodes, updating the configuration files, and restarting all software modules running on master nodes.

The redundancy configuration manager would be the entity responsible for switching the redundancy configuration of the cluster tiers from one redundancy model to another. For instance, when the SSA tier is in the N+M redundancy model, the redundancy configuration manager will be responsible to activate a standby traffic node when a traffic node becomes unavailable. As such, the configuration manager should be aware of active traffic nodes and the states of their components, and their corresponding standby traffic nodes.

5.2.5 Cluster Configuration Manager

The current HAS cluster prototype does not provide a central location for managing the cluster configuration files required by all the system software. As a result, the process of locating and editing all configuration files needed to run the cluster is a daunting experience. There is a need for a central entity, preferably with a user-friendly graphical user interface, that manages all the configuration files that control the operation of the various software modules. This entity, a cluster configuration manager, is a one-stop configuration tool that would enable easy and centralized configuration.

5.2.6 Supporting Cluster Zones and Specialized Traffic Nodes

The concept of cluster zone is as follows: since the cluster is composed of multiple nodes, we divide the cluster into sub-clusters, called cluster zones or simply zones. Each zone provides a specific type of service through defined cluster nodes and as a result, it receives specific type of traffic to those nodes. We have two main challenges in this area: the first is to provide the virtualization of the cluster zones, and the second is the ability to migrate dynamically cluster nodes between several zones based on traffic trends. There are several possible areas of investigation such as defining cluster zones as logical entities in a larger cluster, dynamic node(s) selection to be part of a specialized cluster zone, transitioning the node into the new zone, and investigating queuing theories suitable for such usage models.

5.3 Conclusion

The starting point of this thesis was the question of whether we can have an architecture for web server clusters that is highly available and capable of scaling linearly for up 16 nodes while maintaining baseline performance per each cluster node.

Hypothesis: *Can we have an architecture for web server clusters that is highly available, providing over four nines availability, and capable of scaling linearly for up 16 nodes while maintaining baseline performance per each cluster node?*

High availability, over 99.99%, was achieved by increasing MTBF, which involved improving the quality of the software modules and using redundancy to remove single points of failures, and by decreasing MTTR which involved streamlining and accelerating the fail-over, responding quickly to fault conditions, and making faults more granular in time and scope.

Conclusion: *From this research, we conclude that it is feasible to build highly available cluster architecture for web servers that can scale linearly for up to 16 nodes, while maintaining close to baseline performance per each cluster node.*

From this research, we conclude that it is feasible to design and implement a web server architecture that provides over 99.99% availability and that is linearly scalable for up to 16 nodes. With the HAS architecture we are able to maintain 96% of the baseline performance per each cluster node as we scale the number of traffic nodes in the cluster.

The techniques and methodology used in this work are applicable to scale the architecture beyond 16 nodes while still maintaining high availability. We believe that the HAS architecture is capable of incrementally scaling at all tiers to include over 100 nodes, with support of Gigabit Ethernet and using separate LANs for the HA and SSA tiers, and the storage tier. The HAS architecture brings together aspects of high availability and scalability into a coherent framework. Our experience and evaluation of the architecture demonstrate that the approach is an effective way to build highly available and scalable web clusters.

The HAS architecture represents a new design point for large-scale web servers that supports scalability, high availability, and high performance.

Bibliography

[1] America Online, Press Data Points, http://corp.aol.com/press/press_datapoints.html

[2] Anderson, D., Yang, T., Holmedahl, V., Ibarra, O., *SWEB: Towards a Scalable World Wide Web Server on Multicomputers*, Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii, USA, April 15-19, 1996, pp. 850-856

[3] Andresen, D. et al, *The WWW Prototype of the Alexandria Digital Library*, Proceedings of the International Symposium on Digital Libraries, Japan, August 22 - 25, 1995

[4] Aversa, L., Bestavros, A., *Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting*, Proceedings of the IEEE International Performance Computing and Communications Conference, Phoenix, Arizona, USA, February 2000, pp. 24-29

[5] Barak, A., Shiloh, A., Amar, L., *An Organizational Grid of Federated MOSIX Clusters*, Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), Cardiff, May 2005

[6] Belloum, A. S. Z., Kaletas, E. C., Van Halderen, A. W., Afsarmanesh, H., Peddemors, A. J. H., *A Scalable Web Server Architecture*, IEEE World Wide Web Journal, Volume 5 Number 1, 2002, pp. 5-23

[7] Bestavros, A., Crovella, M., Liu, J., Martin, D., *Distributed Packet Rewriting and its Application to Scalable Server Architectures*, ICNP 1998, pp. 290-297

[8] Bloomberg News, *E*Trade hit by class-action suit*, CNET News.com, February 9, 1999

[9] Brim, M. J., Mattson, T. G., Scott, S. L., *OSCAR: Open Source Cluster Application Resources*, Ottawa Linux Symposium 2001, Ottawa, Canada, July 2001

[10] British Broadcasting Corporation, *Net surge for news sites*, September 12, 2001

[11] Bryhni, H., Klovning, E., Kure, O., *A Comparison of Load Balancing Techniques for Scalable Web Servers*, IEEE Network, July/August 2000, pp. 58-64

[12] Brynjolfsson, E., Kahin, B., *Understanding the Digital Economy: Data, Tool, and Research*, MIT Press, October 2000

[13] Budiaro, S. N., Nishio, S., *MASEMS: A Scalable and Extensible Multimedia Server*, Proceedings of the International Symposium on Database Applications in Non-Traditional Environments, Kyoto, Japan, November 1999, pp. 28-30

[14] Casalicchio, E., Tucci, S., *Static and Dynamic Scheduling Algorithms for Scalable Web Server Farm*, IEEE Network 2001, pp. 368-376

[15] Choi, H., *Markov Regenerative Stochastic Petri Nets,* Computer Performance Evaluation, Vienna 1994, pp. 337-357

[16] Ciardo, G., Muppala, J., Trivedi, K., *SPNP: Stochastic Petri Net Package,* Proceedings of the International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press, Los Alamitos, Ca., USA, December 1989, pp. 142-150

[17] Coffman, K., Odlyzko, A., *The Growth Rate of the Internet*, Technical Report, First Monday, Volume 3 Number 10, October 1998

[18] Damani, O., Chung, P., Huang, Y., Kintala, C., Wang, Y. M., *ONE-IP: Techniques for Hosting a Service on a Cluster of Machines*, IEEE Computer Networks, Volume 29, Numbers 8-13, September 1997, pp. 1019-1027

[19] Dias, D., Kish, W., Mukherjee, R., Tewari, R., *A Scalable and Highly Available Web Server*, Proceedings of the Forty-First IEEE Computer Society International Conference: Technologies for the Information Superhighway, Santa Clara, California, USA, February 25-28, 1996, pp. 85-92

[20] Gan, X., Schroeder, T., Goddard, S., Ramamurthy, B., *LSMAC and LSNAT: Two Approaches for Cluster-based Scalable Web Servers*, Proceedings of the 2000 IEEE International Conference on Communications, New Orleans, USA, June 18-22, 2000, pp. 1164-1168

[21] Gan, X., Schroeder, T., Goddard, S., Ramamurthy, B., *LSMAC vs. LSNAT: Scalable cluster-based Web servers*, IEEE Cluster Computing, November 2000, pp. 175-185

[22] Haddad, I., Butler, B., *Experimental Studies of Scalability in Clustered Web Systems*, Proceedings of the International Parallel and Distributed Processing Symposium 2004, Santa Fe, New Mexico, USA, April 2004

[23] Haddad, I., *Design and Implementation of HA Linux Clusters*, IEEE Cluster 2001, Newport Beach, USA, October 8-11, 2001

[24] Haddad, I., *Designing Large Scale Benchmarking Environments*, ACM Sigmetrics 2002, Marina Del Rey, USA, June 2002

[25] Haddad, I., Leangsuksun, C., Libby, R., Liu, T., Liu, Y., Scott, S., *Highly Reliable Linux HPC Clusters: Self-awareness Approach*, Proceedings of the 2nd International Symposium on Parallel and Distributed Processing and Applications, Hong Kong, China, December 13-15, 2004, pp. 217-222

[26] Haddad, I., Leangsuksun, C., Scott, S., *Towards Highly Available, Scalable, and Secure HPC Clusters with HA-OSCAR*, the 6th International Conference on Linux Clusters, Chapel Hill, NC, USA, April 2005

[27] Hansen, E., *Email outage takes toll on Excite@Home*, CNET News.com, June 28, 2000

[28] Hirel, C., Sahner, R., Zang, X., Trivedi, K. S., *Reliability and Performability Modeling using SHARPE 2000*, Computer Performance Evaluation/TOOLS 2000, Schaumburg, USA, March 2000, pp. 345-349

[29] Hirel, C., Tuffin, B., Trivedi, K. S., *SPNP: Stochastic Petri Nets Version 6.0*, Computer Performance Evaluation/TOOLS 2000, Schaumburg, USA, March 2000, pp. 354-357

[30] Horman, S., *Connection Synchronisation (TCP Fail-Over)*, Technical Paper, November 2003

[31] Horman, S., *Connection Synchronization*, http://www.ultramonkey.org/papers/conn_sync

[32] Horman, S., *Linux Virtual Server Tutorial*, Linux Symposium, Ottawa, Canada, July 2003

[33] HPCinfo, *SMP and MPP Architectures*, http://www.epcc.ed.ac.uk/HPCinfo/hardware.html

[34] Hsieh, J., Leng, T, Fang, Y. C., *OSCAR: A Turnkey Solution for Cluster Computing*, Dell Power Solutions, Issue 1, 2001, pp. 138-140

[35] Hunt, G. D. H., Goldszmidt, G. S., King, R. P., Mukherjee, R., *Network Dispatcher: A Connection Router for Scalable Internet Services*, Proceedings of 7th International World Wide Web Conference, Bribane, Australia, April 1998, pp. 347-357

[36] Intel Corporation, http://www.intel.com

[37] Katz, E. D., Butler, M., McGrawth, M., *A Scalable HTTP Server: The NCSA Prototype*, Proceedings of the International WWW Conference, Geneva, Switzerland, May 25-27, 1994, pp. 155-164

[38] Kim, D., Park, C. H., Park, D., *Request Rate Adaptive Dispatching Architecture for Scalable Internet Server*, IEEE International Conference on Cluster Computing, Chemmnitz, 2000, pp. 289-296

[39] LaLiberte, D., Braverman, A., A *Protocol for Scalable Group and Public Annotations*, Computer Networks and ISDN Systems, Volume 27, Number 6, January 1995, pp. 911-918

[40] Leangsuksun, C., *A Failure Predictive and Policy-Based High Availability Strategy for Linux High Performance Computing Cluster*, The 5th LCI International Conference on Linux Clusters: The HPC Revolution 2004, Austin, USA, May 18-20, 2004

[41] Leangsuksun, C., et al, *Highly Reliable Linux HPC Clusters: Self-awareness Approach*, Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN), Hong Kong, China, May 2004

[42] Leangsuksun, C., Haddad, I., Libby, R., Liu, T., Liu, Y., Scott, S. L., *High-Availability and Performance Clusters: Staging Strategy, Self-Healing Mechanisms, and Availability Analysis*, Proceedings of the IEEE Cluster Conference 2004, San Diego, USA, September 20-23, 2004

[43] Leangsuksun, C., Shen, L., Liu, T., Song, H., Scott, S., *Availability Prediction and Modeling of High Availability OSCAR Cluster*, IEEE International Conference on Cluster Computing, Hong Kong, China, December 2-4, 2003, pp. 227-230

[44] Leangsuksun, C., Shen, L., Liu, T., Song, H., Scott, S., *Dependability Prediction of High Availability OSCAR Cluster Server*, The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, 2003, pp. 23-26

[45] Leangsuksun, C., Shen, L., Lui, T., Scott, S. L., *Achieving High Availability and Performance Computing with an HA-OSCAR Cluster*, Future Generation Computer System, Volume 21, Number 1, January 2005, pp. 597-606

[46] Leangsuksun, C., Shen, L., Song, H., Scott, S., Haddad, I., *The Modeling and Dependability Analysis of High Availability OSCAR Cluster System*, The 17th Annual International Symposium on High Performance Computing Systems and Applications, Sherbrooke, Quebec, Canada, May 11-14, 2003

[47] LeFebvre, W., *Facing a World Crisis*, 15th USENIX System Administration Conference, San Diego, California, USA, December 2-7, 2001

[48] Marcus, E., Sten, H., *BluePrints for High Availability: Designing Resilient Distributed Systems,* Wiley, 2000

[49] Marowsky-Brée, L., *A New Cluster Resource Manager for Heartbeat*, UKUUG LISA/Winter Conference High Availability and Reliability, Bournemouth, UK, February 2004

[50] Massie, M. L., Chun, B. N., Culler, D. E., *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*, Parallel Computing, Vol. 30, Issue 7, July 2004

[51] McNaughton, K., *Is eBay too popular?*, CNET News.com, March 1, 1999

[52] Microsoft Developer Network Platform SDK, *Performance Data Helper*, Microsoft Corp., July 1998

[53] Miller, R. B., *Response Time in Man-Computer Conversational Transactions*, Proceedings of the FIPS Fall Joint Computer Conference, Vol. 33, 1968, pp. 267-277

[54] Muppala, J. K., Ciardo, G. F., Trivedi, K. S., *Stochastic reward nets for reliability prediction*, Communications in Reliability, Maintainability and Serviceability: An International Journal published by SAE International, July 1994 Vol. 1, No. 2, pp. 9-20

[55] Murad, A. N., Liu, H., *Scalable Web Server Architectures*, Technical Report BL0314500-961216TM, Bell Labs, Lucent Technologies, December 1996

[56] Narten, T., Nordmark, E., Simpson, *RFC 2461 Neighbor Discovery for IP Version 6,* December 1998

[57] Nielsen, J., *The Need for Speed*, Technical Report, March 1, 1997

[58] Nielsen, J., *Usability Engineering*, Morgan Kaufmann, San Francisco, 1994

[59] OpenGFS clustered file system, http://opengfs.sourceforge.net

[60] OpenSSI Clustering Solution, http://openssi.org/cgi-bin/view?page=openssi.html

[61] Oracle cluster file system, http://oss.oracle.com/projects/ocfs

[62] OSDL, *Carrier Grade Requirement Definition Document Version 3.1*, February 2006

[63] Pfister, G., *In Search of Clusters*, 2nd Edition, Prentice Hall PTR, 1998

[64] Ramamurthy, B., *LSMAC vs. LSNAT: Scalable Cluster-based Web Servers*, Seminar Presentation, Rice University, October 23, 2000

[65] Robertson, A. L. *The Evolution of the Linux-HA Project*, UKUUG LISA/Winter Conference High-Availability and Reliability, Bournemouth, UK, February 25-26, 2004

[66] Robertson, A. L., *Linux-HA Heartbeat Design,* Proceedings of the 4th International Linux Showcase and Conference, Atlanta, USA, October 10-14, 2000

[67] Roe, C., Gonik, S., *Server-Side Design Principles for Scalable Internet Systems*, IEEE Software, Volume 19, Number 2, March/April 2002, pp. 34-41

[68] Sandoval, G., Wolverton, T., *Leading Web Sites Under Attack*, News.com, February 9, 2000

[69] Schroeder, T., Goddard, S., Ramamurthy, B., *Scalable Web Server Clustering Technologies*, IEEE Network Special Issue on Web Performance, 14 (3), 2000, pp 38-45

[70] Squillante, M., Lazowska, E., *Using Processor-Cache Affinity Information in Shared-Memory Multiprocessors Scheduling*, IEEE Transactions on Parallel and Distributed Systems, Volume 4 Number 2, February 1993, pp. 131-134

[71] Srisuresh, P., Gan, D., *RFC 2391: Load Sharing using IP Network Address Translation*, August 1998

[72] Tewari, R., Dias, D., Mukherjee, R., Vin, H., *High Availability in Clustered Multimedia Servers*, Proceedings of the International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, pp 645-654

[73] The DRDB Tool, http://www.drbd.org

[74] The Linux Virtual Server Project, http://www.linuxvirtualserver.org

[75] The Open Group, *The UNIX® Operating System: A Robust, Standardized Foundation for Cluster Architectures*, White Paper, June 2001

[76] The ROCKS Clustering Package, http://www.rocksclusters.org

[77] The WebBench Tool, http://www.etestinglabs.com/benchmarks/webbench/webbench.asp

[78] Tridgell, A., *Efficient Algorithms for Sorting and Synchronization,* Australian National University, February 1999

[79] Tucker, A., Gupta, A., *Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors*, Proceedings of the Symposium on Operating Systems Principles, ACM, Litchfield Park, Arizona, USA, December 1989, pp. 159-166

[80] TurboLinux Cluster Server, http://www.turbolinux.com/products/middleware/tlcs8.html

[81] WebStone Web Server Benchmarking Tool, http://www.mindcraft.com/webstone

[82] Yahoo! Fourth Quarter 2001 Financial Report, http://docs.yahoo.com/docs/pr/4q01pr.html

[83] Zhang, X., Barrientos, M., Chen, B., Seltzer, M., *HACC: An Architecture for Cluster-Based Web Servers*, Proceedings of the USENIX Windows NT Symposium, Seattle, Washington, USA, July 12-15, 1999, pp. 155-164

Glossary

The definitions of the terms appearing in this glossary are from TechTarget, the dictionary for Internet and Computer Technologies (whatis.techtarget.com).

AAA Authentication, authorization, and accounting (AAA) is a term for a framework for intelligently controlling access to computer resources, enforcing policies, auditing usage, and providing the information necessary to bill for services. These combined processes are considered important for effective network management and security. Authentication, authorization, and accounting services are often provided by a dedicated AAA server, a program that performs these functions. A current standard by which network access servers interface with the AAA server is the Remote Authentication Dial-In User Service (RADIUS).

Daemon A program that runs continuously in the background, until activated by a particular event. A daemon can constantly query for requests or await direct action from a user or other process.

DNS The domain name system (DNS) is the way that Internet domain name are located and translated into IP addresses. A domain name is a meaningful and easy-to-remember "handle" for an Internet address.

Failover The ability to automatically switch a service or capability to a redundant node, system, or network upon the failure or abnormal termination of the currently active node, system, or network.

Failure The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered. Examples of failures include invalid data being provided, slow response time, and the inability for a service to take a request. Causes of failure can be hardware, firmware, software, network, or anything else that interrupts the service.

FTP File Transfer Protocol (FTP) is a standard Internet protocol that defines one way of exchanging files between computers on the Internet.

Gateways Gateways are bridges between two different technologies or administration domains. A media gateway performs the critical function of converting voice messages from a native telecommunications time-division-multiplexed network, to an Internet protocol packet-switched network.

HLR The Home Location Register (HLR) is the main database of permanent subscriber information for a mobile network.

HTML Hypertext Markup Language (HTML) is the set of markup symbols or codes inserted in a file intended for display on a World Wide Web browser page. The markup tells the web browser how to display a web page's words and images for the user.

HTTP Hypertext Transfer Protocol (HTTP) is the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.

Internet The Internet is a worldwide system of computer networks - a network of networks in which users at any one computer can, if they have permission, get information from any other computer (and sometimes talk directly to users at other computers). It was conceived by the Advanced Research Projects Agency (ARPA) of the U.S. government in 1969 and was first known as the ARPANET. The Internet is a public, cooperative, and self-sustaining facility accessible to hundreds of millions of people worldwide. Physically, the Internet uses a portion of the total resources of the currently existing public telecommunication networks.

IP The Internet Protocol (IP) is the method or protocol by which data is sent from one computer to another on the Internet.

`iptables` is a Linux command used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. There are several different tables, which may be defined, and each table contains a number of built-in chains, and may contain user-defined chains. Each chain is a list of rules which can match a set of packets: each rule specifies what to do with a packet which matches. This is called a `target', which may be a jump to a user-defined chain in the same table.

IPv6 Internet Protocol Version 6 (IPv6) is the latest version of the Internet Protocol. IPv6 is a set of specifications from the Internet Engineering Task Force (IETF) that was designed as an evolutionary set of improvements to the current IP Version 4.

I/O I/O describes any operation, program, or device that transfers data to or from a computer.

ISP Internet service provider (ISP) is a company that provides individuals and other companies access to the Internet and other related services such as web site building and virtual hosting.

LAN A local area network (LAN) is a group of computers and associated devices that share a common communications line or wireless link and typically share the resources of a single processor or server within a small geographic area.

Management Server Management servers handle traditional network management operations, as well as service and customer management. These servers provide services such as Home Location Register and Visitor Location Register (for wireless networks) or customer information, such as personal preferences including features the customer is authorized to use.

NAS Network-attached storage (NAS) is hard disk storage that is set up with its own network address rather than being attached to the department computer that is serving applications to a network's workstation users.

Network A connection of [nodes] which facilitates [communication] among them. Usually, the connected nodes in a network use a well defined [network protocol] to communicate with each other.

Network Protocols Rules for determining the format and transmission of data. Examples of network protocols include TCP/IP, and UDP.

NIC A network interface card (NIC) is a computer circuit board or card that is installed in a computer so that it can be connected to a network.

NTP Network Time Protocol (NTP) is a protocol that is used to synchronize computer clock times in a network of computers.

OSI The Open System Interconnection, model defines a networking framework for implementing protocols in seven layers. Control is passed from one layer to the next, starting at the application layer in one station, and proceeding to the bottom layer, over the channel to the next station and back up the hierarchy.

PCI PCI (Peripheral Component Interconnect) is an interconnection system between a microprocessor and attached devices in which expansion slots are spaced closely for high-speed operation. Using PCI, a computer can support both new PCI cards while continuing to support Industry Standard Architecture (ISA) expansion cards, an older standard.

Proxy Server A computer network service that allows clients to make indirect network connections to other network services. A client connects to the proxy server, and then requests a connection, file, or other resource available on a different server. The proxy provides the resource either by connecting to the specified server or by serving it from a cache. In some cases, the proxy may alter the client's request or the server's response for various purposes.

RAID Redundant array of independent disks (RAID) is a way of storing the same data in different places (thus, redundantly) on multiple hard disks.

RAMDISK A RamDisk is a portion of memory that is allocated to be used as a hard disk partition.

Recovery To return a failing component, node or system to a working state. A failing component can be a hardware or a software component of a node or network. Recovery can also be initiated to work around an fault that has been detected; ultimately restoring the service.

RTT Round-Trip Times (RTT) is the time required for a network communication to travel from the source to the destination and back. RTT is used by routing algorithms to aid in calculating optimal routes.

SAN Storage Area Network (SAN) is a high-speed special-purpose network (or sub-network) that interconnects different kinds of data storage devices with associated data servers on behalf of a larger network of users.

SCP A Service Control Point server is an entity in the intelligent network that implements service control function that operation that affects the recording, processing, transmission, or interpretation of data.

Service A set of functions provided by a computer system. Examples of Telco services include media gateway, signal, or soft switch types of applications. Some general examples of services include web based or database transaction types of applications.

Session Series of consecutive page requests to the web server from the same user

Signaling Servers Signaling servers handle call control, session control, and radio recourse control. A signaling server handles the routing and maintains the status of calls over the network. It takes the request of user agents who want to connect to other user agents and routes it to the appropriate signaling.

SLA Service Level Agreement (SLA) is a contract between a network service provider and a customer that specifies, usually in measurable terms, what services the network service provider will furnish.

SSI Single System Image (SSI) is a form of distributed computing in which by using a common interface multiple networks, distributed databases or servers appear to the user as one system. In SSI systems, all nodes share the operating system environment in the system.

System A computer system that consists of one computer [node] or many nodes connected via a computer network mechanism.

Switch-over The term switch-over is used to designate circumstances where the cluster moves the active state of a particular component/node from one component/node to another, after the failure of the active component/node. Switch-over operations are usually the consequence of administrative operations or escalation of recovery procedures.

TCP TCP (Transmission Control Protocol) is a set of rules (protocol) used with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.

TFTP Trivial File Transfer Protocol (TFTP) is an Internet software utility for transferring files that is simpler to use than the File Transfer Protocol (FTP) but less capable. It is used where user authentication and directory visibility are not required.

TTL Time-to-live (TTL) is a value in an Internet Protocol (IP) packet that tells a network router whether the packet has been in the network too long and should be discarded.

User An external entity that acquires service from a computer system. It can be a human being, an external device, or another computer system.

Web Service Web services are loosely coupled software components delivered over Internet standard technologies. A web service can also be defined as a self-contained, modular application that can be described, published, located, and invoked over the web.