Performance and Availability Modeling of Webserver Configurations

Geof Pawlicki
841 Sharmon Palms Lane, Apt. D
Campbell CA 95008

Archana Sathaye
Department of Math and Computer Science
San Jose State University
San Jose CA 95192

Kishor Trivedi
Department of Electrical and Computer Engineering
Duke University
Durham NC 27708

Abstract

Large scale Webserver configurations are widely implemented by Internet portals, and hence have evolved to rely upon numerous performance and high availability system mechanisms. The principle objective of this paper is to provide a method to compare different webserver configurations with respect to performance and availability. Towards this goal, we model the structural and behavioral characteristics of these configurations using Stochastic Reward Nets and introduce several performance and availability measures to compare the webserver configurations. We model three configurations: (1) a round-robin scheduled Domain Name Service (DNS) cluster; (2) a round-robin cluster with a shared file server , specifically a Redundant Array of Inexpensive Disks (RAID) cache; (3) and a Cache Array Routing Protocol (CARP) cluster. These configurations are differentiated by the manner of utilization of their disk caches and job scheduling. The behavioral characteristics common to all the configurations include a caching protocol, fault-tolerance by failover to peer standby systems and distributed parallel processing. Principally, our results show that the superior throughput of the centralized File System cluster comes as a result of improved capacity oriented availability but at the cost of corresponding losses in performance oriented availability – the probability of completion within a specified threshold - when compared to the clusters utilizing distributed storage. We also show an overall increased throughput and corresponding availability gains attributable to the dynamic failover mechanism effected by re-caching in the CARP configuration when compared to the simple Round-Robin configuration.

1. Introduction

The performance and availability of webservers - particularly clusters of web proxy servers - is of central importance to the mass acceptance of web-based commerce. A recent study conducted by Carnegie-Mellon University shows that in cases when a user is unable to connect to a site there is a 50% probability that they will never return. Modeling offers one method for the owners of portals and other large websites to predict and provide for such service anomalies. Much of the work to date by Menasce[6] and others has concentrated primarily on performance issues. While of critical importance, these do not reflect availability and the effects of fault tolerance mechanisms. Similarly, pure availability measures do not characterize the queuing behavior and resulting performance degradation. In this paper we combine these perspectives. Measures are provided for availability, performance, and their combination to investigate performance within defined thresholds of acceptable performance. We provide comparative results between parameterized versions of several common webserver configurations. Specifically, we model (1) a round-robin scheduled Domain Name Service (DNS) cluster, in which each server maintains it's own distinct cache on a local disk; (2) a round-robin cluster with a shared Redundant Array of Inexpensive Disks (RAID) cache (3) and a Cache Array Routing Protocol (CARP) [12,13] cluster in which the namespace of the files served is divided by means of a hash function amongst the local disks of the available proxies.

Such modeling has traditionally been conducted in terms of product form queuing networks, which are generally insufficient to describe systems demonstrating concurrent behavior. While Markov chains are useful for small-scale problems, they are effectively limited by the prohibitive complexity arising from the enumeration of reachable states in larger configurations. Petri Nets[7] provide a higher level formalism, accessible through a graphically intuitive interface, to model logic and time constrained transitions between places representing decision and queuing resources. The state space generated by the reachability graph of a Petri Net can be mapped into a Markov chain model, which in turn is solved by numerical methods[8]. The addition of priorities either explicitly between enabled transitions or by the marking of one place inhibiting transition to another - results in Generalized Stochastic Petri Nets (GSPN) with computational power equivalent to a Turing machine. In particular, we model the configurations using Stochastic Reward Nets, which allow extensive marking dependenices and reward rate specifications for computing complex measures[2]. In addition to exponentially distributed service times and interarrival times, we utilize a Zipf distribution [19] to model the request pattern for particular files. We analyze the models using the Stochastic Petri Net Package (SPNP) [9,10], a general

purpose software package to analyze such nets. Our results show the advantages of distributed storage mechanisms for both performance and availability, and the general utility of the logical failover and load balancing mechanism afforded by the use of CARP.

This paper is organized as follows. Section 2 presents a description of key features of the configurations modeled, along with a discussion of the general modeling techniques utilized relative to the types of measures sought. Section 3 presents the results obtained from analyzing the models. Section 4 summarizes the paper and presents the conclusion and future research directions.

2. SRN Models of Webserver Cluster Configurations

A generic internet client server configuration is shown in Figure 1. This configuration includes proxies at the client and server side. Proxies are commonly used at client sites, e.g. between a router and a corporate intranet. Proxy servers originated as simple firewalls, characteristically preventing unauthorized access to particular subnetworks by particular users or traffic types. Because of this unique sentry position, Proxy servers were later utilized to cache data obtained from prior HTTP read requests. By contrast dynamically generated content - such as server-parsed HTML, CGI scripts, custom-server API applications and specialized servers - never run on proxies. In this paper we refer to Proxy servers simply as servers, and reserve the term Origin Server for the other functions. If the proxy has the requested file, it responds itself in stead of the Origin server it represents, otherwise it forwards the request upstream, to the origin of the requested file.

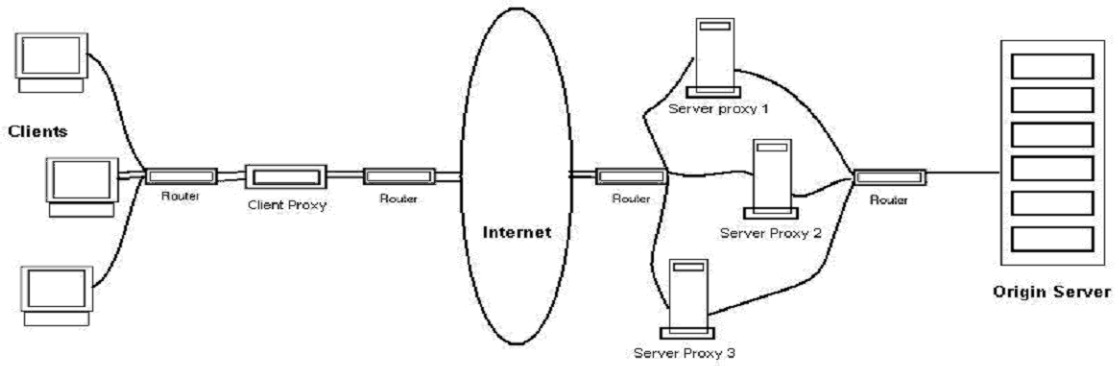


Figure 1: Internet Configuration with Client and Server side Proxies

The key difference between proxies shown in Figure 1 is that a 30-60% cache hit rate can be expected on the client side, whereas the rate for the Proxy server cluster can be 99%+ [6]. This paper focuses on their use on the server side, where they are commonly referred to as reverse proxies. The webserver cluster configurations modeled in this paper are clusters of proxy servers. Throughout this paper we will refer to *proxy servers* simply as *webservers.*

Behavioral characteristics common to all the server configurations modeled include a caching protocol, fault-tolerance by failover to peer standby systems and distributed parallel processing of HTTP requests. The configurations are differentiated by the manner of utilization of their disk caches and job scheduling. Specifically, we model: (1) a round-robin scheduled Domain Name Service (DNS) cluster, in which each server maintains its own distinct cache on a local disk; (2) a round-robin cluster with a shared Redundant Array of Inexpensive Disks (RAID) cache (3) and a Cache Array Routing Protocol (CARP) cluster in which the namespace of the files served is divided by means of a hash function amongst the local disks of the available proxies. The models allow complex configurations such as clusters of heterogeneous servers, and a combination of individual cluster configurations to model globally distributed, heterogeneous websites.

We model these complex clusters using Stochastic Reward nets , which are extensions of Generalized Stochastic Petri Nets (GSPNS). A GSPN is a special case of Stochastic Petri Nets. Formally, a Stochastic Petri net is defined as a 11-tuple[2]:

$SPN = (P, T, DI, DO, DH, PRIO, EN, F, PS, POL, MO)$ where:

$P = \{p_1, p_2, \cdots, p_m\}$ is a finite set of places,

$T = \{t_1, t_2, \cdots, t_n\}$ is a finite set of transitions,

$\forall p_i \in P, \forall t_i \in T, DI_{i,j} : \mathrm{N}^m \rightarrow \mathrm{N}$ is the (possibly) marking dependent multiplicity of the input arc from place p_i to transition t_j; if the multiplicity is zero, the input arc is absent.

$\forall p_i \in P, \forall t_i \in T, DO_{i,j} : \mathrm{N}^m \rightarrow \mathrm{N}$ is the (possibly) marking dependent multiplicity of the output arc from transition t_i to place p_j; if the multiplicity is zero, the output arc is absent.

$\forall p_i \in P, \forall t_i \in T, DH_{i,j} : \mathrm{N}^m \rightarrow \mathrm{N}$ is the (possibly) marking dependent multiplicity of the inhibitor arc from place p_i to transition t_j; if the multiplicity is zero, the inhibitor arc is absent.

$\forall t_i \in T, PRIO_i \in \mathrm{N}$ is the priority of the transition t_i .

$\forall t_i \in T, EN_i : N^m \to \{0,1\}$ is the marking dependent enabling of the transition t_i .; if no enabling function is defined for t_i, then $EN_i = 1$.

$\forall t_i \in T, F_i : \mathrm{N}^m \to F$ is the (possibly) marking dependent firing time distribution of the transition t_i (F is the family of distributions having a non-negative image).

Let $S \in 2^T, \forall s_j \in T_i, \forall s \in S, PS_{i,j} : \mathrm{N}^m \to \{0,1\}$ is the marking dependent firing time distribution of the transition t_j given that the transitions in $s \in S$ are enabled and are scheduled to fire at the same time.

$\forall s \in S, \forall t_i \in T, POL_j : \mathrm{N}^m \to \{PRI, PRD, PRS\}$ is the marking dependent interruption policy for transition t_j given that the interruption was caused by the simultaneous firing of the set of transitions in s (a less general definition is usually adequate, where the policy is not dependent on the cause of the interruption, or even on the marking). $M0 \in \mathrm{N}^M$ is the initial marking.

Figure 2 shows the graphical notation used for the nets throughout the paper.

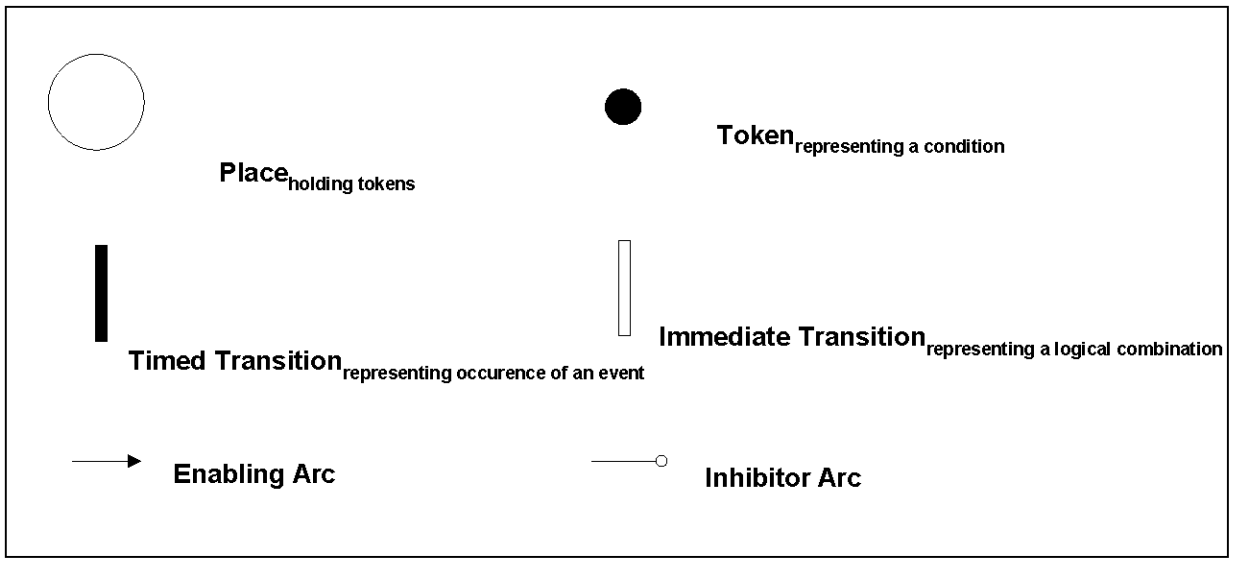


Figure 2: Graphical Notation for Petri Nets used in Our Models.

A Stochastic Petri Net is a Stochastic Reward Net (SRN) if

$\forall t_j \in T$ either F_j is exponentially distributed with rate of firing $\mu(t_j) : \mathrm{N}^M \to \Re^+$ or F_j is deterministic distributed with value 0.

$\forall s \in S, \forall t_i \in T, POL_{s,i} : \mathrm{N}^m \to \{PRD, PRS\}$ is the marking dependent interruption policy for transition t_j.[2]

2.1 Round Robin DNS

In its most basic form, DNS repeatedly cycles through a list of server IP addresses eligible to service HTTP requests destined for a particular hostname. Figure 3 shows a subnet of each of the clusters types studied that represents this function of the cluster as a whole, beginning with entry at place P_{entry} and dispatch to individual servers. The initial marking shows token(s) in P_{entry} representing multiple client requests arriving at the DNS, and a single token in P_{ready_i} representing an available server.

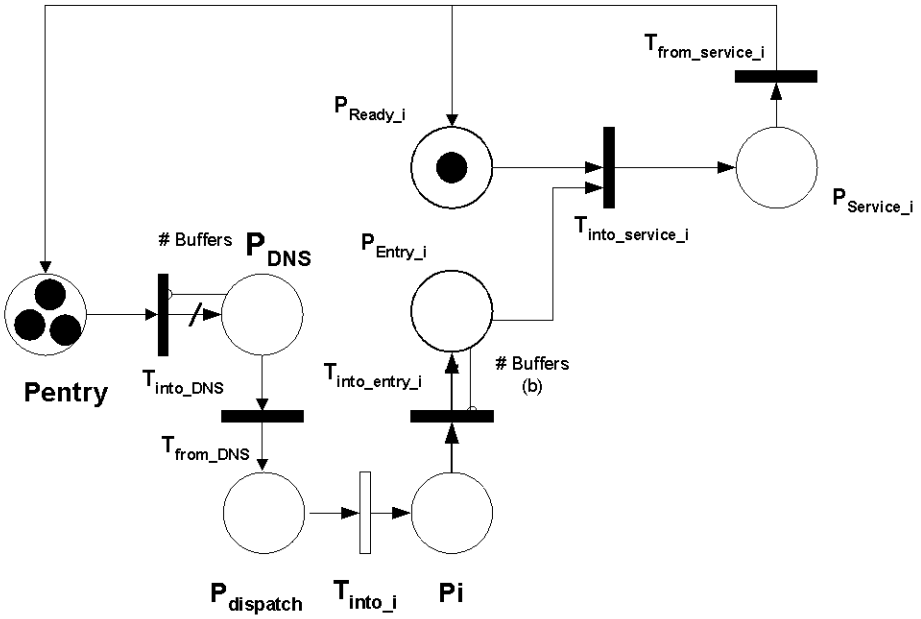


Figure 3: Entry, Dispatch, Buffering of Requests and Admission to Service at Server i.

The initial number of tokens in P_{ready_i} represents the number of processors in the server itself. When there are token(s) in P_{ready_i} and a request is received, the transition $T_{into\_service}$ is enabled. Upon firing, the marking of P_{ready_i} is decremented. Inhibitor arcs to T_{into_DNS} and $T_{into\_entry\_i}$ represent the constraints of available buffer space at the DNS server and the *ith* server respectively. Depending upon proxy server architecture, this latter might for example correspond

to the size of its thread pool. If a job is admitted to service, upon completion its token is returned to both P_{ready_i} and P_{entry} signifying respectively the availability of the server to accept another request, and the possibility that the client will make another request.

Failure conditions added to this basic configuration are represented by a series of transitions terminating in P_{down_i} and $P_{down\_no\_DNS}$ as shown in Figure 4. These beginning at P_{ready_i} and follow either of two primary paths, essentially representing application failure and a broader machine failure. Both happen at user specified rate and require a user configurable period of time to repair, representing for example application thread restart or server reboot. During this time the server is unavailable, so the token deposited in P_{down_i} is used to enable the transition $T_{lost\_job\_i}$ to P_{entry_i} but not to P_{ready}. Upon completion of the reconfiguration or reboot, the token in P_{down} is returned to P_{ready_i} but not to P_{entry}.

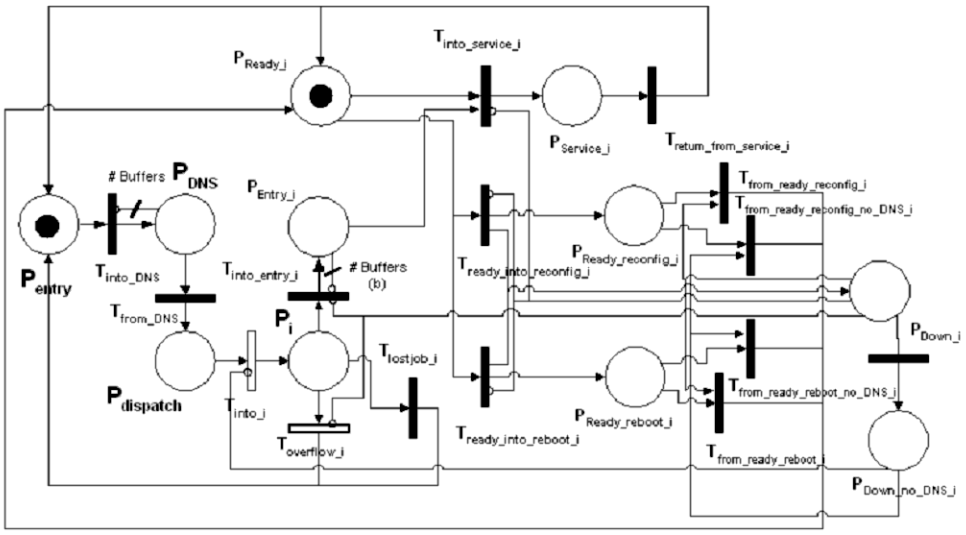


Figure 4: Failure transitions from P_Ready_i into P_Down_i and P_Down_no_DNS_i cause lost jobs or inhibit entry to Server. Tokens are returned to P_Entry at completion of repair.

With respect to availability modeling, DNS maintains no state regarding the health, utilization or quorum status of the servers registered with it. In order to prevent lost traffic, a server's address must be manually removed from the list of available destinations to which a particular URL can be mapped. Subsequent requests are then routed to other servers in the list, effectively representing a linear degradation of service. These factors are represented by the addition of a transition from P_{down_i} into $P_{down\_no\_DNS\_i}$. A token in this latter place inhibits the entry of any more clients into P_i. From the end user's perspective, a lost request means that the browser client

has tried and failed to establish a connection to a remote server. This is shown in the model simply by the returning of a token representing the client to the entry place.

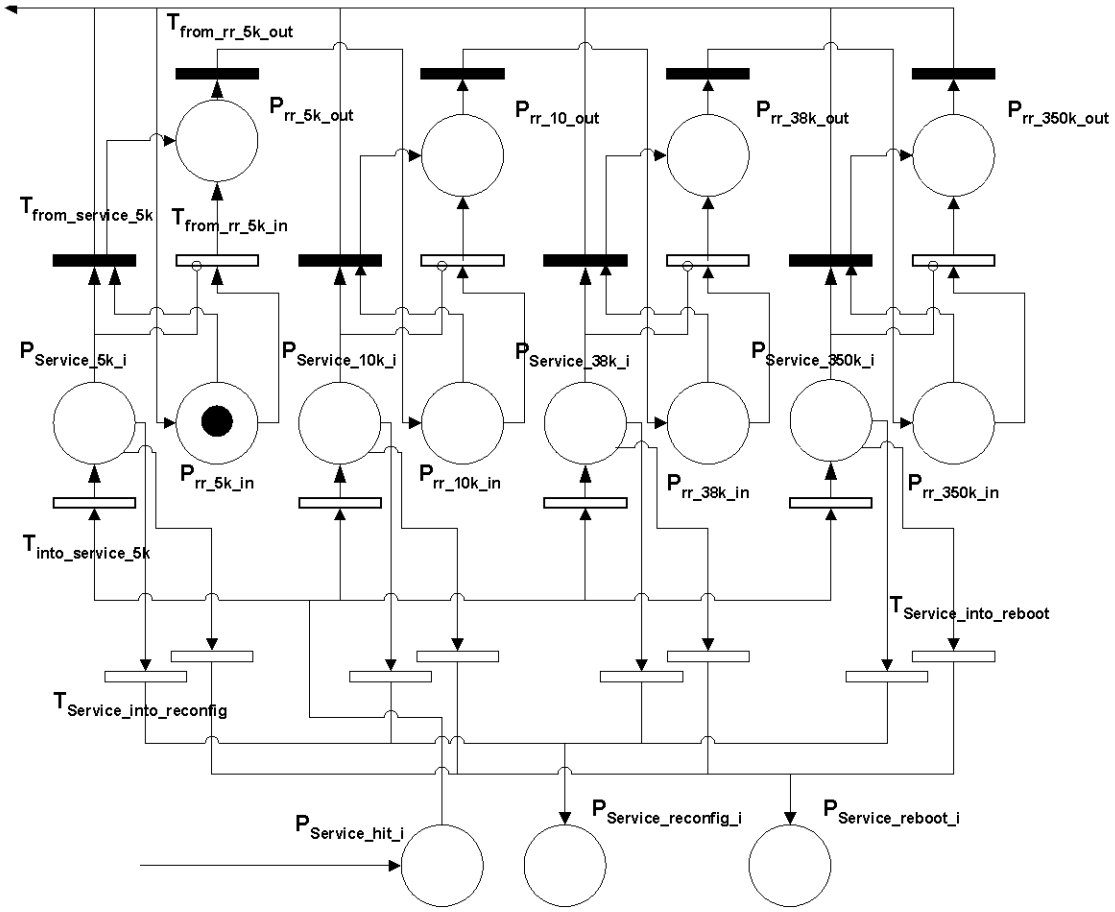


Figure 5: Job Size Classed Service Places with Round-Robin control transitions

The failure scenarios outlined can occur at any time in the HTTP accept-process-reply cycle. As shown in figure 5, a parallel group of failure transitions exists from the service places as well. Once admitted to service, there are two basic types of jobs requests: those satisfied by a disk read at a Proxy server, and those requiring non-cacheable computations such as a database query or FTP access. The latter are simply forwarded for processing to the origin server represented by the proxies. These occur at a user configurable percentage of the workload, with an associated service rates. The more common HTTP cache read requests are further segregated by their job size using the method of centroid analysis of HTTP logs presented by Menasce[6]. Their service is serial and blocking, as represented by an average per class service rate that includes the disk seek and read latencies together with the average network latency. The method of modeling is essentially a round-robin dispatch [8] to a parallel group of places, e.g. $P_{Service\_5k\_i}$, representing the job classes

as shown in Figure 5. Exit from service is enabled by a token in, for example, $P_{rr\_5k\_in\_i}$. Upon completion of a job, this enabling token is forward, e.g. via $P_{rr\_5k\_out\_i}$ to the next enabling position. If there is no job waiting at the corresponding $P_{Service\_10k\_i}$, the enabling token simply cycles to the next enabling place.

In a multi-processor system, the rate of service is dependent upon both the number and the type of the available processors together with the size of the file being serviced. The ability to calculate such marking dependent rewards emphasizes the expressive modeling power of Stochastic Reward Nets. The complete drawing of the Round Robin model SRN and a listing of it's transition's are available at http://www.mathcs.sjsu.edu/faculty/sathaye/IFIP_1_11.pdf

2.2 Round Robin DNS with Shared File Server

A common variant of Proxy caching is to use a shared file server, specifically a Redundant Array of Inexpensive Disks (RAID). This is represented as a dispatch from $P_{in\_Service\ i}$ to P_{disk_j} as shown in Figure 6. The larger and safer cache affords a logarithmically greater hit rate[14], and coherency is assured by on-command refresh of a single copy of each file. Offsetting these advantages are the increased cost of network (LAN) communication. HTTP service is effectively broken into two phases, the parsing of the request and the disk access. For a complete listing of the SRN see http://www.mathcs.sjsu.edu/faculty/sathaye/IFIP_1_11.pdf.

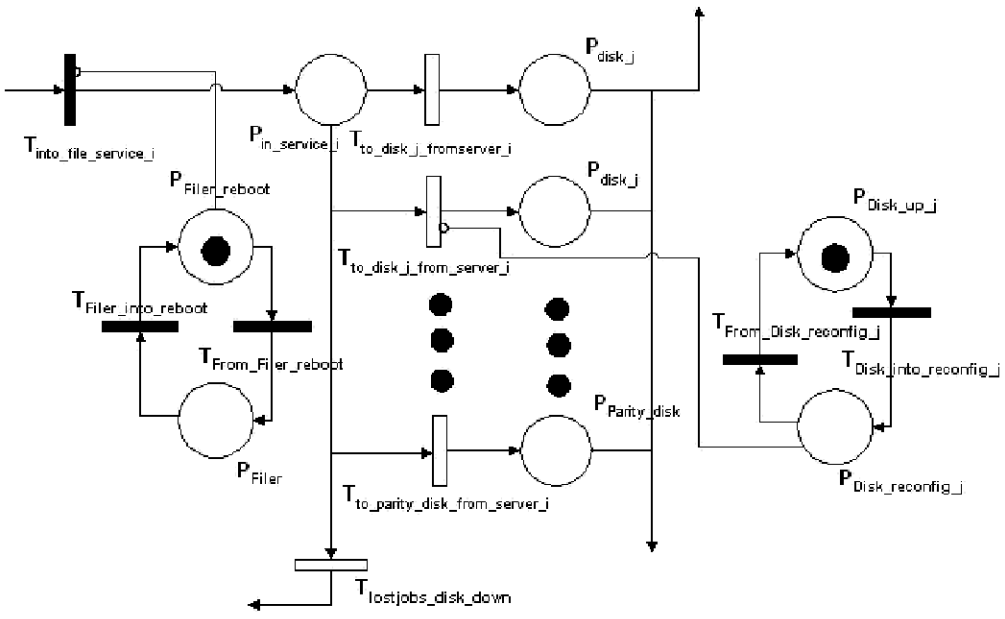


Figure 6: Dispatch of a read request to one of File Server's disks j from Proxy Server i

Entry into the file service and into an individual disk are inhibited by the presence of in P_{Filer_reboot} or $P_{Disk\_reconfig\_j}$, respectively. We simulate a Level 4 RAID protection: a single parity disk is available to dynamically replace any failed disk under the control of a single (non-redundant) network filer. Upon replacement, the new disk assumes the responsibilities of the parity disk. Multiple disk failures result in job losses represented by the transition $T_{lostjob\_disks\_down}$, which operates independently of the File Server processor.

Concurrent with the dispatch to a particular disk for service from $P_{in\_Service\ i}$, another token is forked to a place representing the I/O bound job, $P_{disk\_j\_from\_server\_i}$. The return from file service is represented in Figure 7 by $P_{from\_disk\_j}$. Dispatch into one of the $P_{server\_i\_from\ isk\_j}$ results in enabling the corresponding $T_{from\ disk\_j\_from\_server\_i}$ and the subsequent return of a token to $P_{ready\ i\_}$ and P_{Entry}.

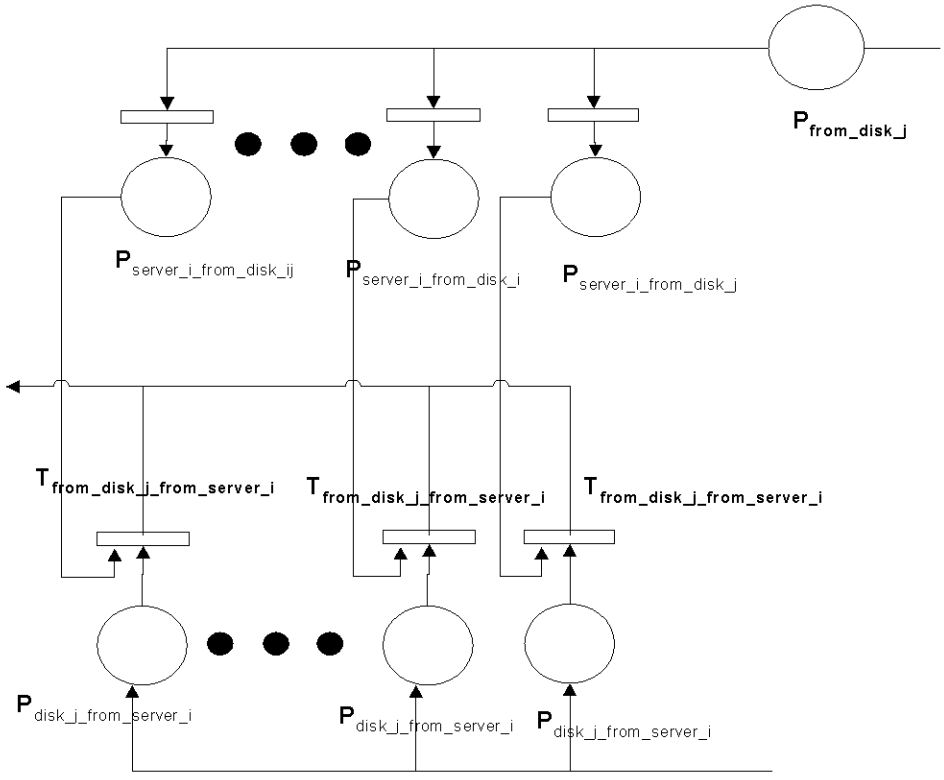


Figure 7: Transitions to synch return from Disk j, Server i

2.3 Cache Array Routing Protocol (CARP) Proxy Cluster

The CARP model most closely resembles round-robin DNS in that each server maintains a local cache. However, a global file address space is effected by use of a hash function performed on the requested file's URL to deterministically locate it. The distribution of location is always uniform amongst the available proxies, and hence vis. modeling is logically equivalent to the round-robin dispatch. It is the handling of proxy failure that most distinguishes the CARP configuration most from those relying upon DNS dispatch. For a complete listing of the SRN and its transitions see http://www.mathcs.sjsu.edu/faculty/sathaye/IFIP_1_11.pdf

Essentially, an additional place $P_{file\_count\_i,}$ is added and marked to represent the files available at each server. Files are added either by the normal service of cache misses, or by a migration from other servers. That is, a proxy failure results in the on-demand re-caching of its files at other proxies in the cluster. If the maximum number is reached, it represents a cache replacement. The mechanism controlling the failover is essentially similar to that used to represent lost jobs in the round-robin DNS model. Deposit of a token in P_{down_i} results in all subsequent requests effectively being forwarded to another probabilistically determined server as shown in Figure 8.

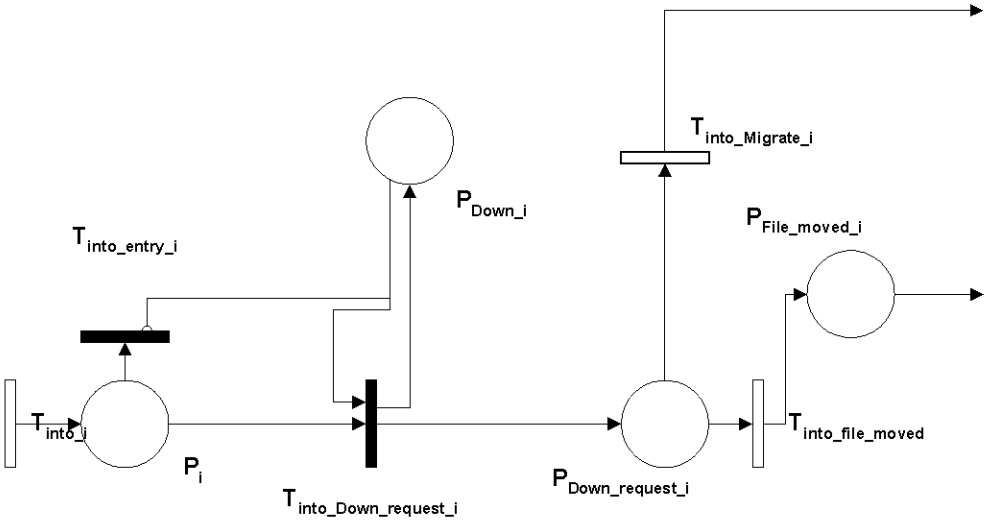


Figure 8: File Migration Enabled by P_down

The receipt of a request when the server is down is subsequently treated as either a file migration or the simple forwarding depending upon the number of files remaining, the number of tokens in $P_{file\_count\_i}$. The probability of a particular files access is assumed to be Zipf distributed with

respect to the files popularity, according to the function $p_m = \frac{G}{m^\alpha}, m = 1,...,M$ where m is the popularity rank of the file and $G = \frac{1}{1 + 1/2^\alpha + 1/3^\alpha + ... + 1/M^\alpha}$. The parameter α may be specified according to the expected pattern of use. For example, requests for a website's home page can comprise 2/3rds of all requests, represented by $\alpha = 0.67$. If a request is for the 5th most popular file and five files have already been moved, the target is assumed to have been one of those moved already, and the request is simply forwarded to the new server. In all cases, the distribution of server location is known by deterministic algorithm to both clients and/or peer servers. Heartbeat messages within the cluster detect a change, e.g. a crash. Incoming requests are redistributed amongst the remaining servers within a cluster-user configurable period, usually minutes. The cost is the one-time re-caching.

3. Results

A range of performance and availability measures are defined for each of the clusters. Basic measures are directly derived from the expected marking of individual places in the cluster, e.g. $P_{service\_5k\_i}$, and the rates of its surrounding transitions, e.g. $T_{from\_service\_5k\_i}$ and $T_{service\_5k\_into\_reconfig}$. These measures are common to all clusters and are.parameterized chiefly by the number of clients and number of servers. Excepting the case of single server clusters, overall availability is effectively 100% for all the test cases presented because of the low failure rates and the independence of the proxy servers themselves.

3.1 Utilization, Job Los and Throughput

Representative results for Utilization, Job loss and Throughput are shown in Figures 9-14. The rate of Job Loss due to service interruptions is the sum of due to software or hardware failures is the sum of the rate of all failure transitions from all in-service positions. The rate of job loss due non-removal from DNS represents the number of jobs lost prior to a down server being removed form service.

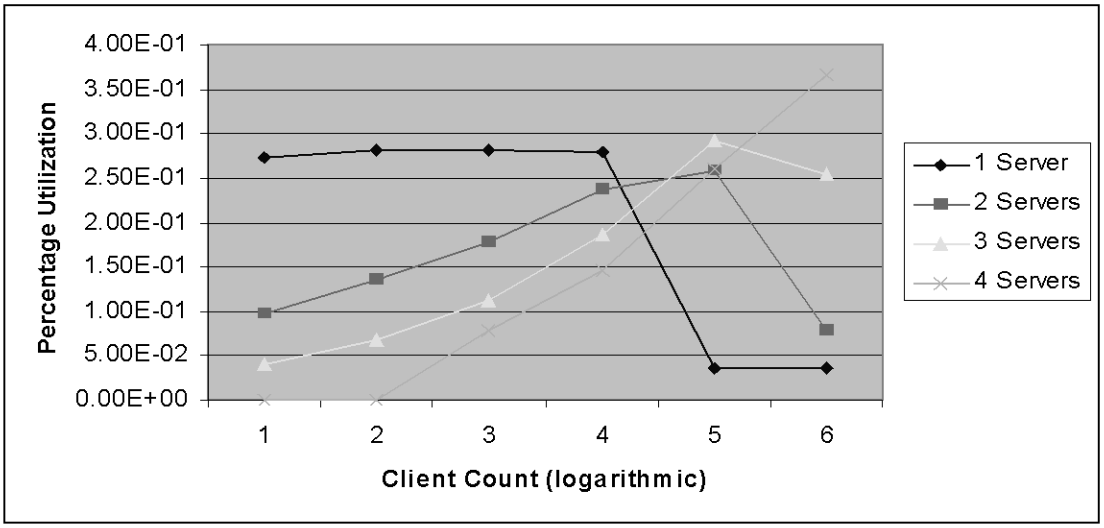


Figure 9 Utilization – Round Robin DNS

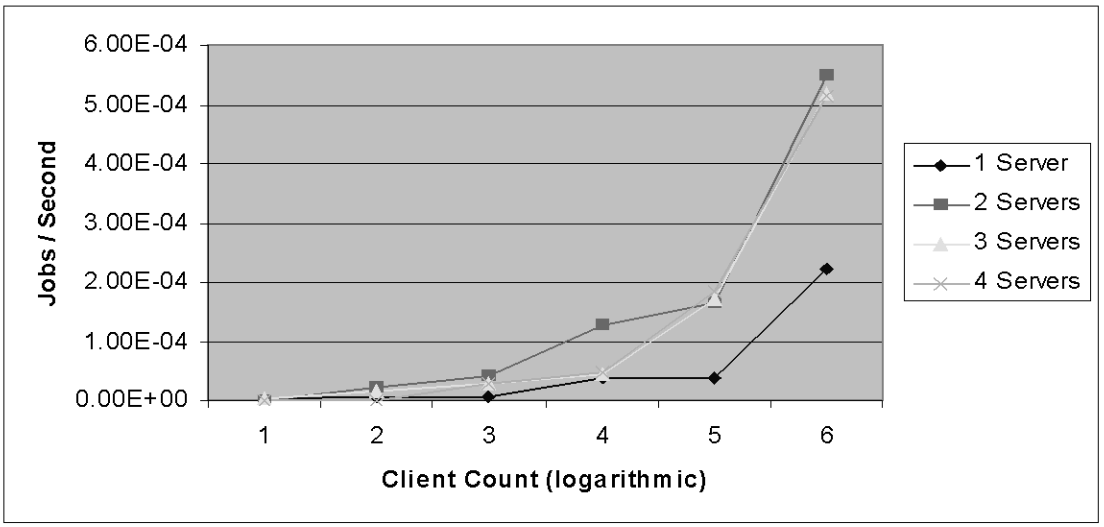


Figure 10 Rate of Lost jobs due to non-removal – Round Robin DNS

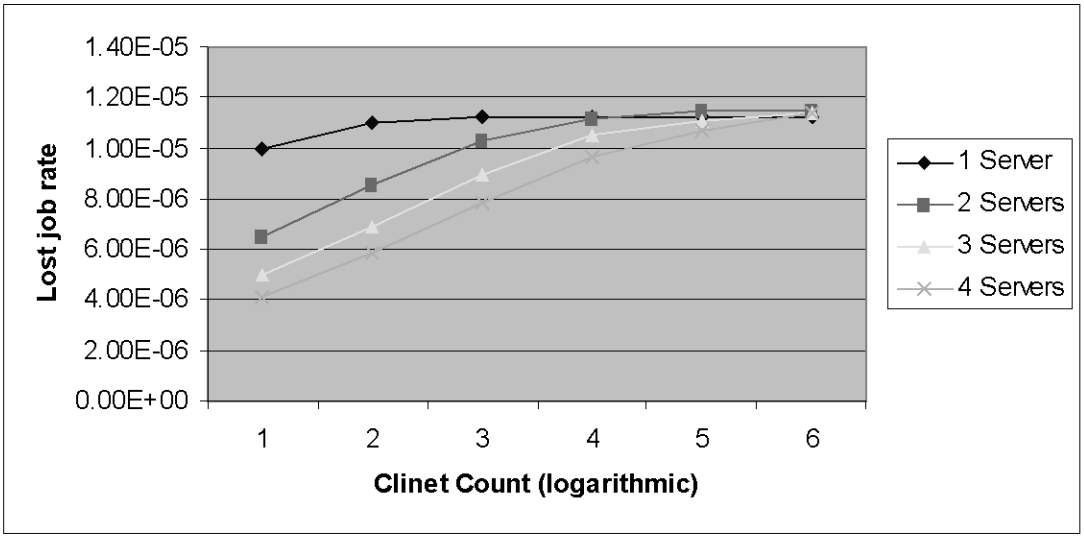


Figure 11 Rate of Lost jobs due to Service Interruption – Shared File Server Configuration

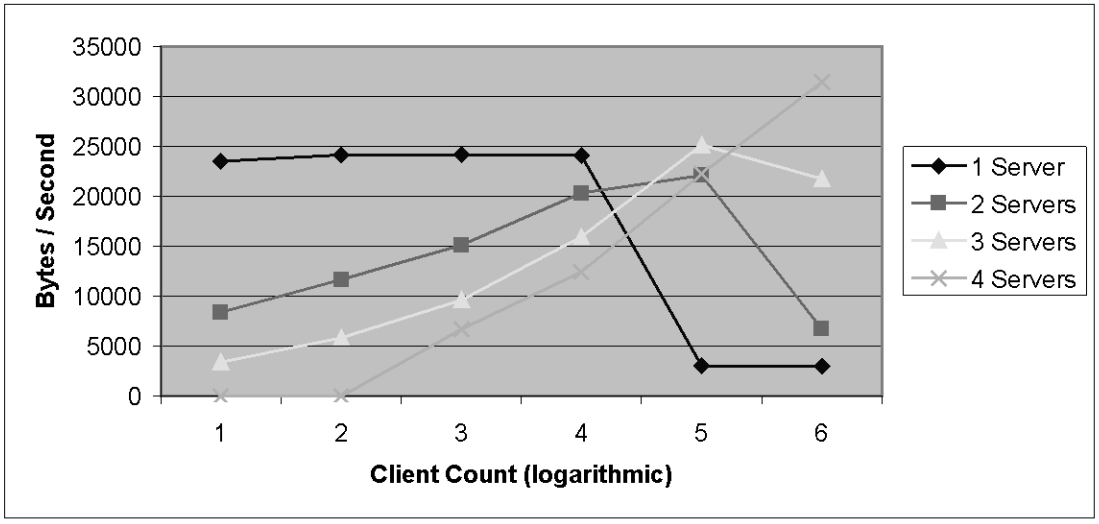


Figure 12 Throughput - Round Robin DNS configuration

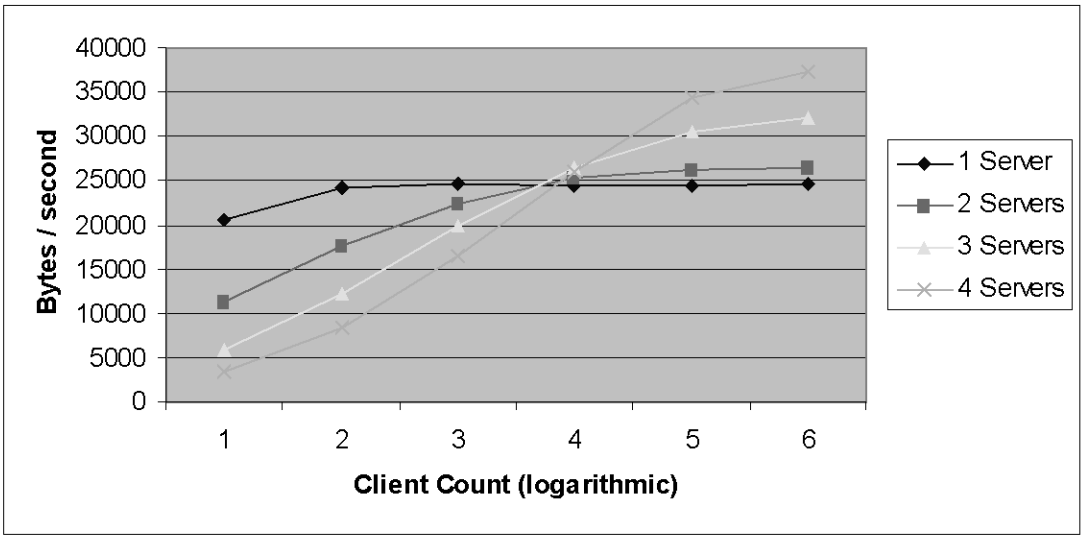


Figure 13 Throughput - Shared File Server configuration

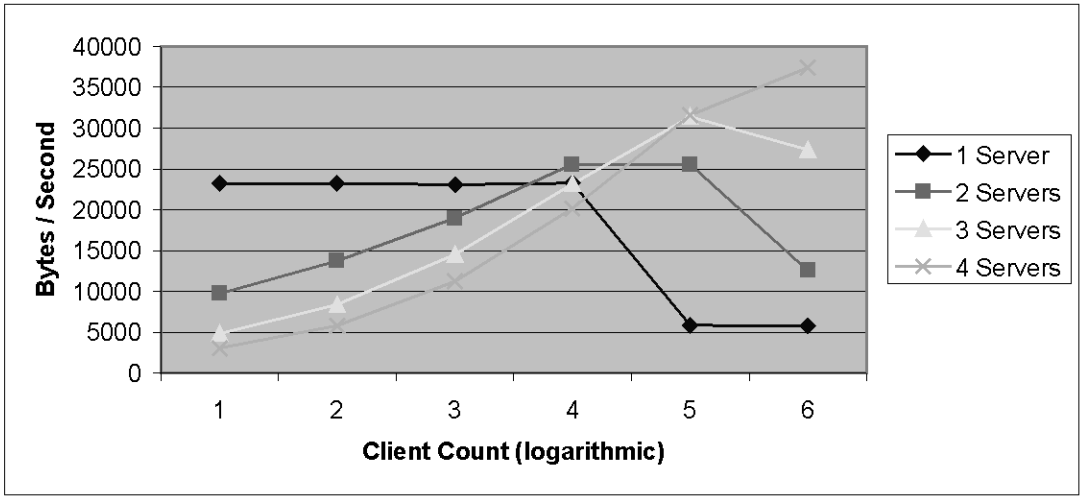


Figure 14 Throughput - Shared File Server configuration

3.2 Capacity Oriented Availability

An upper bound to the performance of a cluster is provided by its capacity, measured as the number of jobs admitted to service when there are finite queues. Figures 12 through 14 compare the results for all three configurations. Since the Shared File Server represents a shared queuing resource, the jobs waiting in buffers at each server effectively form a single queue, whereas the per server queues in the CARP configuration operate independently. All show no loss with fewer clients, which corresponds to the selected parameterization of 16 buffers, the maximum number of jobs admitted per server.

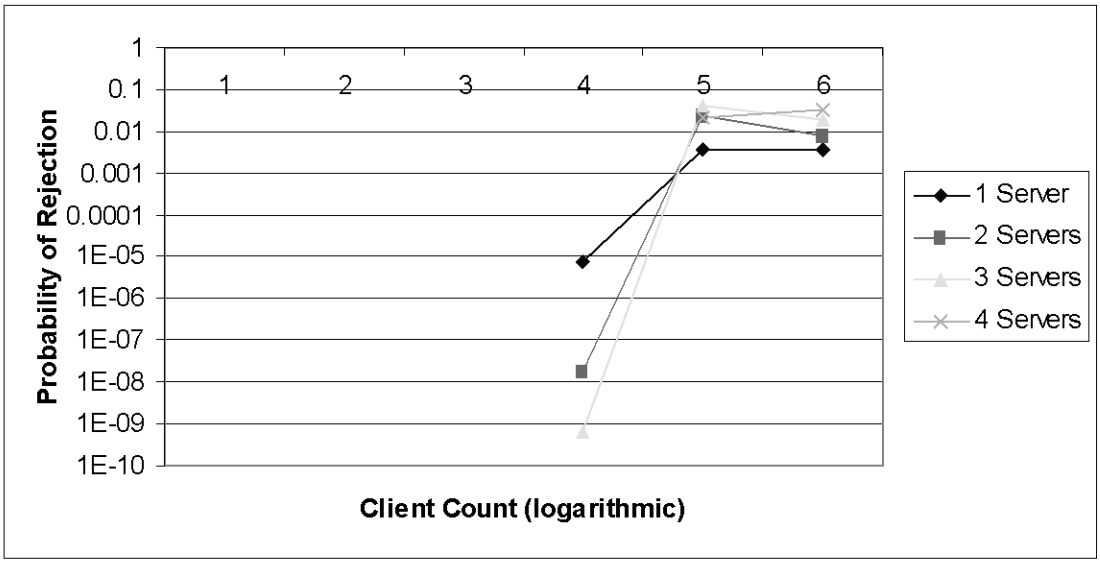


Figure 12 Capacity Oriented Availability – Round Robin DNS

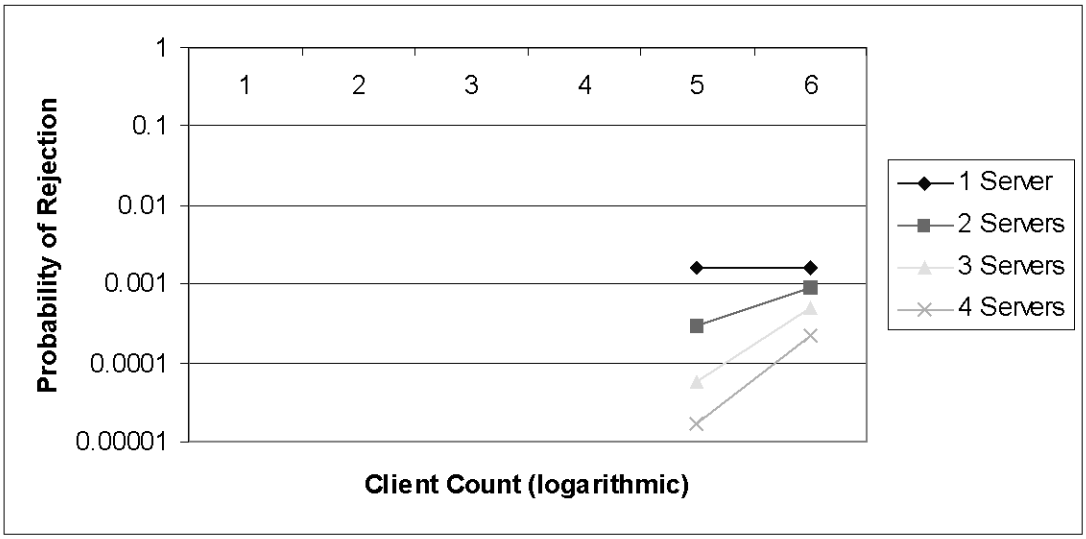


Figure 13 Capacity Oriented Availability – Shared File Service

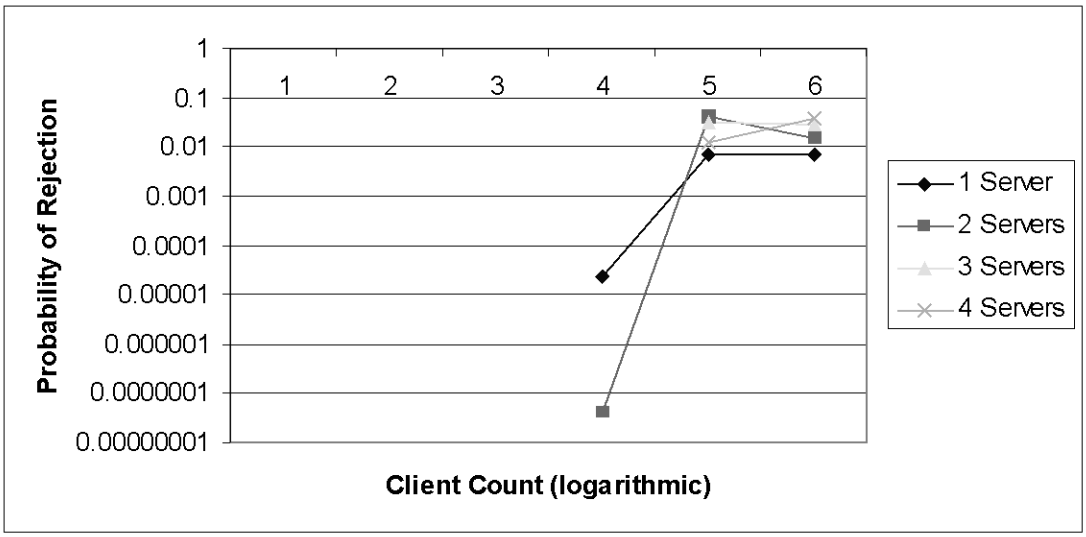


Figure 14 Capacity Oriented Availability - CARP

3.3 Performance Oriented Availability

As the number of jobs admitted to service increases, so does the likelihood that delay induced by their queuing will exceed user tolerance. Figures 15 through 17 show results when parameterized to allow 16 concurrent jobs and maximum delay of 8 seconds. The probability of delay is derived from the degenerate case of the general formula for probability of a specific delay with parallel queues and truncation[3]. For each marking of the queue, $reward = \sum_{i=0}^{n} \frac{(\mu t)^i e^{-\mu}}{i!}$ where n is the number of jobs, t is delay threshold and μ is the average service rate.

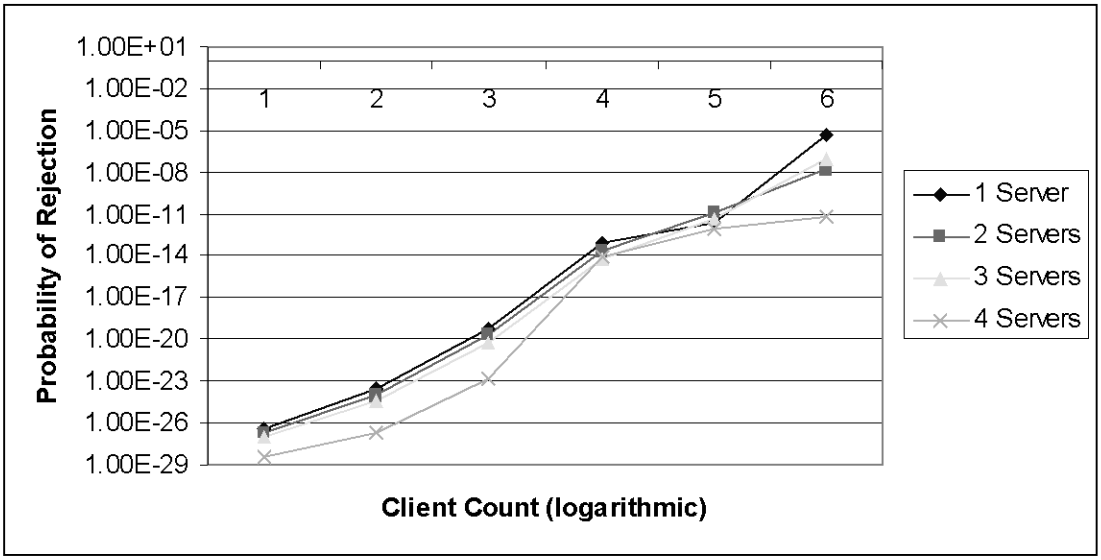


Figure 15 Performance Oriented Availability – Round Robin DNS

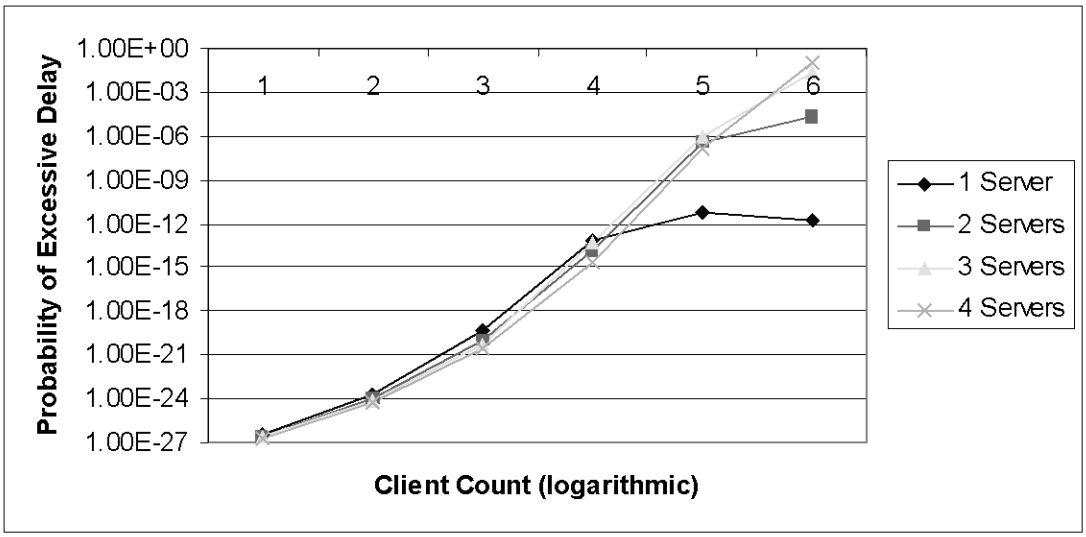


Figure 16 Performance Oriented Availability - Shared File Server configuration

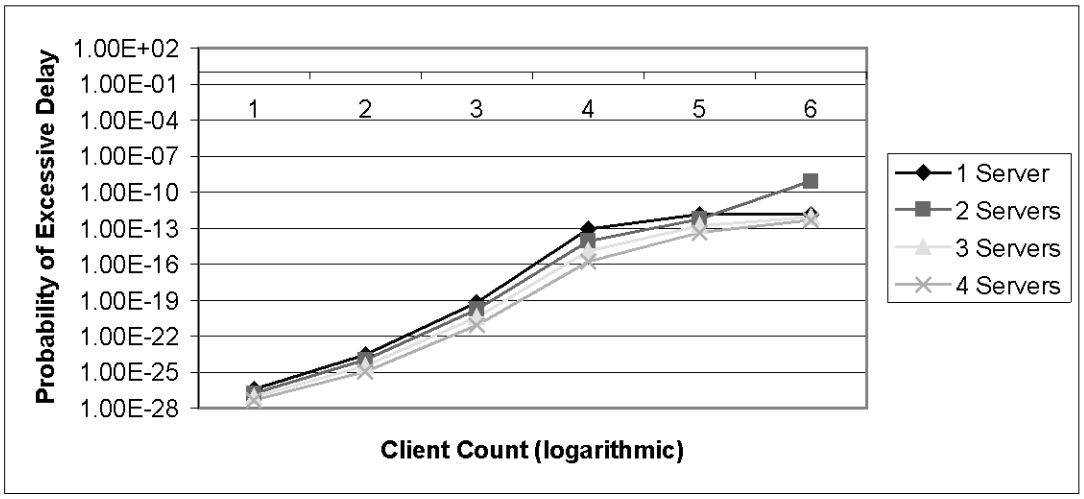


Figure 17 Performance Oriented Availability - CARP

Again we see that because the Shared File Server represents a shared queuing resource amongst all servers, the probability of excessive delay is higher overall than in either the CARP or Round Robin configurations. Similarly, the probability of degraded performance is greater when there are more servers in the Shared File Server configuration.

The inter-relationship between capacity and performance measures is also noteworthy in cases when the system is heavily loaded. Particularly, the Shared File Server configuration suffers most from degraded performance, as compared to the CARP cluster, which simply rejects more jobs.

4. Conclusions and Future Research

In this paper we develop a method to compare webserver configurations with respect to performance and availability. Our models provide a generalized real-time laboratory environment to quickly predict values for the long-term effect failure on performance and the effect of congestion on performance as it in turn effects the user perception of availability. These measures are used to explore the effects of two key design variables in webserver configuration. Particularly, the effect of physical hierarchy in the cluster configuration as given by the model of a Round Robin cluster serviced by a shared File Server was compared to servers with independent disks. The effects of two different routing and job scheduling protocols were compared in the Round-Robin DNS and Cache Array Routing Protocol clusters.

In each of the three configurations we confirm some expected results. Particularly, throughput and the probability of in-service job interruption are shown to increase with the number of servers in all configurations, together with a corresponding decrease in utilization. However, because of the low failure rates and the characteristic independence of the Proxy servers themselves, cluster-wide availability - the probability that at least one server is functioning - was seen to be virtually continuous for all configurations. In order to demonstrate finer grain availability and performance measures more idiomatic to web-servers, other parameters including the per job-class service rate, the number of buffers, the number of processors per server and the acceptable delay threshold are deliberately held constant throughout. This enables Performance and Capacity oriented availability measures to be defined and examined with respect to a varying number of clients and servers. Under this set of parameterizations, a clear tradeoff is seen between greater availability and an increased probability of unacceptable performance – the case with the Shared File server configuration and its effectively larger global queue - as compared to the less available configuration but less unacceptably delayed - the CARP and Round-Robin configurations with their distributed storage. The effect of CARP's dynamic re-caching strategies on the probability of job delay is also contrasted with the much greater probability of job loss when relying upon simple removal from DNS to effect service fail-over.

Future research directions on this work are possible in two areas: (1) the configurations; and (2) the modeling technique. The configurations can be extended to heterogeneous clusters combined to form mirrored, geographically distributed networks, and incorporate physical redundancy mechanisms such as redundant DNS servers, multiple switches in a LAN accessing a remote File Server and dual ported disks in local proxies. We are exploring techniques to extend the models to reduce the complexity of the state space explosion by using modular decomposition and Fluid Stochastic Petri Nets[3]. Either will allow an increased number of active concurrent clients. Particularly, we expect that by allowing the replacement of integral marking with fractional values at sink and source places surrounding timed transitions, FSPNs could directly extend the current Discrete Event simulation to a more numerically tractable Continuous event simulation, with significantly reduced run times.

References

[1] Paul Albiz and Cricket Liu, "DNS and BIND", 3rd ed., c.1998 O'Reilly & Associates, Inc., Sebastopol, CA.

[2] Gianfranco Ciardo, "Analysis of large stochastic Petri Net models: Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University", 1989.

[3] Donald Gross and Carl M. Harris, "Fundamentals of Queuing Theory (Wiley Series in Probability and Mathematical Statistics), 3rd ed., c. December 1997, John Wiley & Sons, 464 p., ISBN: 0471170836.

[4] Graham Horton, Vidyadhar G. Kulkarni, David M. Nicol and Kishor S. Trivedi, "Fluid Stochastic Petri nets: Theory, Applications and solution techniques", c. 1993, source unknown.

[5] Oliver C. Ibe, Kishor S. Trivedi and Archana Sathaye, "Stochastic Petri net modeling of VAXcluster system availability", Appeared in: International Conference of Fault Tolerant Systems and Diagnostics, Varna, Bulgaria, June 20-22 1990, c. 1988 Digital Equipment Corporation; Andover MA.

[6] Ari Luotonen, "Web Proxy Servers", c. 1998 Netscape Communications Corporation, Prentice Hall PTR, 400 p., ISBN: 0136806120.

[7] Daniel A. Menasce and Virgilio A. F. Almeida, "Capacity Planning for Web Performance : Metrics, Models, and Methods", c. June 1998, Prentice Hall, 450 p., ISBN: 0136938221.

[8] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", In: Proceedings of the IEEE, Vol. 77, no. 4, April 1989.

[9] Robin Sahner, Kishor Trivedi and Antonio Puliafito, "Performance and Reliability Analysis of Computer Systems: An Example Based Approach using the SHARPE software package", c. 1996, Kluwer Academic Publishers, New York, NY.

[10] Kishor Trivedi, "SPNP Version 6.1", Software, c. March 2000, Duke University;

[11] Kishor Trivedi, "SPNP User's Manual Version 6.0", c. September 1999, Duke University.

[12] Kishor S. Trivedi, Archana S. Sathaye, Oliver C. Ibe and Richard C. Howe, "Should I add a processor?", In: Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, c. 1990, IEEE Computer Society Press.

[13] Cache Array Routing Protocol (CARP) v1.0 Specifications c. 1998 Microsoft. [An Internet Draft] http://www.linofee.org/~elkner/da/papers/CarpSpec.htm

[14] Cache Array Routing Protocol and MS Proxy Server version 2.0, White Paper, http://www.microsoft.com/technet/Proxy/technote/prxcarp.asp, c.1997 Microsoft Corporation.

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol", http://www.cs.wisc.edu/~cao/papers/summary-cache/, c. 1998.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", Network Working Group, Request for Comments #2068, Standards Track, January 1997.

[16] Domain Names – Concepts and Facilities, Network Working Group Request for Comments: 1034, P. Mockapetris, ISI November 1997.

[17] Domain Names – Implementation and Specification, Network Working Group Request for Comments: 1035, P. Mockapetris, ISI November 1997.

[18] DNS Support for Load Balancing, Network Working Group Request for Comments: 1794, T. Brisco, Rutgers University, April 1995.

[19] Zipf Curves and Website Popularity , Jakob Nielsen's *Alertbox*, April 15,1997, http://www.zdnet.com/devhead/alertbox/zipf.html

[20] Geof Pawlicki, Archana Sathaye, Kishor Trivedi, "Performance and Availability Modeling of Webserver Configurations" (with appendixes), http://www.mathcs.sjsu.edu/faculty/sathaye/IFIP_1_10.pdf

Appendix A. Round Robin DNS SRN listing

Round Robin DNS per-server User Configurable Constants	
Name	Value
lambda (cluster wide)	16
mu_DNS	1000 // fast enough not to be a problem
lambda_fault	1.0 / (24* 60 * 60); // One per day
covered_fault_percentage	5.0 / 6.0
lambda_fail	1.0 / 60.0 ; // 1 minutes to swap out
mu_reconfig	0.1; // 10 seconds to reconfigure, e.g. restart a thread
mu_reboot	1.0 / (5 * 60); // 12 per hour = 5 minutes to reboot
mu_repair	1.0 / (60 * 60); // 1 per hour = 1 hour to repair/replace
server_count	[1..5]
client_count	[1..100]
processor_count	[1..2] DEFAULT: 1
buffer_count	[1..100] DEFAULT: 16
DEFAULT_PERCENTAGE_5K	0.25
DEFAULT_PERCENTAGE_10K	0.30
DEFAULT_PERCENTAGE_38K	0.19
DEFAULT_PERCENTAGE_350K	0.01
DEFAULT_PERCENTAGE_CGI	0.25
PERCENTAGE_CACHE_HIT	0.99
DEFAULT_SERVICE_RATE_MULIPLIER	1.0
rate_5k	14.306
rate_10k	13.793
rate_38k	2.0329
rate_350k	0.2263
rate_CGI	2.857
rate_origin_5k	1.0
rate_origin_10k	0.5
rate_origin_38k	0.25
rate_origin_350k	0.125

Round Robin DNS Transitions			
Name	Rate	Probability	Enabling Function
t_into_DNS	lambda		
t_from_DNS	mu_DNS		

t_into_service_CGI	rate_into_service_CGI		
t_from_service_CGI	rate_from_service_CGI		
t_into_i		1.0 / (float) server_count	(#(p_down_no_DNS_i = 0)
t_overflow_i		1.0	(#(p_entry_i) = buffer_count) and (#(p_down_i) = 0)
t_into_entry_i		1.0	(#(p_entry_i)
t_lostjob_i	1000		(#(p_down_i) 0)
t_into_service_i	1000		(#(p_ready_i) 0) and (#(p_entry_i) 0) and (#(p_down_i) = 0)
t_switch_service_miss_i		1 - PERCENTAGE_CACHE_HIT	
t_into_service_miss_i		1000	
t_into_service_miss_5k_i		DEFAULT_PERCENTAGE_5k	
t_into_service_miss_10k_i		DEFAULT_PERCENTAGE_10k	
t_into_service_miss_38k_i		DEFAULT_PERCENTAGE_38k	
t_into_service_miss_350k_i		DEFAULT_PERCENTAGE_350k	
t_from_service_miss_5k_i	rate_origin_5k		
t_from_service_miss_10k_i	rate_origin_10k		
t_from_service_miss_38k_i	rate_origin_38k		
t_from_service_miss_350k_i	rate_origin_350k		
t_ready_into_reconfig_i	lambda_fault*covered_fault_percentage		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reconfig_i	mu_reconfig		(#(p_ready_reconfig) 0) and (#(p_down_i) 0)
t_from_ready_reconfig_down_no_DNS_i	mu_repair		(#(p_ready_reconfig) 0) and (#(p_down_no_DNS) 0)
t_ready_into_reboot_i	lambda_fault*covered_fault_percentage		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reboot_i	mu_reboot		(#(p_ready_reboot) 0) and (#(p_down_i) 0)
t_from_ready_reboot_down_no_DNS_i	mu_repair		(#(p_ready_reboot) 0) and (#(p_down_no_DNS) 0)
t_switch_service_hit_i		PERCENTA	

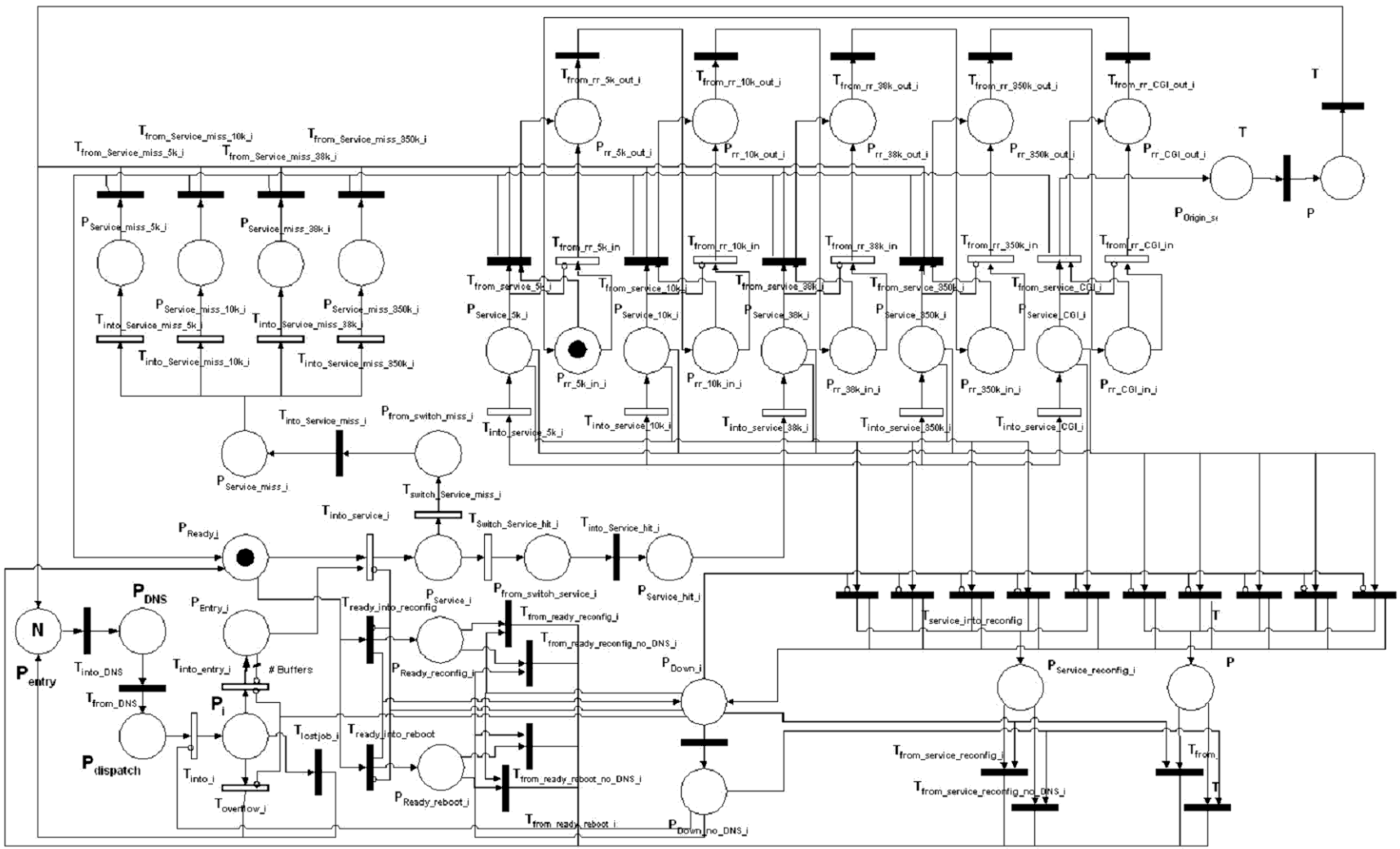
		GE_CACHE_HIT	
t_into_service_hit_i	1000		
t_into_service_5k_i	DEFAULT_PERCENTAGE_5k		
t_into_service_10k_i	DEFAULT_PERCENTAGE_10k		
t_into_service_38k_i	DEFAULT_PERCENTAGE_38k		
t_into_service_350k_i	DEFAULT_PERCENTAGE_350k		
t_into_service_CGI_i	DEFAULT_PERCENTAGE_CGI		
t_from_service_5k_i	rate_5k * MULTIPIERS[i]		(#(p_service_5k_i) 0) and (#(p_rr_5k_in_i) 0)
t_from_service_10k_i	rate_10k * MULTIPIERS[i]		(#(p_service_10k_i) 0) and (#(p_rr_10k_in_i) 0)
t_from_service_38k_i	rate_38k * MULTIPIERS[i]		(#(p_service_38k_i) 0) and (#(p_rr_38k_in_i) 0)
t_from_service_350k_i	rate_350k * MULTIPIERS[i]		(#(p_service_350k_i) 0) and (#(p_rr_350k_in_i) 0)
t_from_service_CGI_i		1.0	(#(p_service_CGI_i) 0) and (#(p_rr_CGI_in_i) 0)
t_from_rr_5k_in_i		1.0	(#(p_service_5k_i) = 0) and (#(p_rr_5k_in_i) 0)
t_from_rr_10k_in_i		1.0	(#(p_service_10k_i) = 0) and (#(p_rr_10k_in_i) 0)
t_from_rr_38k_in_i		1.0	(#(p_service_38k_i) = 0) and (#(p_rr_38k_in_i) 0)
t_from_rr_350k_in_i		1.0	(#(p_service_350k_i) = 0) and (#(p_rr_350k_in_i) 0)
t_from_rr_CGI_in_i		1.0	(#(p_service_CGI_i) = 0) and (#(p_rr_CGI_in_i) 0)
t_from_rr_5k_out_i	10		
t_from_rr_10k_out_i	10		
t_from_rr_38k_out_i	10		
t_from_rr_350k_out_i	10		
t_from_rr_CGI_out_i	10		
t_into_service_CGI_i	100		
t_from_service_CGI_i	rate_CGI		
t_service_into_reconfig_5k_i	(lambda_fault / percentage_5k[i]) * covered_fault_percentage		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_10k_i	(lambda_fault / percentage_10k[i]) *		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)

	covered_fault_percentage		
t_service_into_reconfig_38k_i	(lambda_fault / percentage_38k[i]) * covered_fault_percentage		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_350k_i	(lambda_fault / percentage_350k[i]) * covered_fault_percentage		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_5k_i	(lambda_fault / percentage_5k[i]) * (1-covered_fault_percentage)		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_10k_i	(lambda_fault / percentage_10k[i]) * (1-covered_fault_percentage)		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_38k_i	(lambda_fault / percentage_38k[i]) * (1-covered_fault_percentage)		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_350k_i	(lambda_fault / percentage_350k[i]) * (1-covered_fault_percentage)		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_from_service_reconfig_i	mu_reconfig		(#(p_service_reconfig) 0) and (#(p_down_i) 0)
t_from_service_reconfig_down_no_DNS_i	mu_repair		(#(p_service_reconfig) 0) and (#(p_down_no_DNS) 0)
t_from_service_reboot_i	mu_reboot		(#(p_service_reboot) 0) and (#(p_down_i) 0)
t_from_service_reboot_down_no_DNS_i	mu_repair		(#(p_service_reboot) 0) and (#(p_down_no_DNS) 0)

Round Robin DNS Arcs	
Name	Arc Multiplicity
t_into_entry_i	mharc: buffer_count

Round Robin DNS Proxy Cluster

DNS queuing and remove/restore; Job Size Classes; CGI service on Origin Server; In Service and Ready Failure; Lost Jobs by Buffer Overflow and Down server



Appendix B. Shared File Server SRN listing

Shared File Server per-server User Configurable Constants	
Name	Value
lambda (cluster wide)	16
mu_DNS	1000 // fast enough not to be a problem
lambda_fault	1.0 / (24* 60 * 60); // One per day
covered_fault_percentage	5.0 / 6.0
lambda_fail	1.0 / 60.0 ; // 1 minutes to swap out
mu_reconfig	0.1; // 10 seconds to reconfigure, e.g. restart a thread
mu_reboot	1.0 / (5 * 60); // 12 per hour = 5 minutes to reboot
mu_repair	1.0 / (60 * 60); // 1 per hour = 1 hour to repair/replace
server_count	[1..5]
disk_count	[1..5]
client_count	[1..100]
processor_count	[1..2] DEFAULT: 1
buffer_count	[1..100] DEFAULT: 16
DEFAULT_PERCENTAGE_5K	0.25
DEFAULT_PERCENTAGE_10K	0.30
DEFAULT_PERCENTAGE_38K	0.19
DEFAULT_PERCENTAGE_350K	0.01
DEFAULT_PERCENTAGE_CGI	0.25
PERCENTAGE_CACHE_HIT	0.99
DEFAULT_MULIPLIER	1.0 // for heterogenous disk rates
disk_rate_5k	14.306
disk_rate_10k	13.793
disk_rate_38k	2.0329
disk_rate_350k	0.2263
rate_CGI	2.857
rate_origin_5k	1.0
rate_origin_10k	0.5
rate_origin_38k	0.25
rate_origin_350k	0.125

Shared File Server Transitions			
Name	Rate	Probability	Enabling Function
t_into_DNS	lambda		
t_from_DNS	mu_DNS		
t_into_i		1.0 / (float)	(#(p_down_no_DNS_i = 0)

		server_count	
t_overflow_i		1.0	(#(p_entry_i) = buffer_count) and (#(p_down_i) = 0)
t_into_entry_i		1.0	(#(p_entry_i)
t_lostjob_i	1000		(#(p_down_i) 0)
t_into_service_i	1000		(#(p_ready_i) 0) and (#(p_entry_i) 0) and (#(p_down_i) = 0)
t_to_origin_server_i		DEFAULT_PERCENTAGE_CGI	
t_into_service_CGI_i	1000		
t_from_service_CGI	rate_CGI		
t_to_service_miss_i		(1 - DEFAULT_PERCENTAGE_CGI) * (1 - PERCENTAGE_CACHE_HIT)	
t_into_service_miss_i	1000		
t_into_service_miss_5k_i		DEFAULT_PERCENTAGE_5k	
t_into_service_miss_10k_i		DEFAULT_PERCENTAGE_10k	
t_into_service_miss_38k_i		DEFAULT_PERCENTAGE_38k	
t_into_service_miss_350k_i		DEFAULT_PERCENTAGE_350k	
t_from_service_miss_5k_i	rate_origin_5k		
t_from_service_miss_10k_i	rate_origin_10k		
t_from_service_miss_38k_i	rate_origin_38k		
t_from_service_miss_350k_i	rate_origin_350k		
t_ready_into_reconfig_i	lambda_fault*covered_fault_percentage		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reconfig_i	mu_reconfig		(#(p_ready_reconfig) 0) and (#(p_down_i) 0)
t_from_ready_reconfig_down_no_DNS_i	mu_repair		(#(p_ready_reconfig) 0) and (#(p_down_no_DNS) 0)
t_ready_into_reboot_i	mu_reboot		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reboot_i	mu_repair		(#(p_ready_reboot) 0) and (#(p_down_i) 0)

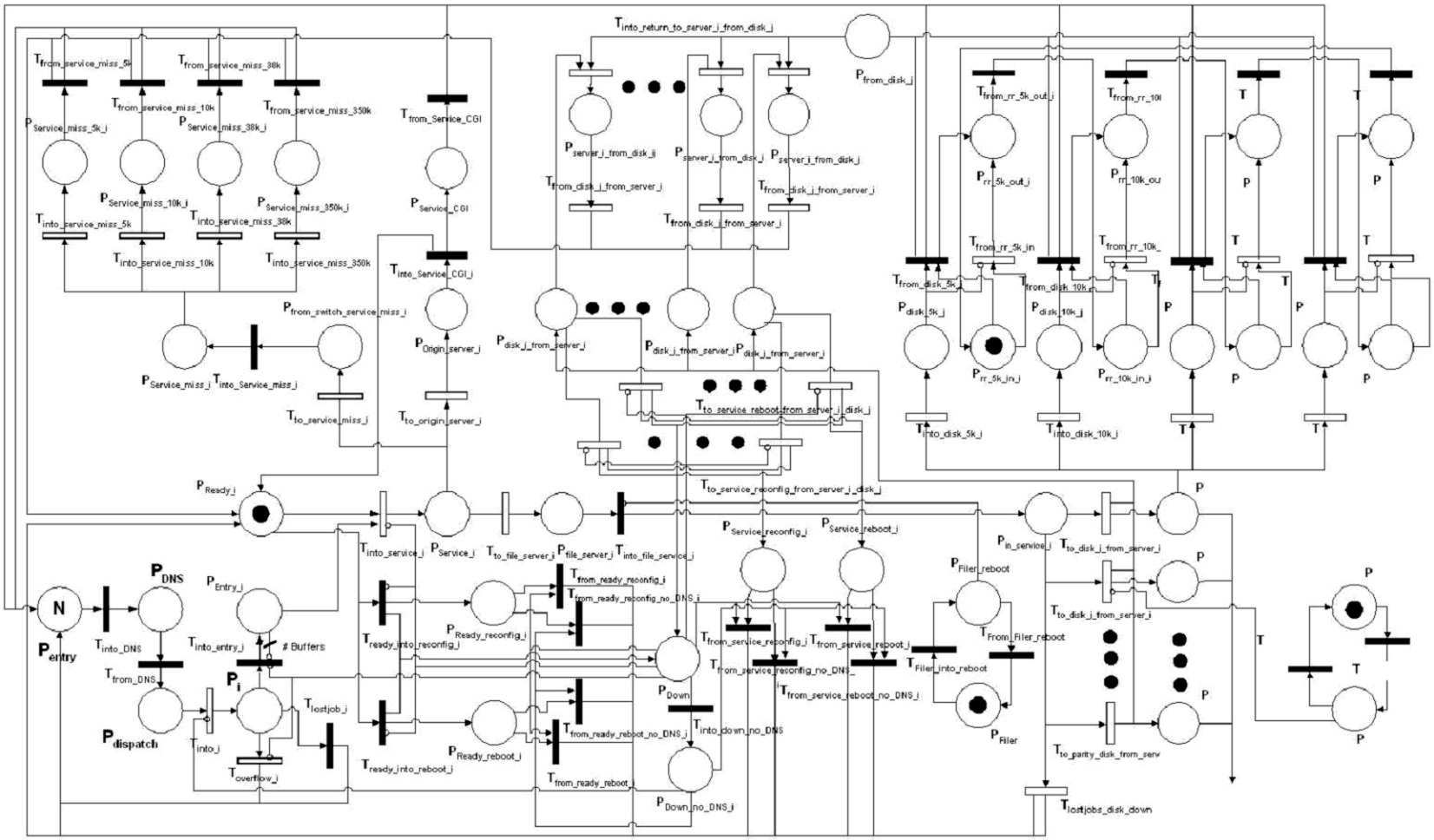
t_from_ready_reboot_down_no_DNS_i	lambda_fault*covered_fault_percentage		(#(p_ready_reboot) 0) and (#(p_down_no_DNS) 0)
t_to_file_server_i		(1 - DEFAULT_PERCENTAGE_CGI) * PERCENTAGE_CACHE_HIT	
t_into_file_service_i	1000		(#(p_to_file_server_i) 0) and (#(p_filer_reboot) = 0)
t_filer_into_reboot	1.0 / (60 * 60 * 24 * 7) // reboot once a week		
t_from_filer_reboot	1/(2*60) // 2 minutes to reboot		
t_to_disk_j_from_server_i		1 /(disk_count + 1)	(#(p_in_service_i) 0) and (#(p_disk_reconfig_j) = 0)
t_lostjobs_disk_down		1.0	(#(p_in_service_i) 0) and (COUNT(#(p_disk_reconfig_j)=1) 1)
t_disk_into_reconfig_j	lambda_disk_fault		
t_from_disk_reconfig_j	1.0 / (60 * 30) // half and hour to replace a down disk		
t_into_disk_5k_j	DEFAULT_PERCENTAGE_5k		
t_into_disk_10k_j	DEFAULT_PERCENTAGE_10k		
t_into_disk_38k_j	DEFAULT_PERCENTAGE_38k		
t_into_disk_350k_j	DEFAULT_PERCENTAGE_350k		
t_from_disk_5k_j	disk_rate_5k * disk_multipliers[j]		(#(p_disk_5k_j) 0) and (#(p_rr_5k_in_j) 0)
t_from_disk_10k_j	disk_rate_10k * disk_multipliers[j]		(#(p_disk_10k_j) 0) and (#(p_rr_10k_in_j) 0)
t_from_disk_38k_j	disk_rate_38k * disk_multipliers[j]		(#(p_disk_38k_j) 0) and (#(p_rr_38k_in_j) 0)
t_from_disk_350k_j	disk_rate_350k * disk_multipliers[j]		(#(p_disk_350k_j) 0) and (#(p_rr_350k_in_j) 0)
t_from_rr_5k_in_i		1.0	(#(p_disk_5k_i) = 0) and (#(p_rr_5k_in_i) 0)
t_from_rr_10k_in_i		1.0	(#(p_disk_10k_i) = 0) and (#(p_rr_10k_in_i) 0)
t_from_rr_38k_in_i		1.0	(#(p_disk_38k_i) = 0) and (#(p_rr_38k_in_i) 0)

t_from_rr_350k_in_i		1.0	(#(p_disk_350k_i) = 0) and (#(p_rr_350k_in_i) 0)
t_from_rr_5k_out_i	10		
t_from_rr_10k_out_i	10		
t_from_rr_38k_out_i	10		
t_from_rr_350k_out_i	10		
t_into_return_to_server_i_from_disk_j		1 / server_count	
t_from_disk_j_from_server_i		1.0	(#(p_server_i_from_disk_j) 0) and (#(p_disk_j_from_server_i) 0)
t_to_service_reboot_from_server_i_disk_j	lambda_fault * (1 - covered_fault_percentage)		(#(p_disk_j_from_server_i) 0) and (#(p_down_i) = 0)
t_to_service_reboot_from_server_i_disk_j	lambda_fault * covered_fault_percentage		(#(p_disk_j_from_server_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_5k_i	(lambda_fault / percentage_5k[i]) * covered_fault_percentage		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_10k_i	(lambda_fault / percentage_10k[i]) * covered_fault_percentage		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_38k_i	(lambda_fault / percentage_38k[i]) * covered_fault_percentage		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_350k_i	(lambda_fault / percentage_350k[i]) * covered_fault_percentage		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_5k_i	(lambda_fault / percentage_5k[i]) * (1- covered_fault_percentage)		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_10k_i	(lambda_fault / percentage_10k[i]) * (1- covered_fault_percentage)		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_38k_i	(lambda_fault / percentage_38k[i]) * (1- covered_fault_percentage)		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)

t_service_into_reboot_350k_i	(lambda_fault / percentage_350k[i]) * (1-covered_fault_percentage)		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_from_service_reconfig_i	mu_reconfig		(#(p_service_reconfig) 0) and (#(p_down_i) 0)
t_from_service_reconfig_down_no_DNS_i	mu_repair		(#(p_service_reconfig) 0) and (#(p_down_no_DNS) 0)
t_from_service_reboot_i	mu_reboot		(#(p_service_reboot) 0) and (#(p_down_i) 0)
t_from_service_reboot_down_no_DNS_i	mu_repair		(#(p_service_reboot) 0) and (#(p_down_no_DNS) 0)

Shared File Server Arcs	
Name	Arc Multiplicity
t_into_entry_i	mharc: buffer_count

Heterogeneous Cluster with Shared File Server
Job Classes, Cache Miss, CGI service on Origin Server,



Appendix C. CARP SRN listing

CARP per-server User Configurable Constants	
Name	Value
lambda (cluster wide)	16
lambda_fault	1.0 / (24* 60 * 60); // One per day
covered_fault_percentage	5.0 / 6.0
lambda_fail	1.0 / 60.0 ; // 1 minutes to swap a bad server out of cluster
mu_reconfig	0.1; // 10 seconds to reconfigure, e.g. restart a thread
mu_reboot	1.0 / (5 * 60); // 12 per hour = 5 minutes to reboot
mu_repair	1.0 / (60 * 60); // 1 per hour = 1 hour to repair/replace
server_count	[1..5]
client_count	[1..100]
processor_count	[1..2] DEFAULT: 1
buffer_count	[1..100] DEFAULT: 16
file_count	[1..100] DEFAULT: 100
DEFAULT_PERCENTAGE_5K	0.25
DEFAULT_PERCENTAGE_10K	0.30
DEFAULT_PERCENTAGE_38K	0.19
DEFAULT_PERCENTAGE_350K	0.01
DEFAULT_PERCENTAGE_CGI	0.25
PERCENTAGE_CACHE_HIT	0.99
DEFAULT_SERVICE_RATE_MULIPLIER	1.0
rate_5k	14.306
rate_10k	13.793
rate_38k	2.0329
rate_350k	0.2263
rate_CGI	2.857
rate_origin_5k	1.0
rate_origin_10k	0.5
rate_origin_38k	0.25
rate_origin_350k	0.125

CARP Transitions			
Name	Rate	Probability	Enabling Function
t_arrive	lambda		
t_into_i		1.0 / (float)	(#(p_removed_i = 0)

		server_count	
t_into_service_CGI	rate_into_service_CGI		
t_from_service_CGI	rate_from_service_CGI		
t_overflow_i		1.0	(#(p_entry_i) = buffer_count) and (#(p_down_i) = 0) and (#(p_removed_i) = 0)
t_into_entry_i		1.0	(#(p_entry_i)
t_lostjob_i	1000		(#(p_down_i) 0)
t_into_down_request_i		1.0	(#(p_removed_i) 0)
t_into_migrate_i			(#(p_file_count) 0) and (#(p_down_request) 0) and (zipf() <= (FILE_COUNT - (#(file_count)))
t_into_migrate_i_to_j		1/(servercount - 1)	
t_from_migrate_i_to_j	rate_migrate		(#(p_removed_j) = 0) and (#(t_migrate_i_to_j) 0)
t_into_file_moved_i			(#(p_down_request)) and (zipf() (FILE_COUNT - (#(p_file_count)))
t_into_file_moved_i_to_j		1/(servercount - 1)	(#(p_removed_j) = 0) and (#(t_file_moved_i_to_j) 0)
t_from_file_count_i	10		(#(p_file_count)) ((file_count / server_count) + 1)
t_from_file_moved_i_to_j	rate_forward		(#(p_down_request)) and (zipf() (FILE_COUNT - (#(p_file_count)))
t_into_service_i	1000		(#(p_ready_i) 0) and (#(p_entry_i) 0) and (#(p_down_i) = 0)
t_switch_service_miss_i		1 - PERCENTAGE_CACHE_HIT	
t_into_service_miss_i	1000		
t_into_service_miss_5k_i	DEFAULT_PERCENTAGE_5k		
t_into_service_miss_10k_i	DEFAULT_PERCENTAGE_10k		
t_into_service_miss_38k_i	DEFAULT_PERCENTAGE_38k		
t_into_service_miss_350k_i	DEFAULT_PERCENTAGE_350k		
t_from_service_miss_5k_i	rate_origin_5k		
t_from_service_miss_10k_i	rate_origin_10		

	k		
t_from_service_miss_38k_i	rate_origin_38k		
t_from_service_miss_350k_i	rate_origin_350k		
t_ready_into_reconfig_i	lambda_fault*covered_fault_percentage		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reconfig_i	mu_reconfig		(#(p_ready_reconfig) 0) and (#(p_down_i) 0)
t_from_ready_reconfig_removed_i	mu_repair		(#(p_ready_reconfig) 0) and (#(p_removed_i) 0)
t_ready_into_reboot_i	lambda_fault*covered_fault_percentage		(#(p_ready_i) 0) and (#(p_down_i) = 0)
t_from_ready_reboot_i	mu_reboot		(#(p_ready_reboot) 0) and (#(p_down_i) 0)
t_from_ready_reboot_removed_i	mu_repair		(#(p_ready_reboot) 0) and (#(p_removed_i) 0)
t_switch_service_hit_i		PERCENTAGE_CACHE_HIT	(#(p_migrated_into)= 0) and (#(p_service_i) 0)
t_clear_migrated_into_i		1.0	(#(p_migrated_into) 0) and (#(p_down_i) 0)
t_from_migrated_into_i		1.0	(#(p_migrated_into) 0) and (#(p_from_switch_service_miss_i) 0)
t_into_service_hit_i	1000		
t_into_service_5k_i		DEFAULT_PERCENTAGE_5k	
t_into_service_10k_i		DEFAULT_PERCENTAGE_10k	
t_into_service_38k_i		DEFAULT_PERCENTAGE_38k	
t_into_service_350k_i		DEFAULT_PERCENTAGE_350k	
t_into_service_CGI_i		DEFAULT_PERCENTAGE_CGI	
t_from_service_5k_i	rate_5k * MULTIPIERS[i]		(#(p_service_5k_i) 0) and (#(p_rr_5k_in_i) 0)
t_from_service_10k_i	rate_10k * MULTIPIERS[i]		(#(p_service_10k_i) 0) and (#(p_rr_10k_in_i) 0)

t_from_service_38k_i	rate_38k * MULTIPIERS[i]		(#(p_service_38k_i) 0) and (#(p_rr_38k_in_i) 0)
t_from_service_350k_i	rate_350k * MULTIPIERS[i]		(#(p_service_350k_i) 0) and (#(p_rr_350k_in_i) 0)
t_from_service_CGI_i		1.0	(#(p_service_CGI_i) 0) and (#(p_rr_CGI_in_i) 0)
t_from_rr_5k_in_i		1.0	(#(p_service_5k_i) = 0) and (#(p_rr_5k_in_i) 0)
t_from_rr_10k_in_i		1.0	(#(p_service_10k_i) = 0) and (#(p_rr_10k_in_i) 0)
t_from_rr_38k_in_i		1.0	(#(p_service_38k_i) = 0) and (#(p_rr_38k_in_i) 0)
t_from_rr_350k_in_i		1.0	(#(p_service_350k_i) = 0) and (#(p_rr_350k_in_i) 0)
t_from_rr_CGI_in_i		1.0	(#(p_service_CGI_i) = 0) and (#(p_rr_CGI_in_i) 0)
t_from_rr_5k_out_i	10		
t_from_rr_10k_out_i	10		
t_from_rr_38k_out_i	10		
t_from_rr_350k_out_i	10		
t_from_rr_CGI_out_i	10		
t_into_service_CGI_i	100		
t_from_service_CGI_i	rate_CGI		
t_service_into_reconfig_5k_i	(lambda_fault / percentage_5k[i]) * covered_fault_percentage		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_10k_i	(lambda_fault / percentage_10k[i]) * covered_fault_percentage		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_38k_i	(lambda_fault / percentage_38k[i]) * covered_fault_percentage		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)
t_service_into_reconfig_350k_i	(lambda_fault / percentage_350k[i]) * covered_fault_percentage		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_5k_i	(lambda_fault / percentage_5k[		(#(p_service_5k_i) 0) and (#(p_down_i) = 0)

	i]) * (1-covered_fault_percentage)		
t_service_into_reboot_10k_i	(lambda_fault / percentage_10k[i]) * (1-covered_fault_percentage)		(#(p_service_10k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_38k_i	(lambda_fault / percentage_38k[i]) * (1-covered_fault_percentage)		(#(p_service_38k_i) 0) and (#(p_down_i) = 0)
t_service_into_reboot_350k_i	(lambda_fault / percentage_350k[i]) * (1-covered_fault_percentage)		(#(p_service_350k_i) 0) and (#(p_down_i) = 0)
t_from_service_reconfig_i	mu_reconfig		(#(p_service_reconfig) 0) and (#(p_down_i) 0)
t_from_service_reconfig_removed_i	mu_repair		(#(p_service_reconfig) 0) and (#(p_removed) 0)
t_from_service_reboot_i	mu_reboot		(#(p_service_reboot) 0) and (#(p_down_i) 0)
t_from_service_reboot_removed_i	mu_repair		(#(p_service_reboot) 0) and (#(p_removed_i) 0)

CARP Arcs	
Name	Arc Multiplicity
t_into_entry_i	mharc: buffer_count

Cache Array Routing Protocol (CARP)

Failover with File Migration

Job Sizes