Cluster Load Balancing for Fine-grain Network Services

Kai Shen
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
kshen@cs.ucsb.edu

Tao Yang
Dept. of Computer Science
UC Santa Barbara
and Teoma/Ask Jeeves
tyang@cs.ucsb.edu

Lingkun Chu
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
lkchu@cs.ucsb.edu

Abstract

This paper studies cluster load balancing policies and system support for fine-grain network services. Load balancing on a cluster of machines has been studied extensively in the literature, mainly focusing on coarse-grain distributed computation. Fine-grain services introduce additional challenges because system states fluctuate rapidly for those services and system performance is highly sensitive to various overhead. The main contribution of our work is to identify effective load balancing schemes for fine-grain services through simulations and empirical evaluations on synthetic workload and real traces. Another contribution is the design and implementation of a load balancing system in a Linux cluster that strikes a balance between acquiring enough load information and minimizing system overhead. Our study concludes that: 1) Random polling based load-balancing policies are well-suited for fine-grain network services; 2) A small poll size provides sufficient information for load balancing, while an excessively large poll size may in fact degrade the performance due to polling overhead; 3) Discarding slow-responding polls can further improve system performance.

1. Introduction

Large-scale cluster-based network services are increasingly emerging to deliver highly scalable, available, and feature-rich user experiences. Inside those service clusters, a node can elect to provide services and it can also access services provided by other nodes. It serves as an internal *server* or *client* in each context respectively. Services are usually partitioned, replicated, aggregated, and then delivered to external clients through protocol gateways. Figure 1 illustrates the architecture of such a service cluster. In this example, the service cluster delivers a discussion group and a photo album service to wide-area browsers and wireless clients through web servers and WAP gateways. The discussion group service is delivered independently while the photo album service relies on an internal image store service. All the components (including protocol gateways) are replicated. In addition, the image store service is partitioned into two partition groups.

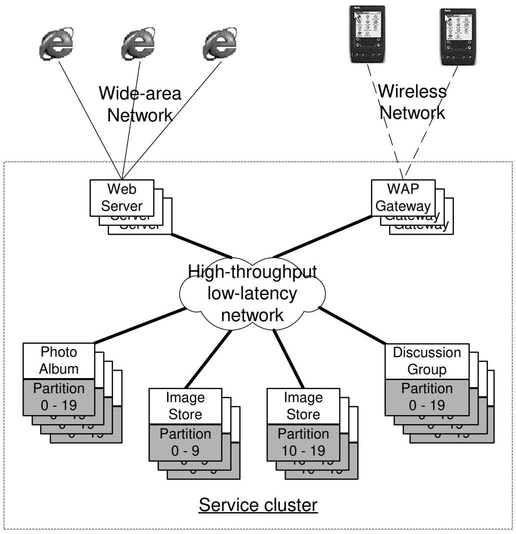

Figure 1. Architecture of a service cluster.

While previous research has addressed the issues of scalability, availability, extensibility, and service replication support in building large-scale network service infrastructures [2, 13, 15, 23], there is still a lack of comprehensive study on load balancing support in this context. This paper studies the issue of providing efficient load balancing support for accessing replicated services inside the service cluster. The request distribution between wide-area external clients and geographically distributed service clusters is out of the scope of this paper.

A large amount of work has been done by the industry and research community to optimize HTTP request distribution among a cluster of Web servers [1, 3, 6, 8, 17, 21].

Most load balancing policies proposed in such a context rely on the premise that all network packets go through a single front-end dispatcher or a TCP-aware (layer 4 or above) switch so that TCP level connection-based statistics can be accurately maintained. In contrast, clients and servers inside the service cluster are often connected by high-throughput, low-latency Ethernet (layer 2) or IP (layer 3) switches, which do not provide any TCP level traffic statistics. This constraint calls for more complex load information dissemination schemes.

Previous research has proposed and evaluated various load balancing policies for cluster-based distributed systems [5, 7, 9, 14, 19, 20, 24, 25]. Load balancing techniques in these studies are valuable in general, but not all of them can be applied for cluster-based network services. This is because they focus on coarse-grain computation and often ignore fine-grain jobs by simply processing them locally. For example, the job trace used in a previous trace-driven simulation study has a mean job execution time of 1.492 seconds [25]. In the context of network services, with the trend towards delivering more feature-rich services in real time, large number of fine-grain sub-services need to be aggregated within a short period of time. For example, a distributed hash table lookup for keyword search usually only takes a couple of milliseconds. Fine-grain services introduce additional challenges because server workload can fluctuate rapidly for those services. The results from previous simulation studies for load balancing may not be valid because fine-grain services are sensitive to various system overhead which is hard to accurately capture in simulations. In recognizing this limitation, we developed a prototype implementation based on a clustering infrastructure and conducted evaluations on a Linux cluster. Overall, our evaluation methodology is based on simulations as well as experiments with a prototype implementation.

The rest of this paper is organized as follows. Section 1.1 describes the service traces and synthetic workloads that are used in this paper. Section 2 presents our simulation studies on load balancing policies and the impact of various parameters. Section 3 describes a prototype system implementation in a Linux cluster with a proposed optimization. Section 4 evaluates the performance of this prototype system. Section 5 discusses related work and Section 6 concludes the paper.

1.1. Evaluation Workload

We collected the traces of two internal service cluster components from search engine *Teoma* [4] and their statistics are listed in Table 1. Both traces were collected across an one-week time span in late July 2001. One of the services provides the translation between query words and their internal representations. The other service supports a similar translation for Web page descriptions. Both services allow multiple translations in one access. The first trace has a mean service time of 22.2 ms and we call it the Fine-Grain trace. The second trace has a mean service time of 208.9 ms and we call it the Medium-Grain trace. We use a peak time portion (early afternoon hours of three consecutive weekdays) from each trace in our study. Most system resources are well under-utilized during non-peak times, therefore load balancing is less critical during those times. Note that the arrival intervals of those two traces may be scaled when necessary to generate workloads at various demand levels during our evaluation.

In addition to the traces, we also include a synthetic workload with Poisson process arrivals and exponentially distributed service times. We call this workload Poisson/Exp in the rest of this paper. Several previous studies on Internet connections and workstation clusters suggested that both the inter-arrival time distribution and the service time distribution exhibit high variance, thus are better modeled by Lognormal, Weibull, or Pareto distributions [10, 16]. We choose Poisson/Exp workload in our study for the following reasons. First, we believe the peak time arrival process is less bursty than the arrival process over a long period of time. Secondly, the service time distribution tends to have a low variance for services of the same type. In fact, Table 1 shows that those distributions in our traces have even lower variance than an exponentially distributed sample would have.

2. Simulation Studies

In this section, we present the results of our simulation studies. We confine our study on fully distributed load balancing policies that do not contain any single point of failure because high availability is essential in building large-scale network service infrastructure. We will first examine the load information inaccuracy caused by its dissemination delay. This delay is generally insignificant for coarse-grain jobs but it can be critical for fine-grain services. We will then move on to study two distributed load balancing policies: 1) the *broadcast* policy in which load information is propagated through server-initiated pushing; and 2) the *random polling* policy in which load information is propagated through client-initiated pulling. We choose them because they represent two broad categories of policies in terms of how load information is propagated from the servers to the clients. In addition, they are both shown to be competitive in a previous trace-driven simulation study [25].

In our simulation model, each server contains a non-preemptive processing unit and a FIFO service queue. The network latency of sending a service request and receiving a service response is set to be half a TCP roundtrip latency with connection setup and teardown, which is measured at

Workload	Number of accesses		Arrival interval		Service time	
	Total	Peak portion	Mean	Std-dev	Mean	Std-dev
Medium-Grain trace	1,055,359	126,518	341.5ms	321.1ms	208.9ms	62.9ms
Fine-Grain trace	1,171,838	98,933	436.7ms	349.4ms	22.2ms	10.0ms

Table 1. Statistics of evaluation traces.

516 us in a switched 100 Mb/s Linux cluster.

We choose the mean service response time as the *performance index* to measure and compare the effectiveness of various policies. We believe this is a better choice than system throughput for evaluating load balancing policies because system throughput is tightly related to the admission control, which is beyond the scope of this paper.

2.1. Accuracy of Load Information

Almost all load balancing policies use some sort of *load index*es to measure server load levels. Prior studies have suggested a linear combination of the resource queue lengths can be an excellent predictor of service response time [11, 24]. We use the total number of active service accesses, i.e. the queue length, on each server as the server load index. In most distributed policies, load indexes are typically propagated from server side to client side in some way and then each client uses acquired information to direct service accesses to lightly loaded servers. Accuracy of the load index is crucial for clients to make effective load balancing decision. However, the load index tends to be stale due to the delay between the moment it is being measured at the server and the moment it is being used at the client. We define the *load index inaccuracy* for a certain delay Δt as the statistical mean of the queue length difference measured at arbitrary time t and $t + \Delta t$. Figure 2 illustrates the impact of this delay (normalized to mean service time) on the load index inaccuracy for a single server through simulations on all three workloads. We also show the upperbound for Poisson/Exp in a straight line. With the assumption that the inaccuracy monotonically increases with the increase of Δt, the upperbound is the statistical mean of the queue length difference measured at any two arbitrary time t_1 and t_2. Let term ρ be defined as the mean service time divided by the mean arrival interval, which reflects the level of server load. For a Poisson/Exp workload, since the limiting probability that a single server system has a queue length of k is $(1-\rho)\rho^k$ [18], the upperbound can be calculated as:

$$\sum_{i,j=0}^{\infty} (1-\rho)^2 \rho^{i+j} |i-j| = \frac{2\rho}{1-\rho^2} \qquad (1)$$

The detailed calculation can be found in a technical report [22].

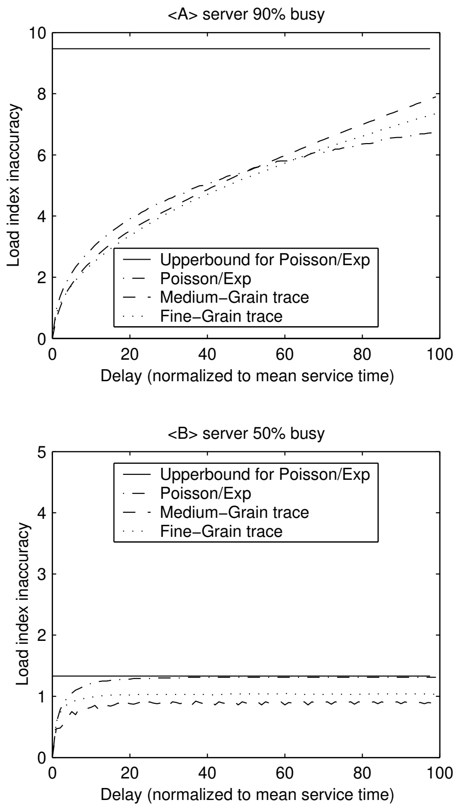

Figure 2. Impact of delay on load index inaccuracy with 1 server (simulation).

When the server is moderately busy (50%), the load index inaccuracy quickly reaches the upperbound (1.33 for Poisson/Exp) when delay increases, but the inaccuracy is moderate even under high delay. This means a random approach is likely to work well when servers are only moderately busy and fancier policies do not improve much. When the server is very busy (90%), the load index inaccuracy is much more significant and it can cause an error of around 3 in the load index when the delay is around 10 times the mean service time. This analysis reveals that when servers are busy, fine-grain services require small information dissemination delays in order to have accurate load information on the client side.

2.2. Broadcast Policy

In the broadcast policy, an agent is deployed at each server which collects the server load information and announces it through a broadcast channel at various intervals. It is important to have non-fixed broadcast intervals to avoid the system self-synchronization [12]. The intervals we use are evenly distributed between 0.5 and 1.5 times the mean value. Each client listens at this broadcast channel and maintains the server load information locally. Then every service request is made to a server with the lightest workload. Since the server load information maintained at the client side is acquired through periodical server broadcasts, this information becomes stale between consecutive broadcasts and the staleness is in large part determined by the broadcast frequency. Figure 3 illustrates the impact of broadcast frequency through simulations. A 50 ms mean service time is used for Poisson/Exp workload. Sixteen servers are used in the simulation. The mean response time shown in Figure 3 is normalized to the mean response time under an *ideal* approach, in which all server load indices can be accurately acquired on the client side free-of-cost whenever a service request is to be made.

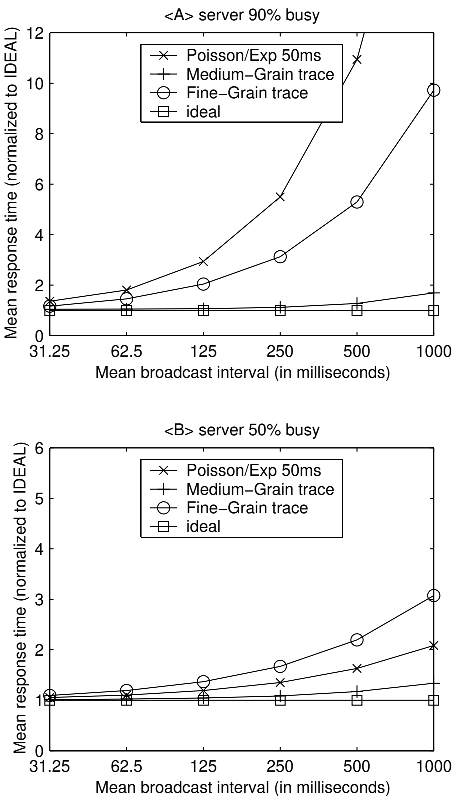

Figure 3. Impact of broadcast frequency with 16 servers (simulation).

When servers are 90% busy, we observe that the performance for broadcast policy with 1 second mean broadcast interval could be an order of magnitude slower than the ideal scenario for fine-grain services (Poisson/Exp and Fine-Grain trace). The degradation is less severe (up to 3 times) when servers are 50% busy, but it is still significant. This problem is mainly caused by the staleness of load information due to low broadcast frequency. But we also want to emphasize that the staleness is severely aggravated by the *flocking* effect of the broadcast policy, i.e. all service requests tend to flock to a single server (the one with the lowest perceived queue length) between consecutive broadcasts. The performance under low broadcast interval, e.g. interval 31.25 ms is close to the ideal scenario. However, we believe the overhead will be prohibitive under such high frequency, e.g. a sixteen server system with 31.25 ms mean broadcast interval will force each client to process a broadcast message every 2 ms.

2.3. Random Polling Policy

For every service access, the random polling policy requires a client to randomly poll several servers for load information and then direct the service access to the most lightly loaded server according to the polling results. An important parameter for a random polling policy is the poll size. Mitzenmacher demonstrated through analytical models that a poll size of two leads to an exponential improvement over pure random policy, but a poll size larger than two leads to much less substantial additional improvement [20]. Figure 4 illustrates our simulation results on the impact of poll size using all three workloads. Policies with the poll size of 2, 3, 4, and 8 are compared with the random and ideal approach in a sixteen server system. A 50 ms mean service time is used for Poisson/Exp workload.

This result basically confirms Mitzenmacher's analytical results in the sense that a poll size of two performs significantly better than a pure random policy while a larger poll size does not provide much additional benefit. Our simulation also suggests that this result is consistent across all service granularity and all server load level, which makes it a very robust policy. We believe the random polling policy is well-suited for fine-grain services because the just-in-time polling always guarantees very little staleness on the load information.

2.4. Summary of Simulation Studies

First, our simulation study shows that a long delay between the load index measurement time at the server and its time of usage at the client can yield significant inaccuracy. This load index inaccuracy tends to be more severe for finer-grain services and busier servers. Then we go on

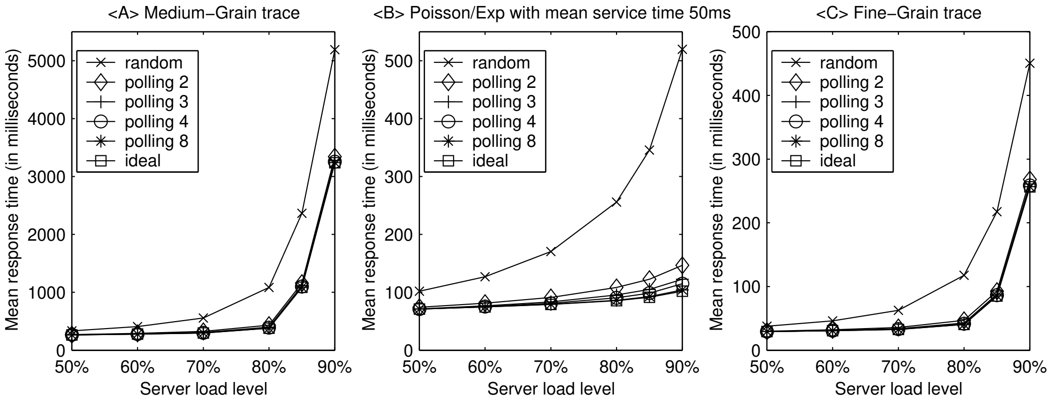

Figure 4. Impact of poll size with 16 servers (simulation).

to study two representative policies, broadcast and random polling. Our results show that random polling based load balancing policies deliver competitive performance across all service granularities and all server load levels. In particular, the policy with a poll size of two already delivers competitive performance with the ideal scenario. As for the broadcast policy, we identify the difficulty of choosing a proper broadcast frequency for fine-grain services. A low broadcast frequency results in severe load index inaccuracy, and in turn degrades the system performance significantly. A high broadcast frequency, on the other hand, introduces high broadcast overhead. Ideally, the broadcast frequency should linearly scale with the system load level to cope with rapid system state fluctuation. This creates a scalability problem because the number of messages under a broadcast policy would linearly scale with three factors: 1) the system load level; 2) the number of servers; and 3) the number of clients. In contrast, the number of messages under the random polling policy only scale with the server load level and the number of servers.

3. Prototype Design and Implementation

We have developed a prototype implementation of the random polling policy on top of a cluster-based service infrastructure. The simulation results in Section 2 favor random polling policy so strongly that we do not consider the broadcast policy in the prototype system.

3.1. System Architecture

This implementation is a continuation of our previous work on *Neptune*, a cluster-based infrastructure for aggregating and replicating partitionable network services [23]. Neptune allows services ranging from read-only to frequently updated be replicated and aggregated in a cluster environment. Neptune encapsulates an application-level network service through a service access interface which contains several RPC-like access methods. Each service access through one of these methods can be fulfilled exclusively on one data partition. We employ a flat architecture in constructing the service network infrastructure. A node can elect to provide services and it can also access services provided by other nodes. It serves as an internal *server* or *client* in each context respectively. Each node, when elects to provide services, maintains a service queue and a worker thread pool. The size of the thread pool is chosen to strike the best balance between concurrency and efficiency.

Conceptually, for each service access, the client first acquires the set of available server nodes through a *service availability subsystem*. Then it chooses one node from the available set through a *load balancing subsystem* before sending the service request. Our service availability subsystem is based on a well-known publish/subscribe channel, which can be implemented using IP multicast or a highly available well-known central directory. Each cluster node can elect to provide services through repeatedly publishing the service type, the data partitions it hosts, and the access interface. Published information is kept as soft state in the channel such that it has to be refreshed frequently to stay alive. Each client node subscribes to this well-known channel and maintains a service/partition mapping table.

We implemented a random polling policy for the load balancing subsystem. On the server side, we augmented each node to respond to *load inquiry* requests. For each service access, the client randomly chooses a certain number of servers out of the available set returned from the service availability subsystem. Then it sends out load inquiry requests to those servers through connected UDP sockets and asynchronously collects the responses using `select` system call.

Figure 5 illustrates the client/server architecture in our service infrastructure. Overall, both subsystems employ a loosely-connected and flat architecture which allows the

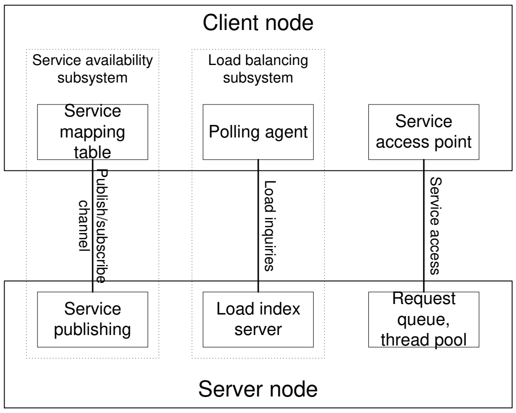

Figure 5. The client/server architecture in our service infrastructure.

service infrastructure to operate smoothly in the presence of transient failures and service evolution.

3.2. Discarding Slow-responding Polls

On top of the basic random polling implementation, we also made an enhancement by discarding slow-responding polls. Through a ping-pong test on two idle machines in our Linux cluster, we measured that a UDP roundtrip cost is around 290 us. However, it may take much longer than that for a busy server to respond a UDP request. We profiled a typical run under a poll size of 3, a server load index of 90%, and 16 server nodes. The profiling shows that 8.1% of the polls are not completed within 10 ms and 5.6% of them are not completed within 20 ms. With this in mind, we enhanced the basic polling policy by discarding polls not responded within 10 ms. Intuitively, this results in a tradeoff between consuming less polling time and acquiring more load information. However, we also realize that long polls result in inaccurate load information due to long delay. Discarding those long polls can avoid using stale load information, which is an additional advantage. And this tends to be more substantial for fine-grain services.

4. Experimental Evaluations

All the evaluations in this section were conducted on a rack-mounted Linux cluster with around 30 dual 400 Mhz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth. All the experiments presented in this section use 16 server nodes and up to 6 client nodes.

Due to various system overhead, we notice that the server load level cannot simply be the mean service time divided by the mean arrival interval. For each workload on a single-server setting, we consider the server reach full load (100%) when around 98% of client requests were successfully completed within two seconds. Then we use this as the basis to calculate the client request rate for various server load levels. The service processing on the server side is emulated using a CPU-spinning microbenchmark that consumes the same amount of CPU time as the intended service time. The ideal scenario in our simulation study is achieved when all server load indices can be accurately acquired on the client side free-of-cost whenever a service request is to be made. For the purpose of comparison, we emulate a corresponding ideal scenario in the evaluations of our prototype implementation. This is achieved through a centralized load index manager which keeps track of all server load indices. Each client contacts the load index manager whenever a service access is to be made. The load index manager returns the server with the shortest service queue and increments that queue length by one. Upon finishing one service access, each client is required to contact the load index manager again so that the corresponding server queue length can be properly decremented. This approach closely emulates the actual ideal scenario with a delay of around one TCP roundtrip without connection setup and teardown (around 339 us in our Linux cluster).

4.1. Evaluation on Poll Size

Figure 6 shows our experimental results on the impact of poll size using all three workloads. We observe that the results for Medium-Grain trace and Poisson/Exp workload largely confirm the simulation results in Section 2. However, for the Fine-Grain trace with very fine-grain service accesses, we notice that a poll size of 8 exhibits far worse performance than policies with smaller poll sizes and it is even slightly worse than the pure random policy. This is caused by excessive polling overhead coming from two sources: 1) longer polling delays resulted from larger poll size; 2) less accurate server load index due to longer polling delay. And those overheads are more severe for fine-grain services. Our conclusion is that a small poll size (e.g. 2 or 3) provides sufficient information for load balancing. And an excessively large poll size may even degrade the performance due to polling overhead, especially for fine-grain services.

4.2. Improvement of Discarding Slow-responding Polls

Table 2 shows the overall improvement and the improvement excluding polling time for discarding slow-responding

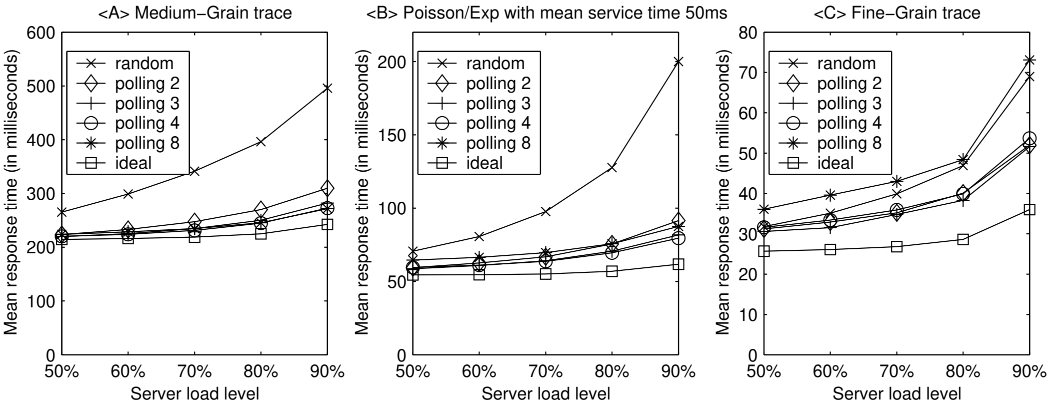

Figure 6. Impact of poll size based on a prototype implementation with 16 servers.

Workload	Mean response time (mean polling time)		Improvement (Excl. polling time)
	Original	Optimized	
Medium-Grain trace	282.1ms (2.6ms)	283.1ms (1.0ms)	-0.4% (-0.9%)
Poisson/Exp	81.8ms (2.7ms)	79.2ms (1.1ms)	3.2% (1.2%)
Fine-Grain trace	51.6ms (2.7ms)	47.3ms (1.1ms)	8.3% (5.2%)

Table 2. Performance improvement of discarding slow-responding polls with poll size 3 and server 90% busy.

polls. The experiments are conducted with a poll size of 3 and a server load index of 90%. The experiment on Medium-Grain trace shows a slight performance degradation due to the loss of load information. However, the results on both Fine-Grain trace and Poisson/Exp workload exhibit sizable improvement in addition to the reduction of polling time and this additional improvement is a result of avoiding the use of stale load information. Overall, the enhancement of discarding slow-responding polls can improve the load balancing performance by up to 8.3%. Note that the performance results shown in Figure 6 are not with discarding slow-responding polls.

5. Related Work

This work is a continuation of our previous research on Neptune: a cluster-based infrastructure for aggregating and replicating partitionable network services [23]. Closely related to a group of work on building large-scale network services in cluster environments [13, 15], Neptune provides a scalable, available, and extensible service infrastructure through service partitioning, replication, and aggregation. The load balancing study in this paper complements these service infrastructure work by providing efficient load balancing support suitable for all service granularities.

A large body of work has been done to optimize HTTP request distribution among a cluster of Web servers [1, 3, 6, 8, 17, 21]. Most load balancing policies proposed in such a context rely on the premise that all network packets go through a single front-end dispatcher or a TCP-aware (layer 4 or above) switch so that TCP level connection-based statistics can be accurately maintained. However, clients and servers inside the service cluster are often connected by high-throughput, low-latency Ethernet (layer 2) or IP (layer 3) switches, which do not provide any TCP level traffic statistics. Our study in this paper shows that an optimized random polling policy that does not require centralized statistics can deliver competitive performance based on a prototype implementation on a Linux cluster.

Previous research has proposed and evaluated various load balancing policies for cluster-based distributed systems [7, 9, 14, 19, 20, 24, 25]. Those studies mostly deal with coarse-grain distributed computation and often ignore fine-grain jobs by simply processing them locally. We put our focus on fine-grain network services by examining the sensitivity of the load information dissemination delay and its overhead. Both are minor issues for coarse-grain jobs but they are critical for fine-grain services.

A recent study shows that network servers based on Virtual Interface (VI) Architecture provide significant performance benefits over standard server networking interfaces [8]. Generally the advance in network performance improves the effectiveness of all load balancing policies. In particular, such an advance has certain impact on our results. First, a high-performance network layer may allow efficient and high frequency server broadcasts, which improves the feasibility of the broadcast policy. Secondly, a reduction in network overhead might change some quantitative results of our experimental evaluations. For instance, the overhead of the random polling policy with a large poll size might not be as severe as those shown in our experiments. Those issues should be addressed when advanced network standards become more widespread.

6. Concluding Remarks

In this paper, we study load balancing policies for cluster-based network services with the emphases on fine-grain services. Our evaluation is based on a synthetic workload and two traces we acquired from an online search engine. In addition to simulations, we also developed a prototype implementation on a Linux cluster and conducted experimental evaluations with it. Our study and evaluations identify techniques that are effective for fine-grain services and lead us to make several conclusions: 1) Random polling based load-balancing policies are well-suited for fine-grain network services; 2) A small poll size provides sufficient information for load balancing, while an excessively large poll size may even degrade the performance due to polling overhead; 3) An optimization of discarding slow-responding polls can further improve the performance by up to 8.3%.

Acknowledgment. This work was supported in part by NSF CCR-9702640, ACIR-0082666 and 0086061. We would like to thank Ricardo Bianchini, Apostolos Gerasoulis, Rich Martin, Hong Tang, and the anonymous referees for their valuable comments and help.

References

[1] ArrowPoint Communications. Web Switching White Papers. http://www.arrowpoint.com/solu-tions/white_papers/.

[2] BEA Systems. WebLogic and Tuxedo Transaction Application Server White Papers. http://www.bea.com/products/tuxedo/papers.html.

[3] Foundry Networks. White Paper: Cutting Through Layer 4 Hype. http://www.foundrynet .com/whitepaper_layer4.html.

[4] Teoma search. http://www.teoma.com.

[5] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proc. of the IEEE Intl. Symposium on Parallel Processing*, pages 850–856, Apr. 1996.

[6] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Services. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[7] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[8] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-based Network Servers. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122, Snowbird, UT, June 2001.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986.

[10] A. Feldmann. Characteristics of TCP Connection Arrivals. Technical report, AT&T Labs Research, 1998.

[11] D. Ferrari. A Study of Load Indices for Load Balancing Schemes. Technical Report CSD-85-262, EECS Department, UC Berkeley, Oct. 1985.

[12] S. Floyd and V. Jacobson. The Synchronization of Periodic Routing Messages. In *Proc. of ACM SIGCOMM'93*, pages 33–44, San Francisco, CA, Sept. 1993.

[13] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint Malo, Oct. 1997.

[14] K. K. Goswami, M. Devarakonda, and R. K. Iyer. Prediction-Based Dynamic Load-Sharing Heuristics. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):638–648, June 1993.

[15] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.* Monterey, CA, 1999.

[16] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[17] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proc. of the 7th Intl. World Wide Web Conf.*, Brisbane, Australia, Apr. 1998.

[18] L. Kleinrock. *Queueing Systems*, volume I: Theory. Wiley, New York, 1975.

[19] T. Kunz. The influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Trans. on Software Engineering*, 17(7):725–730, July 1991.

[20] M. Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 292–301, Newport, RI, June 1997.

[21] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 205–216, San Jose, CA, Oct. 1998.

[22] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-Grain Network Services. Technical Report TRCS2002-02, Dept. of Computer Science, UC Santa Barbara, 2002.

[23] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 197–208, San Francisco, CA, Mar. 2001.

[24] S. Zhou. An Experimental Assessment of Resource Queue Lengths as Load Indices. In *Proc. of the Winter USENIX Technical Conf.*, pages 73–82, Washington, DC, Jan. 1987.

[25] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, 14(9):1327–1341, Sept. 1988.