A Parameter-Free Load Balancing Mechanism For P2P Networks

Tyler Steele, Vivek Vishnumurthy and Paul Francis
Department of Computer Science, Cornell University, Ithaca, NY 14853
`{ths22,francis,vivi}@cs.cornell.edu`

Abstract

Random peer selection is commonly used to provide load-balancing in decentralized P2P systems. This paper addresses two practical concerns with using random peer selection for load balancing. The first has to do with *heterogeneous* peer selection mechanisms that accommodate differences in capacities of different peers. These heterogeneous selection mechanisms are not parameter-free: Each peer assumes knowledge of a particular *selection-parameter*, where correctly setting the parameter requires the knowledge of the global distribution of peer capacities. In this paper, we present a method that realizes parameter-free random peer selection by automatically computing the required parameter at each peer, thus making the selection primitive easier to employ by P2P applications.

A second common problem addressed by this paper is that of ensuring good load-balance in heavily loaded P2P systems, where new requests that result in more load still need to be accommodated. We give a method that estimates the overall utilization in the network, and adaptively determines the number of attempts needed to probabilistically accommodate each new request.

We implement both these enhancements over the Swaplinks peer selection algorithm, and show using experimental evaluations that they work as intended.

1 Introduction

Load balancing is an important and pervasive problem in all distributed systems. Load balancing in decentralized P2P systems has usually been performed using a heterogeneous random peer selection primitive: The peer used to satisfy a request is randomly chosen from the entire P2P system, with high-capacity peers preferred over lower-capacity peers. For example, among unstructured P2P applications, the Chunkyspread overlay multicast system [11] employs the Swaplinks unstructured random selection algorithm [12], and the Gia file-sharing system performs file-search over a graph where high-capacity peers are easier to reach. The CFS structured P2P storage system [2] uses heterogeneous selection to pick peers to store files. In all of these examples, the overhead involved in finding a single random peer is negligible when compared to the load that subsequently results from the application at the chosen peer.

To accommodate heterogeneity, many random selection algorithms take as input a *selection-parameter* from the application at each peer, and (ideally) select each peer with a probability directly proportional to the specified parameter. Unfortunately, the selection parameter impacts the peer's overhead as well. For example, in Swaplinks, each peer's degree in the overlay graph is equal (on average) to twice its selection parameter. In CFS [2], each peer spawns "virtual servers" in direct proportion to the specified parameter. Thus, the higher the specified parameter, the higher the overhead imposed on the peer.

In order to be efficient as well as accurate, the selection parameters should therefore be as low as possible, while still being directly proportional to the peers' capacities. Correctly realizing this, however, requires knowledge of the global distribution of capacities – we term this the *global capacity view*. It would be unrealistic to assume that the application knows the global capacity view at each peer. In this paper, we describe a simple and novel method that provides each peer with an approximate global capacity view, thereby enabling the peer to automatically set the correct selection parameter. In the case of Swaplinks, this results in the first parameter-free random selection system.

A second, related, issue that this paper addresses is that of load balance and request blocking on a heavily loaded system. Consider an example network where more than 50% and less than 75% of the peers are overloaded. [1] The first peer selected to handle a request here is unlikely to have the spare capacity required to handle the request, leading to the rejection of the request. If requests are allowed to be retried however, a node with enough spare capacity is likely to be found here with a limited number of retries. On the other hand, if the system load is excessively heavy (say only one peer in a million not overloaded), it would be intractable to find underloaded peers to handle requests.

In this paper, we give a method that estimates the overall system load, and adaptively tunes the number of request-retries needed to probabilistically find a peer capable of handling each request. In addition, we can leverage the system load estimate to provide (i) Rate-control, where requests are proactively blocked if the overall load is excessively high, and (ii) Different qualities of service (QoS), should the application need these features.

The above ability to work with overloaded peers lets us overcome inaccuracies in the computation of the selection parameter as well. If a few peers have non-optimal selection parameters and thus are overloaded, it is not overly detrimental to the health of the system: Such peers simply reject requests when they are overloaded, resulting in the rejected requests being retried.

The two issues we address in this paper were those we actually faced when using the Swaplinks algorithm to extend the STUNT NAT traversal infrastructure [5]. Our goal here was to efficiently distribute load across a set of nodes where node-capacities vary widely, and where we want to allow for resource scarcity across the entire set. The particular mechanisms we developed to address these issues can be used with other random selection primitives as well.

Overall, this paper makes the following contributions:

1. Design and evaluation of a novel general parameterless approach to heterogeneous P2P load balance, resulting in a specific system (Swaplinks) that is totally parameter-free.
2. Design and evaluation of an algorithm that can automatically fine-tune the trade-off between request retries and request blocking.

2 Related Work

Swaplinks [12] and Gia [1] are unstructured P2P algorithms that achieve load-balancing using random walks. Both of these algorithms build graphs, assign larger node degrees to nodes with higher capacities, and have a notion of the minimum degree a node could have. Optimally setting node-degrees here requires the global capacity view.

The Y_0 algorithm [4] and the load-balancing algorithm in CFS [2] are DHT-based and use virtual servers. The number of virtual servers at each peer is ideally directly proportional to its capacity; correctly setting this number again needs the global capacity view.

In contrast to the above schemes, the selection parameter in the KRB algorithm [13] can be set with local knowledge alone, i.e., it does not require peers to have the global capacity view. However, KRB has other parameters to which it is sensitive, so is still not parameter-free. [2]

There are other DHT-based load-balancing algorithms that do not depend on global capacity views [9, 3, 6]. These however require that load be transferred between different peers in the presence of network churn. This approach only works where jobs can be preempted and shipped to different nodes before completion, limiting its applicability.

We believe that this paper is the first to realize an acceptable response rate in heavily loaded P2P systems without resorting to preemption of active jobs. The schemes mentioned in the previous paragraph [9, 3, 6] can also realize acceptable response rates, but assume preemption of active jobs.

The methods we use to estimate global state are similar in aim to aggregate computation methods proposed in, for example, [7] (see [8] for a survey). The difference is that our requirements for accuracy are

[1]We use “overloaded” here to denote a peer that is using almost all of its allotted capacity, and so cannot support further requests. We use “underloaded” to refer to a peer that is not overloaded.

[2]KRB is sensitive to the rate at which probes are sent between different peers and the various parameters of the Bamboo DHT [10] that it is based on.

weaker: For example, even with some inaccuracies in the selection-parameter computation, overall load-balancing is still largely maintained through our use of request-retries. This fact allows us to use simpler and cheaper methods in our estimation.

3 Parameter-free Random Selection

In Section 1, we noted that optimally setting the selection parameter at each peer requires that each peer's selection parameter be as low as possible while also being directly proportional to the peer's capacity. There is generally a lower-bound on the selection parameter as well: E.g., in Swaplinks, each peer's degree should be set to at least a minimum value to protect against partitions. Optimally assigning the selection parameter requires knowledge of the global capacity distribution. For example, if bandwidth is the bottleneck resource, a completely local solution where each peer sets its selection parameter to the value of its bandwidth in bits/second results in selection parameter values that are not the lowest possible.[3]

We use repeated sampling to automatically provide peers with an approximate global-capacity view, allowing each peer to independently set the selection parameter, thus freeing the application from this responsibility. By "sampling", we simply mean a method that informs peers about capacities of other peers in the system in a random fashion. We assume that the application at each peer is capable of gauging the *absolute* capacity of the peer: the absolute capacity is the total bottleneck-resource (e.g., bandwidth in a file-sharing application) that the peer is willing to commit. We realize the sampling method by having each peer selected to service a request note the capacity of the peer that initiated the request. Assuming that all peers are sending requests frequently enough and at about the same rates, this results in a distribution where each peer has an approximately uniform global capacity-view. When the request-sending rate is too low, we might need to launch dummy "sample-requests" in order to generate enough samples, but since a low request rate also means a low overall system load, we believe this will not be necessary. When the request rate is naturally large however, the specified method has the advantage that it incurs no extra overhead, since each instance of sampling is essentially piggybacked on the existing request servicing method.

[3]On the other hand, if we reduce the value of the selection parameter by setting it to the value of the bandwidth in megabits/second, the parameter could now be smaller than the allowed minimum (and coarser-grained than required as well).

Each peer compiles a fixed-size window of the most recently seen samples, and uses this to compute its selection parameter. Let us denote by c_{min} the minimum capacity of any peer in the window, and by p_{min} the minimum value allowed for the selection parameter. Since the selection parameter should be as low as possible, a peer that has capacity c_{min} should have a selection parameter equal to p_{min}. The next requirement of direct proportionality dictates that the selection parameter p of a peer with capacity c be set as follows:

$$p = c \times \frac{p_{min}}{c_{min}} \tag{1}$$

The use of the window of recent samples lets the method naturally adapt to churn. We however modify the actual selection parameter only when it differs from the value as given by the above equation by more than 10%. This avoids repeated unnecessary changes in the selection parameter. An additional method (not currently implemented) we could use to avoid wide fluctuations caused by outliers is to set c_{min} to the 5^{th} percentile value of the capacities in the window, instead of the absolute minimum. This makes it more likely that all peers settle on about the same c_{min}.

We note here that our selection-parameter computation might occasionally lead to non-optimal values, owing to its distributed nature, and its approximation of the global state from limited information. However, our use of request-retries (Section 4) lets us naturally overcome these imperfections: Overloaded peers do not accept further requests, leading to such requests eventually finding underloaded peers.

We implement the above scheme on the Swaplinks heterogeneous selection algorithm [12] resulting in the first completely parameter-free heterogeneous selection primitive. In the original Swaplinks scheme, the only parameter a node needs is its target degree in the overlay graph, which we set equal to twice its selection parameter. The selection parameter needs to be an integer here, so we round the value computed in Equation 1 to get the selection parameter.

4 Adaptivity Under Heavy Load

In this section, we focus on the problem of ensuring acceptable request-response rates in heavily loaded P2P systems. We assume the paradigm where the application at each peer uses no more than the amount of resource allotted to it. This means that if a peer is already close to its allotted maximum load, and if the peer is selected to service a subsequent request, the request is dropped. We term this *admission-control*.

To ensure that requests are responded to satisfactorily, we have the following goal: Each request should be successfully handled with probability at least equal to a given *assurance* value. We denote this the assurance requirement. The assurance value typically would be a high constant (say 90%) across the system.

In a heavily loaded system, admission control results in a substantial portion of the requests getting dropped. But unless the system is *fully* loaded, there would still be (many) underloaded peers in the system that are capable of handling new requests. In order to satisfy the assurance requirement, we allow dropped requests to be retried multiple times, thereby increasing the likelihood that an underloaded peer is found. We determine the number of retries needed as follows: If f is the probability that a selected peer turns out to be too overloaded to handle a request, and n the number of attempts needed in all, $(1-f^n)$ is the probability that at least one attempt was successful in finding an underloaded peer, resulting in the following equation:

$$n = \lceil log_f(1 - assurance) \rceil \tag{2}$$

We term f, the probability that a selected peer is overloaded, the *failure probability*. If all peers know this value, they can use the above equation to find the number of retries needed. We again use a sampling method to estimate f. Whenever a peer *A* initiates a request that results in the selection of peer *B*, *A* conveys to *B* a 1-bit indicator (1 or 0) of whether it is currently overloaded or not, and its selection parameter. Each peer then estimates f as follows:

$$f = \frac{\Sigma_i \, (over_i \cdot p_i)}{\Sigma_i p_i} \tag{3}$$

where $over_i$ is the 1-bit overload indicator, and p_i the selection parameter conveyed in the i^{th} sample. The weighting by p_i in the above equation accounts for the fact that peers are selected in proportion to their selection parameter. We use a fixed-size window of the most recently seen samples in this computation. We note that a request is rejected if all the n attempts fail to find an underloaded peer.

In our implementation of this solution over Swaplinks, we use *extensions* rather than outright retries. Swaplinks uses fixed-length random walks for selection; an extension is where the walk is extended by one hop when the node found by the walk turns out to be overloaded. Use of extensions reduces the imposed load by an order of magnitude, while its selection quality is comparable to using new full-length walks [12].

An alternative to the above solution is to modify each peer's selection parameter as a function of how loaded it is: If it is close to being overloaded, it would reduce its selection parameter to prevent new requests from finding it, and vice-versa. This solution however is inherently unstable. For example, if most of the peers in the system are close to being overloaded, it results in an avalanche of selection parameter reductions, since the reductions do not really help in reducing the *overall* load on the system.

Equation 2 suggests that the number of attempts needed increases rapidly as f gets very close to 1. Our estimate of f can be used here to proactively block requests when the f-estimate is greater than a given threshold. This represents a form of rate-control, and prevents flooding the system with ultimately wasteful messages.

Another abstraction that this setup can naturally support is that of different classes of quality of service. Different requests here could have differing "importances", i.e., different desired assurances. Using Equation 2, we can compute the number of attempts required in the different classes.

5 Results

The results we present in this section are of a C implementation of the Swaplinks algorithm [12], extended with the enhancements presented in this paper. This implementation is part of an extension of the STUNT NAT traversal infrastructure [5]: random selection is used here to find publicly accessible *relay* nodes that help nodes behind NATs communicate with each

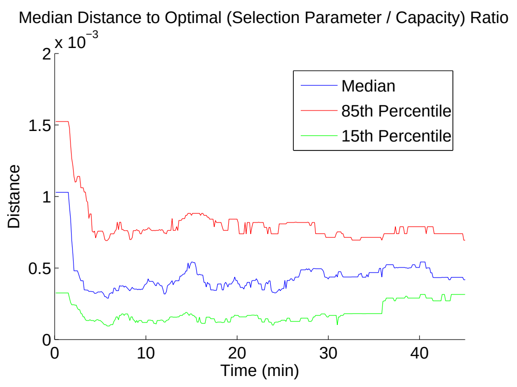

Figure 1: Comparing actual selection parameters to optimal values

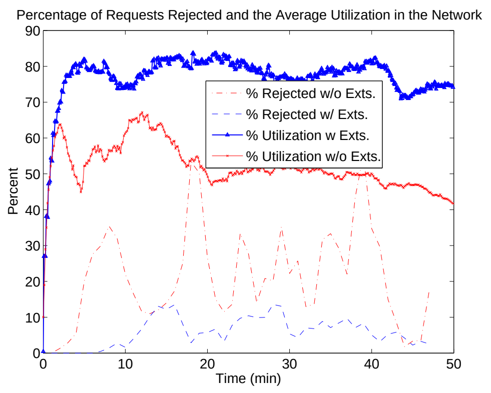

Figure 2: Improvements due to extensions.

other.

We use a network size of about 100 in our experiments: the relatively small size is due to the fact that we use a real implementation, and that we conducted our experiments on a single (Linux) host. Unless otherwise mentioned, peer-capacities in our experiments range uniformly from 1200 to 8000 units, except for a single *supernode* with a capacity of 50,000 units. Each experiment lasts about 45 minutes. Every several seconds (between five and ten) each node requests between 200 and 600 units of resources from a randomly selected peer. If accepted, each request lasts between two and five minutes. We use an assurance value (Section 4) of 0.8. The selection parameter in case of a Swaplinks peer is half its target degree in the Swaplinks overlay graph, with a minimum allowed value p_{min} of 3. We use a window-size of 30 in the sampling methods used to compute the selection parameter and the failure probability.

We do not include node churn in our experiments because we assume that robustness to churn is guaranteed by the selection primitive. We do include a form of “request churn” however, because the above pattern of random incoming requests change the spare capacity of the system with time.

We first evaluate how well the automatic selection parameter computation works. Ideally, the selection parameters are directly proportional to peer-capacities. Each peer’s ratio of its selection parameter to its capacity should therefore be identical, and equal to $\frac{p_{min}}{c_{min}}$ (see Equation 1), where c_{min} is the minimum capacity across the system (1200). We see that this optimal ratio is $\frac{3}{1200} = 2.5 \times 10^{-3}$.

Figure 1 shows the distance between the above optimal ratio and the actual ratios of selection-parameters to capacities, aggregated over all peers in the system. This result shows that for a majority of the peers (more than 70%), the selection parameter is within about 30% of the ideal value, indicating that the automatic selection parameter computation gives mostly acceptable results. Note again that inaccuracies in the selection parameter can be overcome through the use of request-retries.

We next look at how well our mechanisms cope with heavy loads. Figure 2 shows the improvement realized by the controlled use of extensions (Section 4). It shows the proportion of requests rejected and the overall utilization both with and without the use of extensions. Utilization here is the ratio of the total active load in the system to the total node-capacity in the system. The plot shows that the use of extensions improves both the likelihood that requests are successfully handled, and the overall system utilization: Without extensions, there are times when more than 50% of requests are denied, while with extensions the proportion of rejected requests rarely exceeds 10%.

We show the overhead incurred by the use of extensions in Figure 3. In this plot, we track the number of extensions granted to requests, aggregated over all peers. The actual overhead incurred actually is likely to be smaller than this figure, since the number of extensions used is possibly smaller than the number granted. The number of extensions granted is computed using Equation 2. The result indicates that the overhead is acceptable: No more than two or three extensions are granted on average, and the worst case number of extensions is limited by ten.

We next present a simple experiment that demon-

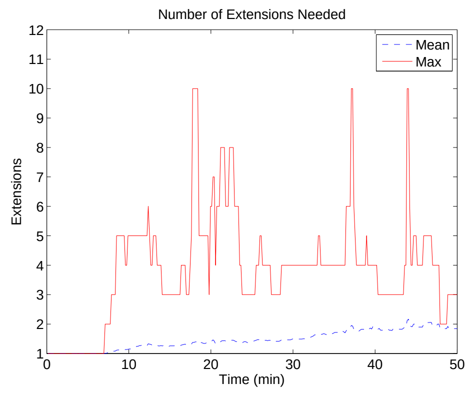

Figure 3: Cost of request retries

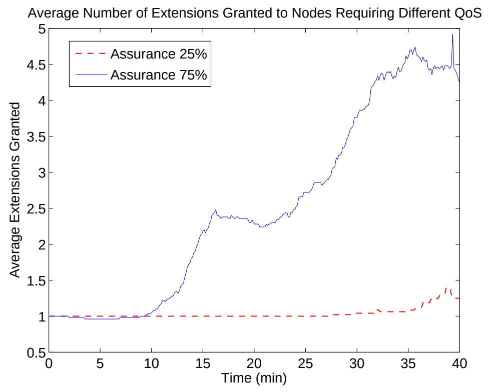

Figure 4: Different qualities of service

strates our ability to support different qualities of service (QoS). We have two classes of nodes, one requiring a 75% assurance that requests are granted, and the other 25%. All nodes are otherwise identical: they all have the same capacity. Figure 4 shows the average number of extensions granted to nodes in each service class. As expected, peers in the 75% class use more extensions than peers in the other class. At the start of the experiment though, both classes show similar behavior. This is because nodes do not act on their failure probability estimates until they have a window of at least 30 samples. Once nodes start accumulating the required number samples, the two curves start to diverge. This result shows that requests from classes with higher required assurances use more extensions, which results in them being accordingly more likely to succeed.

6 Conclusions

In this paper, we identified two real problems that come up when employing heterogeneous random selection primitives for the purpose of load-balancing in P2P applications: They are, namely, collaboratively determining the selection parameter, and the problem of maintaining good request response rates under heavy loads. We gave methods to solve both problems, implemented them over the Swaplinks heterogeneous selection algorithm, and showed that the methods work as intended through a small-scale experimental evaluation.

In terms of future work, we need to conduct a larger-scale evaluation of the mechanisms proposed in this paper under realistic churn rates. We should also test the mechanisms on other peer selection primitives, like the one used by CFS [2].

References

[1] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proc. ACM SIGCOMM*, 2003.

[2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, 2001.

[3] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proc. IEEE Infocom*, 2004.

[4] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proc. IEEE Infocom*, 2005.

[5] S. Guha and P. Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *Proc. IMC*, 2005.

[6] D. R. Karger and M. Ruhl. New algorithms for load balancing in peer-to-peer systems. In *IRIS Student Workshop*, 2003.

[7] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. FOCS*, 2003.

[8] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. In *Elsevier Journal of Systems and Software*, 2007.

[9] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *Proc. IEEE Infocom*, 2005.

[10] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. Usenix Annual Technical Conference*, 2004.

[11] V. Venkataraman and P. Francis. Chunkyspread: Heterogeneous unstructured tree-based peer to peer multicast. In *Proc. ICNP*, 2006.

[12] V. Vishnumurthy and P. Francis. On heterogeneous overlay construction and random node selection in unstructured P2P networks. In *Proc. IEEE Infocom*, 2006.

[13] V. Vishnumurthy and P. Francis. A comparison of structured and unstructured P2P approaches to heterogeneous random peer selection. In *Proc. Usenix Annual Technical Conference*, 2007.