Load Balancing in Structured P2P Systems[⋆]

Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica

Computer Science Division
University of California at Berkeley
Berkeley, CA 94720, USA
{ananthar,karthik,sonesh,karp,istoica}@cs.berkeley.edu

Abstract. Most P2P systems that provide a DHT abstraction distribute objects among "peer nodes" by choosing random identifiers for the objects. This could result in an O(log N) imbalance. Besides, P2P systems can be highly heterogeneous, i.e. they may consist of peers that range from old desktops behind modem lines to powerful servers connected to the Internet through high-bandwidth lines. In this paper, we address the problem of load balancing in such P2P systems. We explore the space of designing load-balancing algorithms that uses the notion of "virtual servers". We present three schemes that differ primarily in the amount of information used to decide how to re-arrange load. Our simulation results show that even the simplest scheme is able to balance the load within 80% of the optimal value, while the most complex scheme is able to balance the load within 95% of the optimal value.

1 Introduction

In this work, we address the problem of load balancing in peer-to-peer (P2P) systems that provide a distributed hash table (DHT) abstraction ([1, 2, 4, 5]). In such structured systems, each data item that is stored is mapped to a unique identifier ID. The identifier space is partitioned among the nodes and each node is responsible for storing all the items that are mapped to an identifier in its portion of the space. Thus, the system provides an interface comprising two functions: *put(ID, item)*, which stores the item associating an identifier ID with it, and *get(ID)* which retrieves the item corresponding to the identifier ID.

While peer-to-peer algorithms are symmetric, that is, all peers play the same role in the protocol, P2P systems can be highly heterogeneous. A P2P system like Gnutella or Kazaa may consist of peers that range from old desktops behind modem lines to powerful servers connected to the Internet through high-bandwidth lines.

If node identifiers are chosen at random (as in [1, 2, 4, 5]), a random choice of item IDs results in an $O(\log N)$ imbalance factor in the number of items stored at a node. Furthermore, applications may associate semantics with IDs, which

[⋆] This research was supported by the NSF under Cooperative Agreement No. ANI-0225660 (http://project-iris.net), and Career Award ANI-0133811.

F. Kaashoek and I. Stoica (Eds.): IPTPS 2003, LNCS 2735, pp. 68–79, 2003.

means that IDs are no longer uniformly distributed. For example, in a database application, each item can be a tuple whose ID represents the value of its primary key [6].

A popular technique to deal with hot-spots is *caching*. However, caching will not work for certain types of resources such as storage. Furthermore, if the load is caused by the popularity of a large number of small items (as can be expected in database applications), then caching has to push out a significant fraction of the items before it is effective. On the other hand, the techniques we propose are not very effective in dealing with hot-spots. Therefore, we believe that caching is both orthogonal and complementary to the load-balancing techniques we describe in this paper.

This paper presents three simple load-balancing schemes that differ primarily in the amount of information used to decide how to rearrange load. Our simulation results show that even the simplest scheme is able to balance the load within 80% of the optimal value, while the most complex scheme is able to balance the load within 95% of the optimal value.

2 Preliminaries

In this work, we use the concept of *virtual servers* [3] for load balancing. A virtual server looks like a single peer to the underlying DHT, but each physical node can be responsible for more than one virtual server. For example, in Chord, each virtual server is responsible for a contiguous region of the identifier space but a node can own non-contiguous portions of the ring by having multiple virtual servers. The key advantage of splitting load into virtual servers is that we can move a virtual server from any node to any other node in the system. This operation looks like a *leave* followed by a *join* to the underlying DHT, and hence is supported by all DHTs. In contrast, if each node has only one virtual server, it can only transfer load to nodes that are its neighbors in the ID space (for example, its successor and predecessor in Chord). Even though splitting load into virtual servers will increase the path length on the overlay, we believe that the flexibility to move load from any node to any other node is crucial to any load-balancing scheme over DHTs.

Even though a large number of applications have been suggested in the literature for DHT-based P2P systems, little can be predicted about which applications will eventually turn out to be popular or about the typical workloads that might be experienced. Since it is very hard to address the load balancing problem in its full generality, we make some simplifying assumptions, which we believe are reasonable in practice. First, while we do not restrict ourselves to a particular type of resource (storage, bandwidth or CPU), we assume that there is only one bottleneck resource we are trying to optimize for. Second, we consider only schemes that achieve load balancing by moving virtual servers from heavily loaded nodes to lightly loaded nodes. Such schemes are appropriate for balancing storage in distributed file systems, bandwidth in systems with a web-server like load, and processing time when serving dynamic HTML content or performing distributed join operations [6]. Third, we assume that the load on a virtual server is stable (or can otherwise be predicted, as in a distributed join operation) over the timescale it takes for the load balancing algorithm to operate.

3 Load-Balancing Schemes

In this section, we present three simple load-balancing schemes. All these schemes try to balance the load by transferring virtual servers from heavily loaded nodes to lightly loaded nodes. The key difference between these three schemes is the amount of information required to make the transfer decision. In the simplest scheme, the transfer decision involves only two nodes, while in the most complex scheme, the transfer decision involves a set consisting of both heavy and light nodes. Before delving into the details of the schemes, we first define the notion of light and heavy nodes more precisely.

3.1 Heavy and Light Nodes

Let L_i denote the load of node i, where L_i represents the sum of the loads of all virtual servers of node i. We assume that every node also has a *target load* (T_i) chosen beforehand. A node is considered to be *heavy* if $L_i > T_i$, and is *light* otherwise. The goal of all our load balancing algorithms is to decrease the total number of heavy nodes in the system by moving load from heavy nodes to light nodes.

While this binary modeling of the state of a node may seem very restrictive at first glance, we believe that it is both simple and sufficient for a number of applications. For systems with a well-defined cliff in the load-response curve, the load at which the cliff occurs is a natural choice for the target load. On the other hand, if the goal is to equalize the load on all the nodes in the system, we can choose the target close to average load in the system (a rough estimate by random sampling might be good enough). Assume that C_i denotes the capacity of a node[1], and that the goal is to divide the load in proportion to the capacity. Ideally, we want the load on node i to be $(\bar{L}/\bar{C})C_i$, where N is the total number of nodes in the system, the average load $\bar{L} = (\sum_{i=1}^{N} L_i)/N$, and the average capacity $\bar{C} = (\sum_{i=1}^{N} C_i)/N$. However, since in practice this target may be hard to achieve, we approximate it with $T_i = (\bar{L}/\bar{C} + \delta)C_i$, where δ is a slack variable and represents a trade-off between the amount of load moved and the quality of balance achieved.

3.2 Virtual Server Transfer

The fundamental operation performed for balancing the loads is transferring a virtual server from a heavy node to a light node. Given a heavy node h and a light node l, we define the *best virtual server* to be transferred from h to l as the virtual server v the transfer of which satisfies the following constraints:

[1] For example, the up-link bandwidth in the case of a web server.

1. Transferring v from h to l will not make l heavy.
2. v is the lightest virtual server that makes h light.
3. If there is no virtual server whose transfer can make h light, transfer the heaviest virtual server v from h to l.

Intuitively, the above scheme tries to transfer the minimum amount of load to make h light while still maintaining l light. If this is not possible, the scheme will transfer the largest virtual server that will not change l's status. The idea is to increase the chance of h finding another light node that eventually will allow h to shed all its excess load.

Note that this scheme guarantees that a transfer can only decrease the number of heavy nodes. In addition, we do not consider transfers between nodes of the same type (i.e., when both nodes are either heavy or light). This way, we guarantee that when the load in the system is high ($\bar{L} > \bar{T}$), no thrashing will occur. Also, we *can stop at any time* if the desired performance is reached.

3.3 Splitting of Virtual Servers

If no virtual server in a heavy node can be transferred in its entirety to another node, then a possibility is to split it into smaller virtual servers and transfer a smaller virtual server to a light node. While this would improve the time taken to achieve balance and possibly reduce the total load transferred, there is a risk of excessively fragmenting the identifier space. An increase in the number of virtual servers would imply an increase in the overlay hop length and size of routing tables. Hence, a scheme to periodically merge virtual servers would be needed to counteract the increase in the number of virtual servers caused by splitting.

Since this would complicate our algorithms considerably, we consider only load-balancing schemes that do not need to split virtual servers. Instead, we assume that the load of all virtual servers is bounded by a predefined threshold. Each node is responsible for enforcing this threshold by splitting the virtual servers when needed. In our simulations, we set the threshold for splitting to $\bar{T}$. This choice has the property that if the target is achievable ($\bar{L} < \bar{T}$), no more than N virtual servers need to be split. Recall that N is the number of nodes in the system.

3.4 One-to-One Scheme

The first scheme is based on a one-to-one rendezvous mechanism, where two nodes are picked at random. A virtual server transfer is initiated if one of the nodes is heavy and the other is light.

This scheme is easy to implement in a distributed fashion. Each light node can periodically pick a random ID and then perform a lookup operation to find the node that is responsible for that ID. If that node is a heavy node, then a transfer may take place between the two nodes.

In this scheme only light nodes perform probing; heavy nodes do not perform any probing. There are three advantages of this design choice. First, heavy nodes are relieved of the burden of doing the probing as well. Second, when the system load is very high and most of the nodes are heavy, there is no danger of either overloading the network or thrashing. Third, if the load of a node is correlated with the length of the ID space owned by that node, a random probe performed by a light node is more likely to find a heavy node.

3.5 One-to-Many Scheme

Unlike the first scheme, this scheme allows a heavy node to consider more than one light node at a time. Let h denote the heavy node and let $l_1, l_2, \ldots, l_k$ be the set of light nodes considered by h. For each pair (h, l_i) we pick a virtual server v_i using the same procedure described in Section 3.2. Among the virtual servers that this procedure gives, we choose the lightest one that makes heavy node h light. If there is no such a virtual server, we pick the heaviest virtual server among the virtual server v_i $(1 \leq i \leq k)$ to transfer.

We implement this scheme by maintaining *directories* that store load information about a set of light nodes in the system. We use the same DHT system to store these directories. Assume that there are d directories in the system, where d is significantly smaller than the number of physical nodes N. A light node l is hashed into a directory by using a well-known hash function h' that takes values in the interval $[0, d)$. A directory i is stored at the node which is responsible for the identifier $h(i)$, where h is another well-known hash function.

A light node l will periodically advertise its target load and current load to node $i = h(h'(l))$, which is responsible for directory $h'(l)$. In turn, the heavy nodes will periodically sample the existing directories. A heavy node n picks a random number $k \in [0, d]$ and sends the information about its target load and the loads of all its virtual servers to node $j = h(h'(k))$. Upon receiving such a message, node j looks at the light nodes in its directory (i.e., directory $h'(k)$) to find the best virtual server that can be transferred from n to a light node in its directory. This process repeats until all the heavy nodes become light.

3.6 Many-to-Many Scheme

This scheme is a logical extension of the first two schemes. While in the first scheme we match one heavy node to a light node and in the second scheme we match one heavy node to many light nodes, in this scheme we match many heavy nodes to many light nodes.

We first start with the description of a centralized scheme that has full information about all nodes in the system. Our goal is to bring the loads on each node to a value less than the corresponding target. To allow many heavy nodes and many light nodes to interact together, we use the concept of a *global pool* of virtual servers, an intermediate step in moving a virtual server from a heavy node to a light node. The *pool* is only a local data structure used to compute the final allocation; no load is actually moved until the algorithm terminates.

The scheme consists of three phases:

- **Unload:** In this phase, each heavy node i transfers its virtual servers greedily into a global *pool* till it becomes light. At the end of this phase, all the nodes are light, but the virtual servers that are in the pool must be transferred to nodes that can accommodate them.
- **Insert:** This phase aims to transfer all virtual servers from the pool to light nodes without creating any new heavy nodes. This phase is executed in stages. In each stage, we choose the heaviest virtual server v from the pool, and then transfer it to the light node k determined using a best-fit heuristic, i.e., we pick the node that minimizes $T_k - L_k$ subject to the condition that $(T_k - L_k) \geq load(v)$. This phase continues until the pool becomes empty, or until no more virtual servers can be transferred. In the former case, the algorithm terminates, as all the nodes are light and there are no virtual servers in the pool. In the latter case, the algorithm continues with the dislodge phase.
- **Dislodge:** This phase swaps the largest virtual server v from the pool with another virtual server v' of a light node i such that $L_i + load(v) - load(v') \leq T_i$. Among all light nodes, we pick the one from which we can remove the lightest virtual server. If we cannot identify a light node i such that $load(v') < load(v)$, the algorithm terminates. Otherwise the algorithm returns to the insert phase. Since we are considering nodes from the pool in descending order of their load, insert might work for the next node in the pool.

To implement this scheme in a distributed fashion we can use similar techniques as in the One-to-Many scheme. The main difference in this case is that we hash heavy nodes into directories as well. In particular, each node i chooses a random number k between 1 and d (where d is the number of directories) and then sends its complete load information to node $j = h(h'(k))$. After it receives information from enough nodes, node j performs the algorithm presented above and then sends the solution back to these nodes. The solution specifies to each node the virtual servers it has to transfer. The algorithm continues with nodes rehashing periodically to other directories. In the distributed version, at any stage if an insert fails, we have the choice of either going into the dislodge phase or just moving the virtual server v being considered back to the node from which it came into the pool. After re-hashing, we may be able to find a better light node to move v to in the next round. Thus, we avoid the overhead of moving load out of a light node at the cost of having to go through more rounds.

4 Simulations

We simulated all three algorithms under a variety of conditions to understand their performance and their limitations. The goal of our simulations is to understand the fundamental trade-offs in the different approaches; we do not claim that we have a bullet-proof algorithm that can be used efficiently in all DHT based systems. Given the lack of information about applications and their workloads, we only make conservative assumptions that stress our algorithms. We consider two types of distributions to generate the load on a virtual server.

- **Gaussian Distribution** Let f be the fraction of the identifier space owned by a given virtual server. This fraction is assumed to be exponentially distributed (this is true in both Chord and CAN). The load on the virtual server is then chosen from a Gaussian distribution with mean μf and standard deviation $\sigma\sqrt{f}$. Here, μ and σ are the mean and the standard deviation of the total load on the system (we set $\sigma/\mu = 10^{-3}$). This distribution would result if the total load on a virtual server is due to a large number of small items it stores, and the individual loads on the items are independent.
- **Pareto Distribution** The load on a virtual server is chosen from a power-law distribution with exponent 3 and mean μf. The standard deviation for this distribution is ∞. The heavy-tailed nature of this distribution makes it a particularly bad case for load balancing.

We consider two key metrics in our simulations: the **total load** moved between the nodes to achieve a state where all nodes are light, and the **number of probes** (or the *number of rounds*). The second metric gives us an idea of the total time taken to converge and the control traffic overhead caused by the load balancing algorithm.

Scalability: First, we look at the scalability of the different algorithms. In the one-to-many and the many-to-many schemes, the fraction of load moved and the number of probes per node depend only on N/d, where N is the number of nodes in the system and d is the number of directories. This is because each directory contains a random sample of size N/d, and the characteristics of this sample do not depend on N for a large enough N. A similar argument holds for the one-to-one scheme also. Thus, *all three schemes do very well in terms of scalability.* In the remainder of this section we consider only simulations with 4096 nodes. However, from the above reasoning, the results should hold for larger systems as well.

Efficiency: Next, we look at the efficiency of the different schemes in terms of the amount of load transferred. Fig. 1 plots the the total load moved (as a fraction of the total load of the system) as a function of $\bar{L}/\bar{T}$ for different schemes when the load is Pareto-distributed. Due to space limitations we do not present the results for Gaussian distribution. However, we note that, not surprisingly, all schemes perform better under Gaussian distribution. The plot in Fig. 1 shows the trade-off between the slack δ (defined in Section 3) and the load moved in the system. There are two points worth noting. First, the load moved depends only on the distribution of the load and not on the particular load balancing scheme. This is because all three schemes do only "useful" moves and hence *move only the minimum required to make nodes light.*

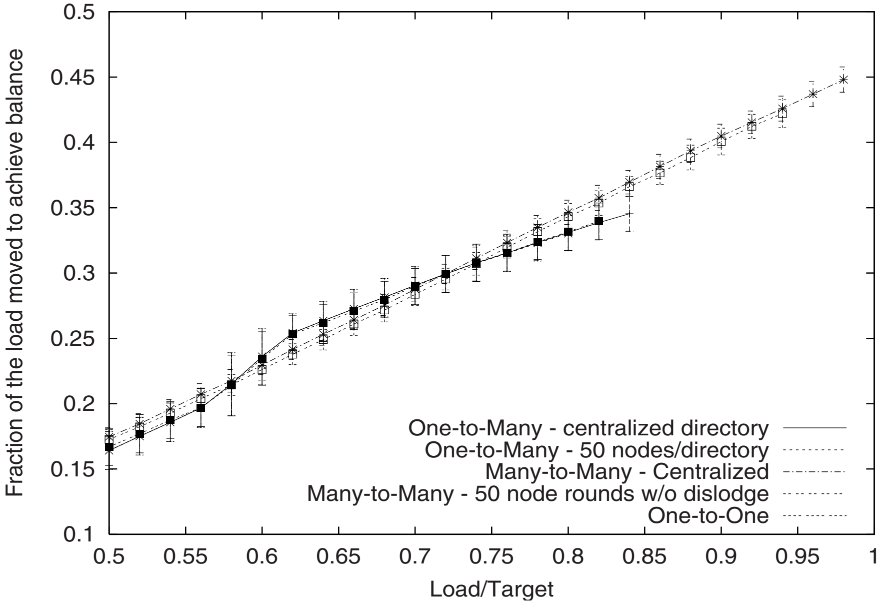

Fig. 1. The fraction of the total load moved for different schemes. In the beginning, each node is assigned 5 virtual servers at random

Second, we plot a point in this graph only if all 10 runs of the simulation result in a scenario where *all nodes are light*. This means that the range on the x-axis of a line is the range of loads over which the algorithm converged. Thus, the many-to-many scheme is capable of balancing the load within a factor of 0.94 from the ideal case (i.e., the case when $\delta = 0$), whereas the other two algorithms are able to balance the load only within a factor of 0.8 from the ideal case. The reason why the many-to-many scheme performs better than the other two schemes is that it uses a best-fit heuristic to match a set of heavy nodes to a set of light nodes. In contrast, with the first two schemes, there is a higher chance that a very light node i may not be able to accept a large virtual server v, despite the fact that $load(v) < T_i - L_i$. This can happen when other heavy nodes with smaller virtual servers contact node i first. To conclude, *the many-to-many scheme is capable of achieving balance even at very small* δ within a reasonable number of rounds [2], whereas the other two schemes cannot.

Total Number of Probes: The above results show that the one-to-one scheme achieves similar results with respect to load moved and the quality of balance achieved as the one-to-many scheme. But the main problem for the one-to-one scheme is the number of probes, which negatively impacts both the convergence

[2] Approximately 50 rounds with the Pareto distribution with $\bar{L}/\bar{T} = 0.94$.

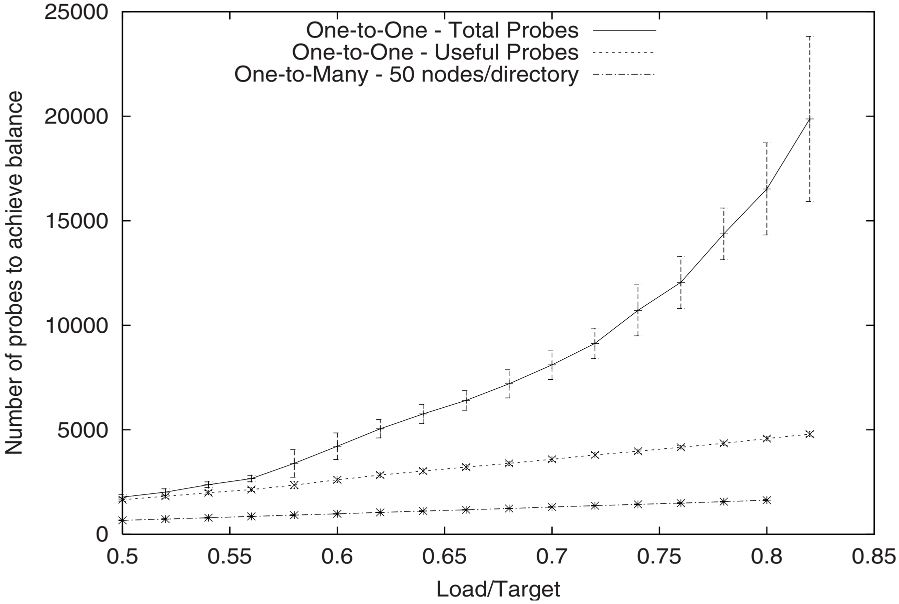

Fig. 2. The number of probes (total over all nodes) required for all nodes to become light

time and the control traffic overhead. To quantify this overhead, in Fig. 2 we plot the total number of probes performed by heavy nodes before they completely shed their excess load. A probe is considered *useful* if it results in the transfer of a virtual node. This graph shows that the *one-to-one scheme may be sufficient if loads remain stable over long time scales*, and if the control traffic overhead does not affect the system adversely.

Effect of Size of Directory: Given that we need to look at more than two nodes at a time to reduce the number of probes, the question arises as to how many nodes we must look at to perform efficiently in terms of control traffic. To answer this question, we look at the effect of the size of the directory on the number of probes in the one-to-many scheme. In Fig. 3, the x-axis shows the average size of a directory N/d, and the y-axis shows the total and the useful number of probes performed. Note that the initial number of heavy nodes is a lower bound on the number of probes. The graph shows that *even when* $N/d = 16$*, most heavy nodes are successful in shedding their load by probing* only *one directory*. We have observed the same trend in the number of rounds taken by the many-to-many scheme.

Trade-Off Involved in Doing Dislodge: In the the many-to-many scheme with dislodge, it is no longer true that only useful moves are done. We found

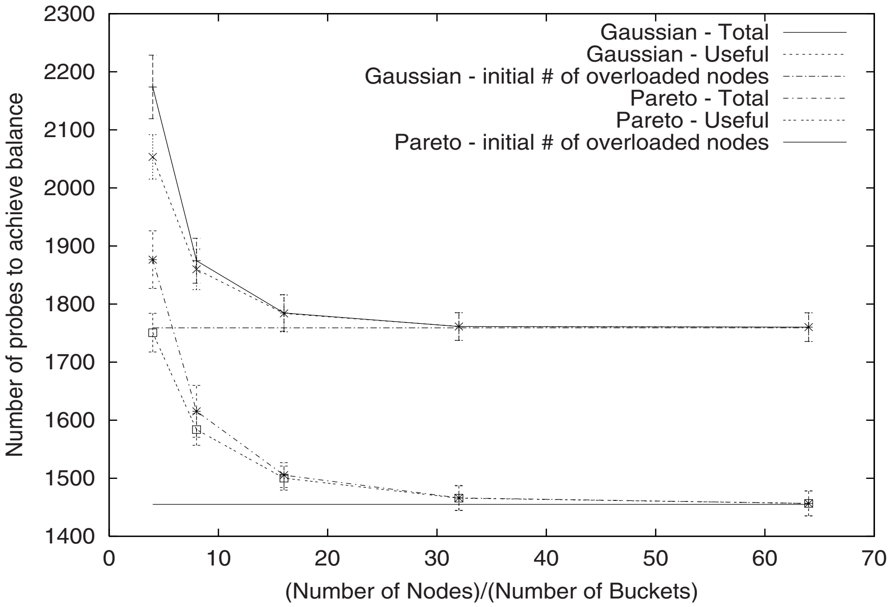

Fig. 3. The number of probes needed as a function of the expected number of nodes that will get hashed into a single directory ($\bar{L}/\bar{T} = 0.75$)

that with dislodge enabled, around 10% to 20% more of the total load was being moved than with dislodge disabled. The natural question is whether the extra load moved is justified by the reduction in the number of rounds. Fig. 4 shows that dislodging is useful only when $\bar{T}$ is very close to $\bar{L}$ even if we are trying to optimize only for the number of rounds. Also, note that even when the number of rounds required to make *all* nodes light is as high as 40, almost 95% of the heavy nodes became light after the first round. So, we conclude that *dislodge may not be a useful operation in practice.*

5 Related Work

Most structured P2P systems ([1, 2, 4, 5]) assume that object IDs are uniformly distributed. Under this assumption, the number of objects per node varies within a factor of $O(\log N)$, where n is the number of nodes in the system. CAN [1] improves this factor by considering a subset of existing nodes (i.e., a node along with neighbors) instead of a single node when deciding what portion of the ID space to allocate to a new node. Chord [2] was the first to propose the notion of virtual servers as a means of improving load balance. By allocating $\log N$ virtual servers per real node, Chord ensures that with high probability the number of objects per node is within a constant factor from optimal. However, to achieve

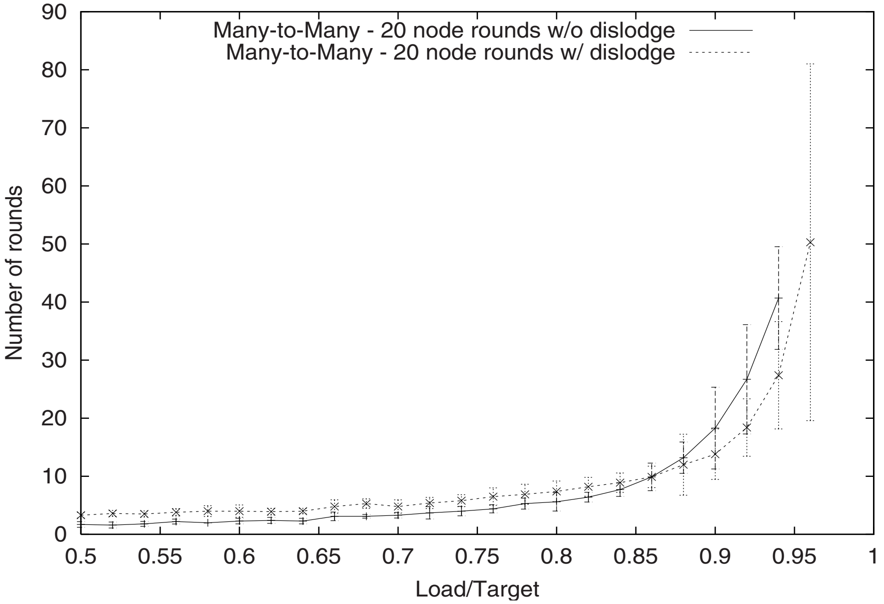

Fig. 4. The effect of dislodges on the number of rounds. The load distribution is Gaussian

load balancing, these schemes assume that nodes are homogeneous, objects have the same size, and object IDs are uniformly distributed.

CFS [3] accounts for node heterogeneity by allocating to each node, some number of virtual servers proportional to the node capacity. In addition, CFS proposes a simple solution to shed the load from an overloaded node by having the overloaded node remove some of its virtual servers. However, this scheme may result in thrashing as removing some virtual nodes from an overloaded node may result in another node becoming overloaded, and so on.

Douceur and Wattenhofer [7] have proposed algorithms for replica placement in a distributed filesystem which are similar in spirit with our algorithms. However, their primary goal is to place object replicas to maximize the availability in an untrusted P2P system, while we consider the load-balancing problem in a cooperative system.

Triantafillou *et al.* [9] have recently studied the problem of load-balancing in the context of content and resource management in P2P systems. However, their work considers an unstructured P2P system, in which meta-data is aggregated over a two-level hierarchy. A re-assignment of objects is then computed using the aggregated global information.

6 Conclusions and Future Work

We have presented three simple techniques to achieve load-balancing in structured peer-to-peer systems. The simulation results demonstrate the effectiveness of these schemes by showing that it is possible to balance the load within 95% of the optimal value with minimal load movement.

We plan to extend this work along three directions. First, we plan to study the effectiveness of our schemes in a dynamic system where items are continuously inserted and deleted, or/and where the access patterns of the items changes continuously. Second, we plan to develop the theoretical underpinnings of the proposed schemes. This would allow us to study the trade-offs between the transfer overhead and the effectiveness of each scheme better. Third, we plan to build a prototype of the load-balancing schemes on top of the Chord lookup system.

References

[1] S. Ratnasamy and P. Francis and M. Handley and R. Karp and S. Shenker. “A Scalable Content-Addressable Network”, *Proc. ACM SIGCOMM 2001.* 68, 77

[2] I. Stoica and R. Morris and D. Karger and M. F. Kaashoek and H. Balakrishnan. “Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications”, *Proc. ACM SIGCOMM 2001.* 68, 77

[3] F. Dabek and M. F. Kaashoek and D. Karger and R. Morris and I. Stoica. “Wide-area Cooperative Storage with CFS”, *Proc. ACM SOSP 2001.* 69, 78

[4] K. Hildrum and J. Kubiatowicz and S. Rao and B. Y. Zhao. “Distributed Object Location in a Dynamic Network”, *Proc. ACM SPAA, 2002.* 68, 77

[5] A. Rowstron and P. Druschel. “Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems”, Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001. 68, 77

[6] M. Harren and J. M. Hellerstein and R. Huebsch and B. T. Loo, S. Shenker and I. Stoica. “Complex Queries in DHT-based Peer-to-Peer Networks”, *Proc. IPTPS 2002.* 69, 70

[7] J. R. Douceur and R. P. Wattenhofer. “Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System”, *Lecture Notes in Computer Science, Vol. 2180, 2001.* 78

[8] J. R. Douceur and R. P. Wattenhofer. “Optimizing File Availability in a Secure Serverless Distributed File System”, *Proc. of 20th IEEE SRDS, 2001.*

[9] P. Triantafillou and C. Xiruhaki and M. Koubarakis and N. Ntarmos. “Towards High Performance Peer-to-Peer Content and Resource Sharing Systems”, *Proc. of CIDR, 2003.* 78