Load Balancing in Dynamic Structured Peer-To-Peer Systems[⋆]

Sonesh Surana[*], Brighten Godfrey,
Karthik Lakshminarayanan, Richard Karp, Ion Stoica

Computer Science Division, University of California at Berkeley, Berkeley, CA-94720, USA

Abstract

Most P2P systems that provide a DHT abstraction distribute objects randomly among "peer nodes" in a way that results in some nodes having $\Theta(\log N)$ times as many objects as the average node. Further imbalance may result due to nonuniform distribution of objects in the identifier space and a high degree of heterogeneity in object loads and node capacities. Additionally, a node's load may vary greatly over time since the system can experience continuous insertions and deletions of objects, skewed object arrival patterns, and continuous arrival and departure of nodes.

In this paper, we propose an algorithm for load balancing in such heterogeneous, dynamic P2P systems. Our simulation results show that in the face of rapid arrivals and departures of objects of widely varying load, our algorithm improves load balance by more than an order of magnitude for system utilizations as high as 80% while incurring an overhead of only about 6%. We also show that our distributed algorithm performs only negligibly worse than a similar centralized algorithm, and that node heterogeneity helps, not hurts, the scalability of our algorithm. Although many of these results are dependent on the workload, we believe the efficiency and performance improvement demonstrated over the case of no load balancing shows that our technique holds promise for deployed systems.

Key words: Dynamic Load Balancing, Structured Peer-to-Peer, Distributed Hash Table

[⋆] Conference version presented at Infocom 2004, Hong Kong
[*] *Corresponding Author*: Sonesh Surana, ph:+1-510-642-9669
Email addresses: `sonesh@cs.berkeley.edu` (Sonesh Surana), `pbg@cs.berkeley.edu` (Brighten Godfrey), `karthik@cs.berkeley.edu` (Karthik Lakshminarayanan), `karp@cs.berkeley.edu` (Richard Karp), `istoica@cs.berkeley.edu` (Ion Stoica).

1 Introduction

The last several years have seen the emergence of a class of structured peer-to-peer systems that provide a distributed hash table (DHT) abstraction [13,20,22,24].

A DHT manages a global identifier (ID) space that is partitioned among n nodes organized in an overlay network. To partition the space, each node is given a unique ID x and owns the set of IDs that are "closest" to x. Each object is given an ID, and the DHT stores it at the node which owns its ID. To locate the owner of a given ID, a DHT typically implements a greedy lookup protocol that contacts $O(\log N)$ other nodes, and requires each node to maintain a routing table of size $O(\log N)$, when there are N total nodes. The system provides an interface consisting of (at least) two functions: $put(id, item)$, which stores an item, associating with it a given identifier id; and $get(id)$ which retrieves the item with the identifier id.

If node and item identifiers are randomly chosen as in [13,20,22,24], there is a $\Theta(\log N)$ imbalance factor in the number of items stored at a node. Furthermore, if applications associate semantics with the item IDs, the imbalance factor can become arbitrarily bad since IDs would no longer be uniformly distributed. For example, a database application may wish to store all tuples (data items) of a relation according to the primary key using the tuple keys as IDs. This would allow the application to efficiently implement range querying (i.e., finding all items with keys in a given interval) and sorting operations, but would assign all the tuples to a small region of the ID space. In addition, the fact that in typical P2P systems, the capabilities of nodes (storage and bandwidth) can differ by many orders of magnitude further aggravates the problem of load imbalance.

Many solutions have been proposed to address the load balancing problem [24,14,1,18,17,16,20,11,9,15,7,8,25,19]. However, most make restrictive assumptions about the environment. In particular, none can handle heterogeneity in the system in terms of both node capacity and object load. In this paper, we present a solution for a system in which

- data items are continuously inserted and deleted,
- nodes of varying capacity join and depart the system continuously, and
- the distribution of data item IDs and item sizes can be skewed.

Our algorithm uses the concept of *virtual servers* previously proposed in [9]. A virtual server represents a peer in the DHT; that is, the storage of data items and routing happen at the virtual server level rather than at the physical node level. A physical node hosts one or more virtual servers. Load balancing is achieved by moving virtual servers from heavily loaded physical nodes to lightly loaded physical nodes.

In this paper we make the following contributions:

(1) We propose an algorithm which to the best of our knowledge is the first to provide dynamic load balancing in heterogeneous, structured P2P systems.
(2) We study the proposed algorithm by using extensive simulations over a wide set of algorithm parameters and system scenarios, in part derived from real-world trace data.

Our main results are as follows:

(1) Our simulations show that in the face of object arrivals and departures and systems loaded at up to 80% of their capacity, the algorithm achieves a good load balance while incurring an overhead of only about 6% in terms of the bandwidth needed to run the system. Furthermore, in a dynamic system where nodes arrive and depart, we achieve a similar result with 14% overhead.
(2) Compared to a similar fully centralized load balancer, our distributed algorithm produces a load balance less than 8% worse with overhead less than than 17% greater, showing that the price of decentralization is negligible.
(3) Heterogeneity of node capacity allows us to use many fewer virtual servers per node than in the equal-capacity case, thus increasing the scalability of the system.

The rest of the paper is organized as follows. In Section 2, we formulate the load balancing problem more explicitly and discuss what resources we may balance effectively. In Section 3, we discuss background material, including our use of virtual servers and our previous load balancing schemes given in [19]. In Section 4, we describe our algorithm for load balancing in dynamic P2P systems, and we evaluate its performance through simulation in Section 5. We discuss future directions in Section 6, related work in Section 7, and conclude in Section 8.

2 Problem formulation and motivation

2.1 Definitions and goals

Each object (data item) that enters the system has an associated *load*, which might represent, for example, the storage size of the object, the average bandwidth necessary to serve requests for the object, or the amount of processor time needed to serve the object. Thus, we do not assume a particular resource, but we assume that there is only one bottleneck resource in the system, leaving multi-resource balancing to future work.

Each object also has a *movement cost*, which we are charged each time we move the object between nodes. We assume this cost is the same regardless of which two nodes are involved in the transfer. In our simulations, we take movement cost to be the size of the object, and load to be the product of size and popularity.

The *load* ℓ_i on a node i at a particular time is the sum of the loads of the objects stored on that node at that time. Each node i has a fixed *capacity* $c_i > 0$, which might represent, for example, available disk space, processor speed, or bandwidth. A node's *utilization* u_i is the fraction of its capacity that is used: $u_i = \ell_i / c_i$. The *system utilization* μ is the fraction of the system's total capacity which is used:

$$\mu = \frac{\sum_{\text{nodes } n} \ell_n}{\sum_{\text{nodes } n} c_n}.$$

When $u_n > 1$, we say that node n is *overloaded*; otherwise node i is said to be *underloaded*. We use *light* and *heavy* to informally refer to nodes of low or high utilization, respectively.

A load balancing algorithm should strive to achieve the following (often conflicting) goals:

- **Minimize the load imbalance.** To provide the best quality of service, every node would have the same utilization. Furthermore, for resources with a well-defined cliff in the load-response curve, it is of primary importance that no node's load is above the load at which the cliff occurs. We can take this point to be the capacity of the node.
- **Minimize the amount of load moved.** Moving a large amount of load uses bandwidth and may be infeasible if a node's load changes quickly in relation to the time needed to move objects.

We formalize these goals in Section 5.

2.2 Relevance of load balancing

In this subsection, we answer two natural questions with respect to the relevance of load balancing in the context of two particular resources: storage and bandwidth.

Is load balancing of storage feasible? This question is raised by the huge disparity between the storage capacity of the end-hosts and the access bandwidth in a wide area network. Even if the end-hosts have a bottleneck bandwidth of 1.2 Mbps (higher than the uplink bandwidth of DSL and cable modem connections), it would take well over an hour to transfer 1 GB of data, which is not a large amount of data considering the fact that notebooks today come with 20-30 GB disks. Thus in many situations the amount of data movement necessary to significantly improve the load balance might not be achievable quickly enough. In spite of this fact, we believe that it is feasible to balance storage in contexts with higher node lifetime and bandwidth, such as in managed systems like PlanetLab (studied in [4]) or data centers with thousands or tens of thousand of machines connected by very high speed network connections (e.g., > 1 Gbps). Furthermore, bandwidth concerns can limit the use of DHTs even without load balancing [6]. Our experimental results suggest that our load balancer increases the DHT maintenance overhead by a small enough amount that it will be deployable in most cases where the underlying DHT itself is deployable.

Why use load balancing for bandwidth? For the purposes of relieving hot-spots, an alternative to load balancing is replication (caching). Why not replicate a popular data item instead of shifting sole responsibility for the popular item to a more powerful node? Replication, though a good solution in the case of immutable data, would require complex algorithms to maintain data consistency in the case of mutable data. Furthermore, many peer-to-peer systems are highly heterogeneous, with uplink capacity varying by two or more orders of magnitude [23]. Thus, moving a data item to a well-connected machine would be equivalent to generating and maintaining as many as 100 replicas of that data item, which may add significant overhead. Finally, we note that replication and load balancing are orthogonal and one can combine them to improve system performance.

3 Background

In this section, we argue for our design decision to use virtual servers as a fundamental unit of load balancing, and describe our earlier load balancing schemes on which the algorithm of this paper is based.

3.1 Use of virtual servers

One of the difficulties of load balancing in DHTs is that the load balancer has little control over where the objects are stored. Most DHTs use *consistent hashing* to map objects onto nodes [14]: both objects and nodes in the system are assigned unique IDs in the same identifier space, and an object is stored at the node with the "closest" ID in the space. This associates with each node a region of the ID space for which it is responsible. More generally, if we allow the use of virtual servers, a node may have multiple IDs and therefore owns a set of noncontiguous regions.

Under the assumption that we preserve the use of consistent hashing, the load balancer is restricted to moving load by either (1) remapping objects to different points in the ID space, i.e., changing objects' IDs, or (2) changing the region(s) associated with a node, i.e., changing nodes' IDs.

However, since items are queried by their IDs, changing the ID of an object would make it difficult to locate that object subsequently. Furthermore, some applications compute the ID of an object by hashing its content [9], thus rendering its ID static. We could attempt to fix these issues through indirection: the node owning ID x keeps only a pointer to the arbitrary node chosen to store the object whose ID is x. But indirection would not effectively manage the load of small but popular objects (e.g., tuples in a database relation) since all requests for the object still travel through the owner of ID x. Furthermore, indirection adds complexity and increases lookup time.

Thus we take the approach of changing the set of regions associated with a node. Since we wish to avoid large load movement, we need to be able to remap a small fraction of the ID space associated with a node. We ensure that the average number of regions (virtual servers) per node is large enough that a single region is likely to represent only a small fraction of a node's load.

There are several drawbacks to this approach. First, many DHTs insist that a node's identifier be a hash of its IP address, which makes it more difficult for a rogue node to usurp control of a particular region of the ID space. Our load balancer breaks this assumption since a virtual server may be placed on any node. Second, if there are an average of m virtual servers per node, the per-node routing state increases by a factor of m since a node must maintain the links associated with each of its virtual servers. However, as we will see in Section 5.4, we need a relatively modest number of virtual servers per node (e.g., $m = \log N$) to achieve good load balancing and substantially fewer when node capacities are heterogeneous. We believe this overhead is acceptable in practice. Furthermore, note that although the number of peers increases with m, in Chord, the lookup path length does not increase when shortcut routing is employed [9].

One of the main advantages of using virtual servers for balancing the load is that this approach does not require any changes to the underlying DHT. Indeed, the transfer of a virtual server can be implemented simply as a peer leaving and another peer joining the system. The ID-to-peer (i.e., ID-to-virtual server) and ID-to-object mappings that the underlying DHT performs are unaffected. If a node leaves the system, its share of identifier space is taken over by other nodes which are present in the system just as the underlying DHT would do. In the case of Chord [9], each virtual server v of a node that leaves the system would be taken over by a node that is responsible for a virtual server v' which immediately succeeds v in the identifier space. Similarly, when a node joins, it picks m random points in the ID space and splits the virtual servers there, thereby acquiring m virtual servers.

Since virtual server transfers are implemented as joins and leaves of peers, the resulting churn might adversely affect performance by causing an increase in the quantity of control traffic and in the time to fix routing entries. However, recent work shows that DHTs can handle churn with at low cost. Rhea et al [21] show that for the Bamboo DHT to function efficiently, only about 1KB/second/node of maintenance traffic is required when the median node session time is 1.4 minutes in a 1000-node system. Note that the cost of object movement is addressed in our model and simulations.

We assume that there are external methods to make sure that node departures do not cause loss of data objects. In particular, we assume that there is replication of data objects as proposed in CFS [9], and departure of a node would result in the load being transferred to the neighbors in the identifier space.

3.2 Static load balancing techniques

In a previous paper, we introduced three simple load balancing schemes that use the concept of virtual servers for static systems [19]. Since the algorithm presented in this paper is a natural extension of those schemes, we briefly review them here. The schemes differ primarily in the number and type of nodes involved in the decision process of load balancing.

In the simplest scheme, called *one-to-one*, each lightly loaded node v periodically contacts a random node w. If w is heavily loaded, virtual servers are transferred from w to v such that w becomes light without making v heavy.

The second scheme, called *one-to-many*, allows a heavy node to consider more than one light node at a time. A heavy node h examines the loads of a set of light nodes by contacting a random *directory node* to which a random set of light nodes have sent their load information. Some of h's virtual servers are then moved to one or more of the lighter nodes registered in the directory.

Finally, in the *many-to-many* scheme each directory maintains load information for a set of both light and heavy nodes. An algorithm run by each directory decides the reassignment of virtual servers from heavy nodes registered in that directory to light nodes registered in that directory. This knowledge of nodes' loads, which is more centralized than in the first two schemes, can be expected to provide a better load balance. Indeed, our results showed that the many-to-many technique performs the best.

Our new algorithm presented in the next section combines elements of the many-to-many scheme (for periodic load balancing of all nodes) and of the one-to-many scheme (for emergency load balancing of one particularly overloaded node).

4 Load Balancing Algorithm

The basic idea of our load balancing algorithm is to store load information of the peer nodes in a number of *directories* which periodically schedule reassignments of virtual servers to achieve better balance. Thus we essentially reduce the distributed load balancing problem to a centralized problem at each directory.

Each directory has an ID known to all nodes and is stored at the node responsible for that ID. Thus, ownership of a directory may change as the partitioning of the ID space among nodes changes, which can be due to node churn or to our own load balancing operations. Regardless, any node can contact any directory via the DHT's lookup protocol. We assume the number of directories is fixed. As we will see in Section 5.2, the performance of the system is quite stable as the number of directories varies.

Upon joining the system, a node n reports to a random directory (1) the loads $\ell_{v_1}, \ldots, \ell_{v_{m_n}}$ of the virtual servers for which n is responsible and (2) its capacity c_n. Each directory collects load and capacity information from nodes which contact it. Every T seconds, it computes a schedule of virtual server transfers among those nodes with the goal of reducing their maximum utilization to a parameterized *periodic threshold* k_p. After completing a set of transfers scheduled by a directory, a node chooses a new random directory and the process repeats.

When a node n's utilization $u_n = \ell_n / c_n$ jumps above a parameterized *emer-*

gency threshold k_e, it immediately reports to the directory d which it last contacted. The directory then schedules immediate transfers from n to more lightly loaded nodes, without waiting for its next periodic balance.

Finally, we describe how virtual servers are created and destroyed. When a node joins the system, it instantiates m_n virtual servers at random IDs according to the DHT's protocol. We choose m_n to be proportional to n's capacity c_n such that for nodes of average capacity, m_n is equal to a parameter m. When a node leaves, the DHT merges whichever virtual servers it currently owns with their neighbors. Since this does not guarantee that the number of virtual servers per node remains stable, we have each directory issue explicit commands to merge or split virtual servers if the number of virtual servers per node reporting to it is significantly different than the desired average m.

Below we give pseudocode for the algorithm run at each node n.

```
Node(time period $T$, threshold $k_e$)
• Initialization:
(1) $d \leftarrow$ RandomDirectory()
(2) $m_n \leftarrow \left\lfloor m \cdot c_n/\tilde{c}_d + \frac{1}{2} \right\rfloor$, where $\tilde{c}_d$ is the average capacity of nodes
    reporting to $d$
(3) Instantiate $m_n$ virtual servers at random IDs
(4) Send $(c_n, \{\ell_{v_1}, \ldots, \ell_{v_{m_n}}\})$ to $d$
• Emergency action: When $u_n$ jumps above $k_e$:
(1) Repeat up to twice while $u_n > k_e$:
(2)     Send $(c_n, \{\ell_{v_1}, \ldots, \ell_{v_m}\})$ to $d$
(3)     PerformTransfer($v, n'$) for each transfer $v \rightarrow n'$ scheduled by $d$
(4)     $d \leftarrow$ RandomDirectory()
• Periodic action: Upon receipt of list of transfers from a directory:
(1) PerformTransfer($v, n'$) for each transfer $v \rightarrow n'$
(2) Report $(c_n, \{\ell_{v_1}, \ldots, \ell_{v_m}\})$ to RandomDirectory()
```

In the above pseudocode, `RandomDirectory()` selects two random directories and returns the one to which fewer nodes have reported since its last periodic balance. This reduces the imbalance in number of nodes reporting to directories. `PerformTransfer`(v, n') transfers virtual server v to node n' if it would not overload n', i.e. if $\ell_{n'} + \ell_v \leq c_{n'}$. Thus a transfer may be aborted if the directory scheduled a transfer based on outdated information (see below).

Each directory runs the following algorithm.

```
Directory(time period $T$, thresholds $k_e, k_p$)
• Initialization: $I \leftarrow \{\}$
• Information receipt and emergency balancing: Upon receipt of $J = (c_n, \{\ell_{v_1}, \ldots, \ell_{v_m}\})$ from node $n$:
(1) $I \leftarrow I \cup J$
(2) If $u_n > k_e$:
(3)     reassignment $\leftarrow$ ReassignVS($I, k_e$)
(4)     Schedule transfers according to reassignment
• Periodic balancing: Every $T$ seconds:
(1) reassignment $\leftarrow$ ReassignVS($I, k_p$)
(2) Schedule transfers according to reassignment
(3) While average number of virtual servers per node in $I$ is $> 1.25m$ or
    $< 0.75m$, remove the least-loaded virtual server or split the largest-
    loaded virtual server in half, respectively
(4) $I \leftarrow \{\}$
```

The subroutine `ReassignVS`, given a threshold k and the load information I reported to a directory, computes a reassignment of virtual servers from nodes with utilization greater than k to those with utilization less than k. Since computing an optimal such reassignment (e.g. one which minimizes maximum node utilization) is **NP**-complete, we use a simple greedy algorithm to find an approximate solution. The algorithm runs in $O(r \log r)$ time, where r is the number of virtual servers that have reported to the directory.

```
ReassignVS(Load & capacity information $I$, threshold $k$)
 (1) pool $\leftarrow \{\}$
 (2) For each node $n \in I$, while $\ell_n/c_n > k$, remove the virtual server
     on $n$ with highest ratio of load to movement cost, and move it to
     pool.
 (3) For each virtual server $v \in$ pool, from heaviest to lightest, assign
     $v$ to the node $n$ which minimizes $(\ell_n + \ell_v)/c_n$.
 (4) Return the virtual server reassignment.
```

We next discuss several important design issues.

Periodic vs. emergency balancing. We prefer to schedule transfers in large periodic batches since this gives `ReassignVS` more flexibility, thus producing a better balance. However, we do not have the luxury to wait when a node is (about to be) overloaded. In these situations, we resort to emergency load balancing. See Section 5.1 for a further discussion of these issues.

Choice of parameters. We set the emergency balancing threshold k_e to 1 so that load will be moved off a node when load increases above its capacity. We compute the periodic threshold k_p dynamically based on the average utilization $\hat{\mu}$ of the nodes reporting to a directory, setting $k_p = (1+\hat{\mu})/2$. Thus directories do not all use the same value of k_p. As the names of the parameters suggest, we use the same time period T between nodes' load information reports and directories' periodic balances. These parameters control the tradeoff between load movement and quality of balance: intuitively, smaller values of T, k_p, and k_e provide a better balance at the expense of greater load movement. It is possible that a dynamic choice of T could be beneficial, but we do not investigate such strategies in this paper.

Stale information. We do not attempt to synchronize the times at which nodes report to directories with the times at which directories perform periodic balancing. Indeed, in our simulations, these times are aligned independently at random. Thus, directories do not perform periodic balances at the same time, and the information a directory uses to decide virtual server reassignment may be up to T seconds old.

Thrashing. A natural concern is that our algorithm could thrash, moving virtual servers repeatedly between nodes without significant improvement in load balance. At low system utilizations, this is unlikely to occur because load is only moved off a node if its utilization is significantly above the average. At high utilizations, we do not expect thrashing to occur because if a virtual server is scheduled to be transfered to an overloaded node, the transfer is aborted. The low load movement of our algorithm, demonstrated in the next section, confirms that thrashing is unlikely to be a problem.

5 Evaluation

We evaluate our load balancing algorithm through simulation. We show

- the basic effect of our algorithm, and the benefit of the combination of periodic and emergency action (Section 5.1);
- the tradeoff between load movement and quality of load balance, for various system and algorithm parameters (Section 5.2);
- the number of virtual servers necessary at various system utilizations (Section 5.3);
- our algorithm's robustness to scale and the effect of node capacity heterogeneity, concluding that we can use many fewer virtual servers in a heterogeneous system (Section 5.4);
- the effect of nonuniform object arrival patterns, showing that our algorithm is robust in this case (Section 5.5); and

- the effect of node arrival and departure, concluding that our overhead remains reasonable (Section 5.6).

Metrics. Our simulation studies the case wherein bandwidth is the constrained resource. Using the terminology introduced in Section 2.1, each object has a size s and a popularity p; its movement cost is s, and its load is $s \cdot p$. A node is *overloaded* when its utilization (load divided by capacity) is ≥ 1. Presumably, some or all of the requests directed to an overloaded node would fail.

We evaluate our algorithm using two primary metrics:

(1) *Movement ratio* is the total object movement cost incurred due to load balancing divided by the total object movement cost due to (1) initial insertion of objects, (2) serving object requests, and (3) the DHT's movement of objects when nodes arrive and depart. A movement ratio of 0.1 implies that the overhead of load balancing is 10% as much network bandwidth as the system must consume for normal operations.

(2) *Fraction of ill-fated requests* intuitively represents the fraction of end-users' requests which would be sent to overloaded nodes. More precisely we define this quantity at a particular time in a simulation trial to be

$$\frac{\sum_{o \in O'} popularity(o)}{\sum_{o \in O} popularity(o)},$$

where O is the set of all objects in the system and O' is the set of objects stored on overloaded nodes. Since some of these requests may actually be successful, this metric represents a pessimistic upper bound on the fraction of requests that would fail, assuming all requests to underloaded nodes succeed.

The challenge is to achieve the best possible tradeoffs between these two conflicting metrics.

Simulation methodology. Table 1 lists the parameters of our event-based simulated environment and of our algorithm, and the values to which we set them unless otherwise specified. Our object distribution is based on a nearly-complete subset of Gummadi et al's measurements of files stored in Kazaa [12] for 200 days in 2002, depicted in Table 2. Their data includes a size (in bits) and popularity (number of downloads per unit time) for each object. Each of our objects is picked uniformly at random from the set of objects in the trace, so that we may choose to have more or fewer objects than appear in the trace. Popularity is scaled to achieve a desired average system utilization. The trace of [12] did not measure node capacity, so we pick each node's capacity uniformly at random from the bottleneck upload bandwidth of Gnutella nodes as measured by [23], depicted in Figure 1.

Table 1
Simulated environment and algorithm parameters.

Environment Parameter	**Default value**
System utilization	0.8
Object arrival distribution	Poisson with mean inter-arrival time 0.01 sec
Object arrival location	Uniform over ID space
Object lifetime	Exponential; mean depends on system utilization
Average number of objects	1 million
Object load	popularity $\times$ size from Kazaa measurements of [12]
Object movement cost	size from Kazaa measurements of [12]
Number of nodes	4096 (no arrivals or departures)
Node capacity	From Gnutella measurements of [23]
Algorithm Parameter	**Default value**
Periodic load balance interval T	60 seconds
Emergency threshold k_e	1
Periodic threshold k_p	$(1+\hat{\mu})/2$ [a]
Number of virtual servers per node	12
Number of directories	16

[a] $\hat{\mu}$ is the average utilization of the nodes reporting to a particular directory.

We run each trial of the simulation for $20T$ simulated seconds, where T is the parameterized load balance period. To allow the system to stabilize, we measure our two metrics only over the time period $[10T, 20T]$. That is, to compute movement ratio, we sum the cost of movement performed in $[10T, 20T]$ and divide by the one-time movement cost of all objects that entered the system during $[10T, 20T]$. We sample the fraction of ill-fated requests every second over the period $[10T, 20T]$ and average the resulting $10T$ samples. Finally, each data point in our plots represents the average over 5 trials.

We note an important limitation of our simulator is that our model of a node's load is based only on the cost of serving objects stored on that node, and does not include the cost incurred by our load balancing operations. We measure that overhead separately via our movement ratio metric, but this does not reveal the distribution of the overhead among nodes. Accurate evaluation of overhead would be an important contribution of evaluating our technique through real-world experimentation rather than simulation.

Table 2
Summary of our object distribution, a subset of the Kazaa trace of [12]. Shown are the number of objects that fall into each (size, popularity) class.

	Popularity (# requests)		
Size	1-9	10-99	100-999
0-10 bytes	3	0	0
10-100 bytes	4	0	0
100-1000 bytes	14	0	0
1 KB - 10 KB	61	0	0
10 KB - 100 KB	312	0	0
100 KB - 1000 KB	10548	423	7
1 MB - 10 MB	471447	14807	371
10 MB - 100 MB	35336	3975	110
100 MB - 1000 MB	14885	2259	50
1 GB - 10 GB	64	0	1

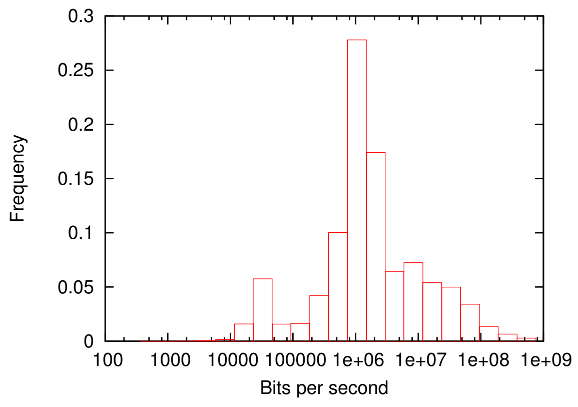

Fig. 1. Our capacity distribution: uplink bandwidth of Gnutella peers measured in [23], with bandwidth binned by the next smaller power of two.

5.1 Basic effect of load balancing

Figure 2 captures the tradeoff between movement ratio and ill-fated requests. Each point on the "Periodic + Emergency" line corresponds to the effects of our algorithm with a particular choice of load balance period T. For this and in subsequent plots wherein we vary T, we use $T \in \{1200, 600, 300, 120, 60\}$. The intuitive trend is that as T decreases (moving from left to right along the line), the fraction of ill-fated requests decreases but movement ratio increases. One has the flexibility of choosing T to compromise between these two metrics in the way which is most appropriate for the target application.

Figure 2 also demonstrates the value of the combination of periodic and emergency balancing by showing the effects of using only periodic or only emergency balancing. For the "Only Periodic" line, emergency balancing is turned off and we vary T as mentioned earlier. For the "Only Emergency"

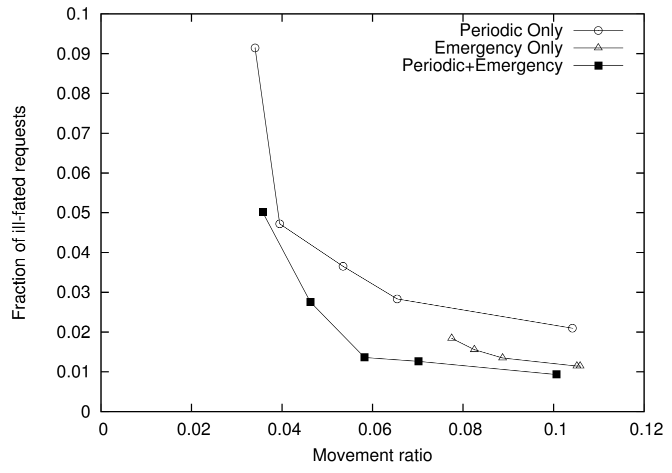

Fig. 2. Fraction of ill-fated requests vs. movement ratio, for our periodic+emergency algorithm, periodic only, and emergency only.

line, periodic balancing is turned off and we achieve different points in the tradeoff space by setting the emergency balancing activation threshold $k_e \in \{1, 0.975, 0.95, 0.925, 0.9\}$.

Intuitively, periodic balancing achieves a better instantaneous balance since it has the freedom to move virtual servers between any pair of nodes. However, during the interval between periodic balances, the load on a node may grow significantly due to object arrivals or node departures. In contrast, emergency balancing ensures that the load of a node never exceeds a given threshold (assuming there exist other nodes which can accept the load), but is constrained in the movement of virtual servers because it only moves load off of the single overloaded node. The combination of the two schemes produces a better tradeoff between node utilization and load movement than either scheme alone.

In the simulations of the rest of this paper, both emergency and periodic balancing are enabled as in the description of our algorithm in Section 4.

5.2 Movement ratio vs. fraction of ill-fated requests

With a basic understanding of the tradeoff between our two metrics demonstrated in the previous section, we now explore the effect of various environment and system parameters on this tradeoff.

In Figure 3, each line corresponds to a particular system utilization, and as in Figure 2, each line contains a point for several choices of T. In addition, we include a point representing the result of performing no load balancing. Note that these results are also sufficient to demonstrate the effect of object arrival rate and lifetime, since doubling T is equivalent in our simulation to halving

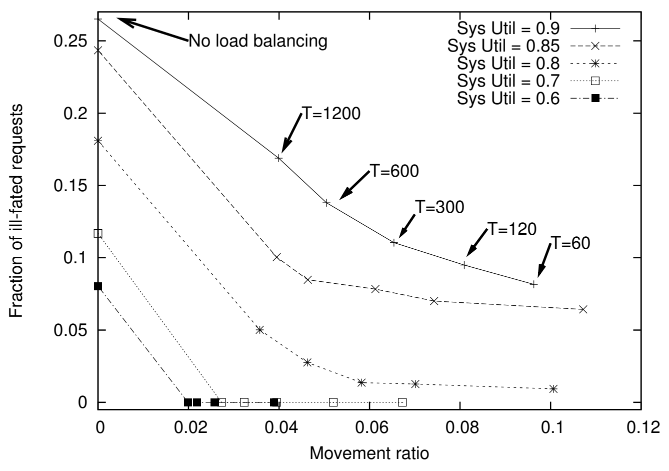

Fig. 3. Tradeoff between fraction of ill-fated requests and movement ratio as controlled by load balance period T, for various system utilizations.

the object arrival rate and lifetime.

With system utilizations up to $\mu = 0.7$, we are able to keep the fraction of ill-fated requests at zero, with a movement ratio of less than 3%. In contrast, with no load balancing, 11.6% of requests would be ill-fated at that utilization. The fraction of ill-fated requests remains high for all choices of T when $\mu > 0.8$.

Figure 4 shows that the tradeoff between our two metrics gets worse when the system contains fewer objects of commensurately higher load, so that the total system utilization is constant. For at least $750,000$ objects, which corresponds to 183 objects per node, we achieve good load balance with a movement ratio of $< 7\%$. The Kazaa trace of [12] found 633,106 unique objects, so our results indicate that to handle this number of objects we would need $\mu < 0.8$ or more than 12 virtual servers (the default values we used to produce Figure 4). This sensitivity to the number of objects demonstrates that uneven distribution of object load in the ID space is a significant problem for load balancing. A second source of imbalance is the partitioning of the ID space among virtual servers, which we study in Section 5.3.

Figure 5 shows that the number of directories in the system has only a small effect on our metrics. For any fixed T and any number of directories tested, movement ratio was never more than 23% greater than in the fully centralized case of case of one directory, and the fraction of ill-fated requests was never more than 16% greater. For our default choice of 16 directories, these numbers are 17% and 8% respectively. Thus, we pay only a small price for distributing the load balancing decisions.

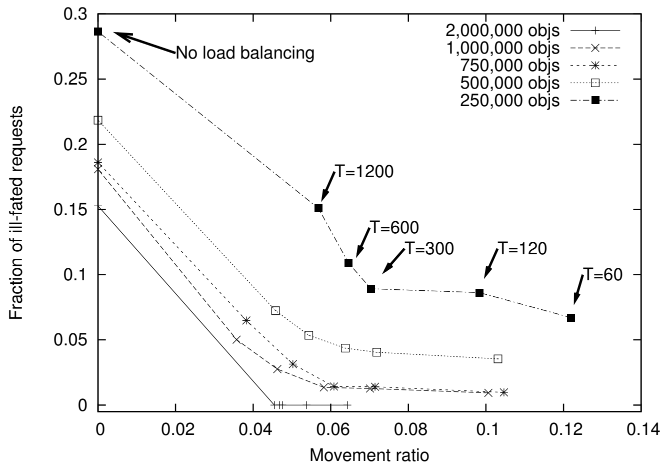

Fig. 4. Tradeoff between fraction of ill-fated requests and movement ratio as controlled by load balance period T, for various numbers of objects in the system.

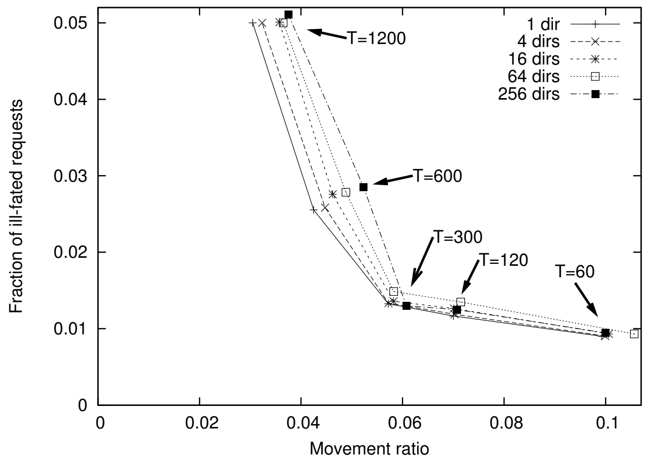

Fig. 5. Tradeoff between fraction of ill-fated requests and movement ratio as controlled by load balance period T, for various numbers of directories.

5.3 Number of virtual servers

Figures 6 and 7 plot our two metrics as functions of system utilization. Each line corresponds to a different average (over all nodes) number of virtual servers per node. In both metrics, there is a significant benefit to increasing the number of virtual servers from 2 to 8, with diminishing returns thereafter. The cost is increased overhead in maintaining these virtual servers' overlay links, which is not modeled in our simulator.

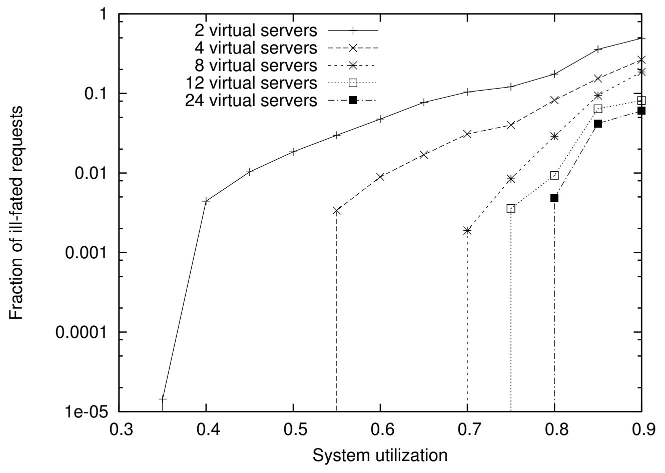

Fig. 6. Fraction of ill-fated requests vs. system utilization for various numbers of virtual servers per node. Note the log scale: points at $-\infty$ correspond to zero ill-fated requests (that is, all nodes are underloaded).

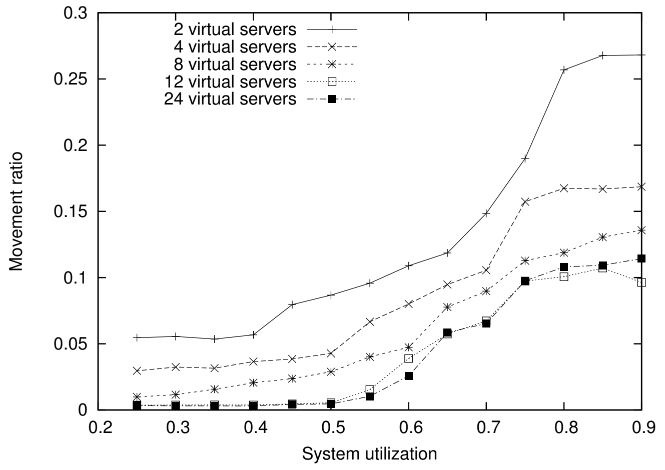

Fig. 7. Movement ratio vs. system utilization for various numbers of virtual servers per node.

5.4 Scaling and the effect of heterogeneous capacities

Figures 8 and 9 show our performance on both metrics is stable as the number of nodes scales. Note that a different common load balancing metric — the *maximum utilization* of a node – would be $\Theta(\log n)$ w.h.p. for a fixed number of virtual servers. Our metric is an average rather than a maximum, so we don't see this effect.

The figures also reveal the effect of the node capacity distribution. One set of lines corresponds to our default distribution taken from Gnutella hosts (labelled *heterogeneous* in the plot), and in the other set all nodes have the same capacity (*homogeneous*). In both cases the total system capacity is the

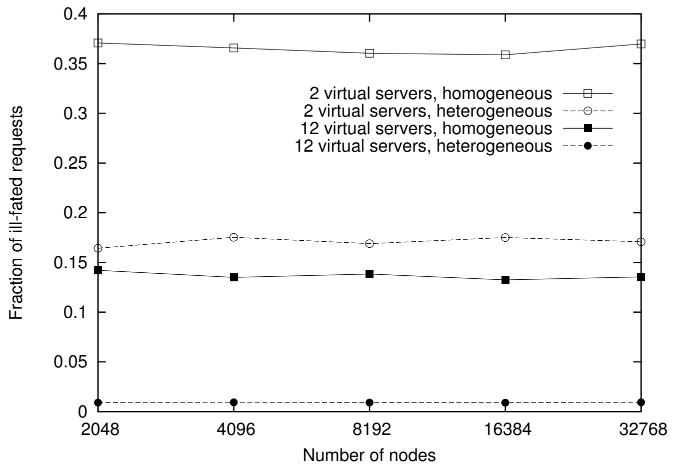

Fig. 8. Fraction of ill-fated requests vs. number of nodes for various numbers of virtual servers per node, with both *homogeneous* node capacities and *heterogeneous* capacities (default distribution).

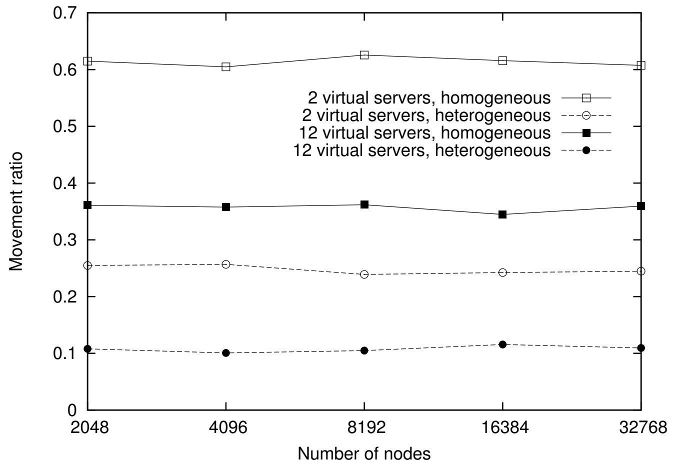

Fig. 9. Movement ratio vs. number of nodes for various numbers of virtual servers per node, with both *homogeneous* node capacities and *heterogeneous* capacities (default distribution).

same. Somewhat surprisingly, we achieve a much better load balance at lower movement cost in the heterogeneous case. Intuitively, this is because the virtual servers with very high load can be handled by the nodes with large capacities.

5.5 Nonuniform object arrival patterns

In this section we consider nonuniform object arrival patterns in both time and ID space.

We consider an “impulse” of objects whose IDs are distributed over a con-

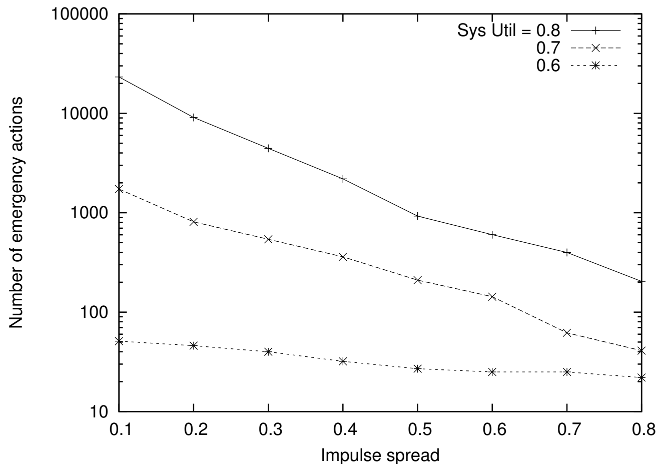

Fig. 10. Number of emergency actions taken vs. fraction of ID space over which impulse occurs, for various initial system utilizations.

tiguous interval of the ID space, and whose aggregate load represents 10% of the total load in the system. We vary the spread of the interval between 10% and 100% of the ID space. Thus, an impulse spread over 10% of the ID space essentially produces a rapid doubling of load on that region of the ID space, and hence a doubling of load on roughly 10% of the virtual servers (but not on 10% of the nodes since nodes have multiple virtual servers). The objects all arrive fast enough that periodic load balancing does not have a chance to run, but slow enough that emergency load balancing may be invoked for each arriving object. These impulses not only create unequal loading of objects in the ID space but also increase the overall system utilization in the short term.

To evaluate our algorithm's response to an impulse, we consider the *number of emergency load balance requests*. Note that since emergency load balancing can be invoked after each object arrival, some nodes may require multiple emergency load balances. Figure 10 demonstrates the intuitive fact that the number of emergency actions is high when the system utilization is high and the spread of the impulse is low. Results not depicted here indicate that even in the worst tested case – with the impulse spread over 10% of the ID space in a system at utilization 0.8 – only about 4% of the nodes initiate any emergency action.

Finally, Figures 11 and 12 plot our two main metrics in this setting. When the system utilization is high, the load moved is higher than the load of the impulse and a large fraction of the user requests go to overloaded nodes. However, having greater numbers of virtual servers helps significantly, in part because it spreads the impulse over more physical nodes.

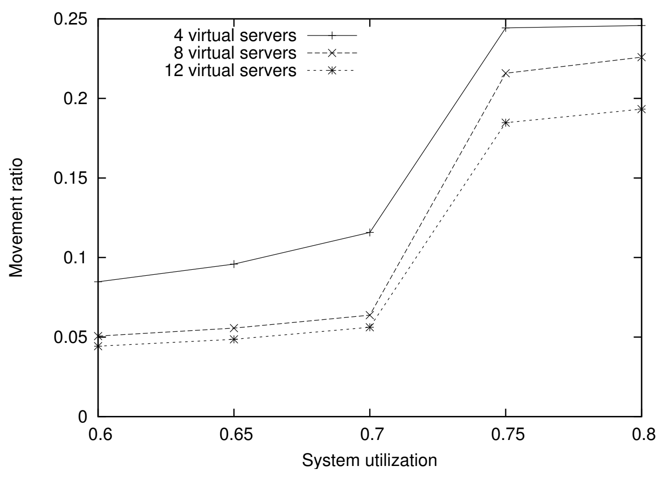

Fig. 11. Movement ratio vs. system utilization after an impulse spread over 10% of the ID space.

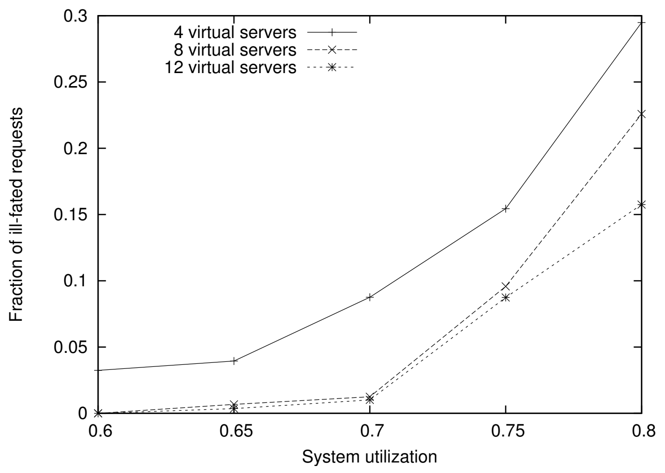

Fig. 12. Fraction of ill-fated requests vs. system utilization after an impulse spread over 10% of the ID space.

5.6 Node arrivals and departures

Note that all our previous experiments were conducted with a fixed set of nodes. In this section, we consider the impact of the node arrival and departure rates. Arrivals are modeled by a Poisson process, and node lifetimes are drawn from an exponential distribution. We vary interarrival time between 10 and 90 seconds. Since we fix the steady-state number of nodes in the system to 4096, a node interarrival time of 10 seconds corresponds to a node lifetime of about 11 hours.

Figure 13 shows that the quality of our load balance is only slightly worse in the highest-churn case than in the lowest-churn case. With a 10-second interarrival time, we begin to see a significant number of ill-fated requests at

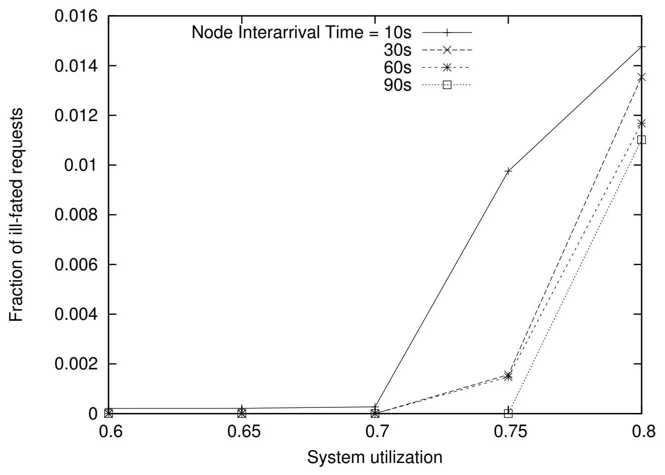

Fig. 13. Fraction of ill-fated requests vs. system utilization, for various node interarrival times.

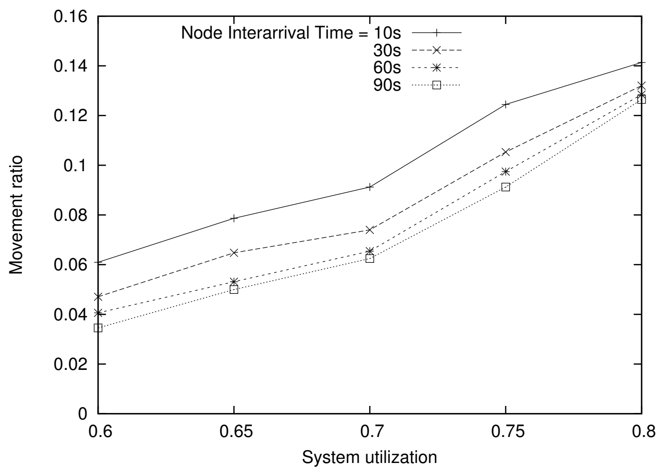

Fig. 14. Movement ratio vs. system utilization, for various node interarrival times.

a system utilization of 0.75 rather than 0.8 as in lower-churn cases.

Figure 14 shows that although movement ratio is greater than in the static case (Figure 7), the overhead is still acceptable. In the worst case, at system utilization $\mu = 0.6$, movement ratio increases over the static case by 57% to 6.1%. At $\mu = 0.8$, our overhead is still only 14%. More rebalancing is required as nodes arrive and depart, but the effect on movement ratio — which measures *relative* overhead — is mitigated by the fact that the underlying DHT must also move objects due to churn.

Additional experiments (not shown) demonstrate that this robustness to churn is also true for various numbers of virtual servers per node, between 4 and 24.

6 Future Work

The most important next step would be to evaluate an implementation of our technique, to rigorously examine effects that our simulator does not capture. Additionally, our algorithm would benefit from research in the following areas.

Virtual server management. It would be natural to have directories control all virtual server creation, so that arriving nodes cause the ID space to be partitioned in the way most expedient for load balancing, rather than at random IDs as in our present simulation. Indeed, a directory could add a virtual server whenever it found it convenient, while constraining the total number of virtual servers since each virtual server adds overhead. Note that when a node fails, its virtual servers are merged with their neighbors by the DHT protocol, so we cannot give directories full control over merging virtual servers without modifying the underlying DHT.

Smarter reassignment algorithm. Our virtual server reassignment algorithm is only a heuristic, and algorithms which come closer to the optimal balance while minimizing movement cost could be beneficial. Additionally, since volume of ID space is often closely correlated with rate of object arrival, we could reduce the chance that a node's load increases significantly between periodic load balances by avoiding the assignment of virtual servers with low load but large volume to nodes with little unused capacity.

Balance of multiple resources. In this paper we have assumed that there is only one bottleneck resource. However, a system may be constrained, for example, in both bandwidth and storage. This would be modeled by associating a load vector with each object, rather than a single scalar value. The load balancing algorithm run at our directories would have to be modified to handle this generalization, although our underlying directory-based approach should remain effective.

Beneficial effect of heterogeneous capacities. As shown in Section 5.4, having nonuniform node capacities allows us to use fewer virtual servers per node than in the equal-capacity case. It would be interesting to more precisely quantify the impact of the degree of heterogeneity on the number of virtual servers needed to balance load, and exploit that to dynamically control the number of virtual servers based on the capacity distribution.

7 Related Work

Load balance by fair ID space partitioning. A number of proposals improve load balance by partitioning the ID space more fairly. The simplest technique is for each node to simulate multiple *virtual servers* [24,14]. By allocating $\log N$ virtual servers per physical node, with high probability the maximum amount of ID space assigned to a node drops from $\Theta(\frac{\log n}{n})$ to $\Theta(\frac{1}{n})$. Since virtual servers can increase maintenence overhead, several proposals [1,18,17,16], evaluated theoretically but not under real workloads, give similar partitioning guarantees without the use of virtual servers. CAN [20] considers 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. Godfrey and Stoica [11] show how virtual servers can be adapted so that their overhead is minimal, but the technique cannot be applied to this paper because it does not allow movement of virtual servers between nodes. With the exception of [16], for a fair ID space partitioning to result in a good load balance, these schemes assume that object IDs are uniformly distributed, objects are similarly sized, and there are $\Omega(N \log N)$ objects. All except [11] assume nodes are homogeneous.

CFS [9] 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 servers from an overloaded node may result in another node becoming overloaded.

Load balance by object reassignment. The above strategies balance load by changing the assignment of IDs to nodes. Another approach is to assign data objects stored in the DHT to different IDs.

Karger and Ruhl [15] can handle capacities with limited heterogeneity and obtain a load balance within a constant factor of optimal. Each node periodically contacts another, and they exchange objects if one's load is significantly more than the other's. However, their bound on movement cost depends on the ratio of the maximum and minimum node capacities. Byers et al [7,8] use a "power of two choices" approach, hashing an object to $d \geq 2$ IDs and storing it on the corresponding node with least load, which results in a maximum load of $\log\log N / \log d + O(1)$ times optimal. Swart [25] places object replicas on a lightly-loaded subset of nodes in Chord's successor list. Neither [7], [8], nor [25] provide results for the case of heterogeneous nodes and objects. Object reassignment also requires placement of pointers from each object's canonical location to its current location so it can be found, which requires a pointer update whenever an object is moved, and adds latency to lookups.

Other work. Douceur and Wattenhofer [10] 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.* [26] 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.

There is a large body of theoretical work in load balancing problems similar to ours in that they seek to minimize both maximum load and amount of load moved. This includes Aggarwal et al [2] in an offline setting similar to that of our periodic load balancer, and Westbrook [27], Andrews et al [3], and others (see Azar's survey [5]) in an online setting. It would be interesting to study whether these algorithms can be adapted to our system.

8 Summary

We proposed an algorithm for load balancing in dynamic, heterogeneous peer-to-peer systems. Our algorithm may be applied to balance one of several different types of resources, including storage, bandwidth, and processor cycles. The algorithm is designed to handle heterogeneity in object load and node capacity, and dynamism in the form of (1) continuous insertion and deletion of objects, (2) skewed object arrival patterns, and (3) continuous arrival and departure of nodes.

Our technique has several drawbacks. We require multiple virtual servers per node to obtain a good balance, which increases maintenance overhead, and we cannot employ the security technique of requiring that a node's virtual server IDs are hashes of its IP address.

However, the results of our simulations on real-world data sets are promising: our algorithm is effective in achieving load balancing for system utilizations as high as 80% with low overhead, and performs only slightly less effectively than a similar but fully centralized balancer. In addition, we found that heterogeneity of the system can improve scalability by reducing the necessary number of virtual servers per node as compared to a system in which all nodes have the same capacity. We consider the next step of this reseach to be implementation of our algorithm on top of a real DHT.

Acknowledgements

We thank the authors of Gummadi et al [12] and Saroiu et al [23] for making their measurement data available to us.

References

[1] M. Adler, E. Halperin, R. M. Karp, and V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proc. STOC*, 2003.

[2] G. Aggarwal, R. Motwani, and A. Zhu. The Load Rebalancing Problem. In *Proc. ACM SPAA*, 2003.

[3] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. In *Proc. COCOON*, 1996.

[4] A. AuYoung, B. N. Chun, A. C. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proc. 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure*, October 2004.

[5] Y. Azar. *Online Algorithms - The State of the Art*, chapter 8, pages 178–195. Springer Verlag, 1998.

[6] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proc. HotOS IX*, 2003.

[7] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proc. IPTPS*, Feb. 2003.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Geometric Generalizations of the Power of Two Choices. In *Proc. SPAA*, 2004.

[9] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proc. ACM SOSP*, Banff, Canada, 2001.

[10] J. R. Douceur and R. P. Wattenhofer. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In *Proc. DISC*, 2001.

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

[12] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. SOSP*, 2003.

[13] K. Hildrum, J. D. Kubatowicz, S. Rao, and B. Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proc. ACM SPAA*, Aug. 2002.

[14] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. ACM STOC*, May 1997.

[15] D. Karger and M. Ruhl. New Algorithms for Load Balancing in Peer-to-Peer Systems. Technical Report MIT-LCS-TR-911, MIT LCS, July 2003.

[16] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proc. SPAA*, 2004.

[17] G. Manku. Randomized ID selection for peer to peer networks. In *Proc. PODC 2004*, 2004.

[18] M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *Proc. SPAA*, 2003.

[19] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proc. IPTPS*, Feb. 2003.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM*, San Diego, 2001.

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

[22] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. Middleware*, 2001.

[23] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. MMCN*, January 2002.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, pages 149–160, San Diego, 2001.

[25] G. Swart. Spreading the load using consistent hashing: A preliminary report. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, 2004.

[26] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards High Performance Peer-to-Peer Content and Resource Sharing Systems. In *Proc. CIDR*, 2003.

[27] J. Westbrook. Load balancing for response time. In *European Symposium on Algorithms*, pages 355–368, 1995.