Multimed Tools Appl (2006) 29: 325–339
DOI 10.1007/s11042-006-0015-5

On index load balancing in scalable P2P media distribution

Alok Nandan · Michael G. Parker · Giovanni Pau · Paola Salomoni

Published online: 11 July 2006

Abstract Peer-Peer (P2P) technologies have recently been in the limelight for their disruptive power in particular they have emerged as a powerful multimedia content distribution mechanism. However, the widespread deployment of P2P networks are hindered by several issues, especially the ones that influence end-user satisfaction, including reliability. In this paper, we propose a solution for an efficient and user-oriented keyword lookup service on P2P networks. The proposed mechanism has been designed to achieve reliability via index load balancing and address the scalability issues of extremely popular keywords in the index. The system performance have been analytically derived as well implemented using the OpenDHT framework on PlanetLab.

Keywords Peer-peer systems · Load balancing · Performance

Abbreviations
P2P Peer to Peer
DHT Distributed Hash Table

1 Introduction

File-Sharing applications such as Napster [10], and BitTorrent [9] have pushed peer-to-peer (P2P) technology to the forefront of public attention. These programs allow users to share

This work has been partially supported by the Italian Ministry for Research and Education under the E-Grid initiative, FIRB initiative, ex 60%, ex 40% and Interlink Initiatives, by NSF under the Overprobe contract, by the UC Discovery Grant: UC-Core/STM contract.

A. Nandan · M. G. Parker · G. Pau (✉)
Department of Computer Science, University of California, Los Angeles, CA 90095, USA
e-mail: gpau@cs.ucla.edu

A. Nandan
e-mail: alok@cs.ucla.edu

M. G. Parker
e-mail: mgp@cs.ucla.edu

P. Salomoni
Dipartimento di Scienze dell'Informazione, University of Bologna, Bologna 40126, Italy
e-mail: salomoni@cs.unibo.it

files with one another in a direct fashion, circumventing the centralized distribution model that dominates the Internet today. A new service model is emerging in the home video rental market. Typically, a customer pays a fixed monthly fee to get several movies (e.g., NetFlix). NetFlix [11] is not suitable for a "I want to see it now" scenario where users are not willing to tolerate a delay between the request and the delivery.

GhostShare a P2P application aims to deliver video content reliably and anonymously over the Internet. In particular GhostShare provides: (i) a *reliable name lookup service* to support advanced content search that includes a built-in index load-balancing, (ii) *anonymous downloading, searching and publishing capabilities* to protect user privacy and (iii) a download mechanism with a built-in content replication system to strengthen content availability. The proposed download mechanism, described in [9], is based on a swarming protocol that assures user collaboration while they are on line and active. Each user downloads the media in parts that are shared to the community immediately after downloading, even while the file in its entirety may be incomplete. Incentive mechanisms provided to customers to stay online and serve as relays during periods of inactivity can also be built-in. In the ideal scenario, the customer's subscription grants him or her and *access key* that allows the customer to join the network and download content. In this paper we assume the media content is encrypted with a strong encryption algorithm, however more sophisticated digital right management can support a "Virtual Rent and Return" operation according to the service agreement offered by the content provider. The "Virtualization of NetFlix" using the P2P paradigm needs to provide a minimal core infrastructure of nodes to guarantee the service is available at all times.

In this paper we present *GhostShare* and analyze the download balancing properties. Ghostshare is built on Pastry [15] substrate, however, we point that the mechanism presented are substrate agnostic. Load balancing is indeed relevant in distributing large files such as movies. Distribution of items is a core problem in any P2P platform, both in terms of items to be stored and computations to be carried out on the nodes. Typically, efficiency of P2P systems is measured in terms of fair work distribution among all peer nodes and load balancing becomes a key issue in increasing scalability and availability of services based on P2P architectures. Many solutions have been proposed to deal with load balancing in P2P systems [1, 4, 16], however the proposed idea of using keyword lookup and load balancing the index is our contribution.

The rest of the paper is organized as follows. Section 2 gives a background of the GhostShare protocol. We then describe the load balancing mechanisms in GhostShare in Section 2.3. Section 4 details our evaluation of the performance of the load balancing algorithm using simulation. Finally, Section 6 concludes the paper.

2 System description

In this section we describe the basic approach taken by GhostShare to provide an efficient load balancing and scalable content delivery to end users. We give a brief background on the Pastry substrate we used for our middleware layer.

2.1 Background

Each node in a Pastry [15] overlay network has a unique, randomly assigned 160-bit node ID, assuming a value in the address space $[0, 2^{160} - 1]$. Given a network-wide parameter b evenly dividing 160, even for the purposes of routing, node IDs can be thought of as a

sequence of $160/b$ digits in base 2^b. Every node maintains a routing table and a set known as the "leafset" to help it direct each message to anode whose node ID is numerically closest to the message's 160-bit key [15]. When a Pastry node receives a message with a given key, it first checks whether the key falls in the range of node IDs spanned by its set. If so, the node routes the message to the node in the "leafset" whose node ID is numerically closest to the key, which must be the closest node in the set of all existing nodes. Otherwise, using the routing table, the message is forwarded to a node whose node ID shares a prefix with the key one digit longer than the present node does. If no such node exists, the message is forwarded to a node whose node id matches a prefix of the same length, but whose node ID is numerically closer to the key [15]. By this method, the expected number of routing steps is $O(log_{2^b}N)$, where N is the number of Pastry nodes on the network.

2.2 Keyword lookup service

In this section we describe our main contribution, a novel keyword lookup mechanism on a Pastry network, with the benefits of redundancy and load balancing. The method of searching requires a hash function consistent with the hash function used to generate Pastry node IDs. For the purposes of this paper, we will assume this is the SHA-1 160-bit hash function [8].

When a user elects to share a file on the Pastry network, the hash function is applied to the file in two different ways. First, the hash function *digests* the contents of the entire file, generating a 160-bit hash for that file. Henceforth, this will be known as the content hash. By the so-called "birthday paradox," with very high likelihood, no two distinct files will generate the same content hash. Second, the hash function is applied to each keyword of the file, producing an array of 160-bit hashes for the filename. Henceforth, these will be called *keyword hashes*. Again, by the birthday paradox, we can assume no two distinct keywords will generate the same digest hash.

To understand how the hashes applied above, can enable keyword searching of files, let node Z have a file with filename "Matrix Revolutions" which it wishes to share. The method above will produce 2 distinct keyword hashes, X_1 and X_2, and a content hash Y. After this is completed, for each filename keyword hash X_i, a message containing the tuple $< X_i, Y, Z >$ is sent to the Pastry node whose ID is numerically closest to the filename keyword hash X_i. This can be done simply by choosing the destination node ID of the message, also called the message key, as the hash X_i. The numerically closest node to some filename keyword hash X_i knows that node Z has a file whose filename contains a keyword hashing to X_i, and whose contents hash to Y. If a new node joins the network that is closer to X_i than the current node simply transfers the tuple $< X_i, Y, Z >$ to the new node, preserving the property that the tuple is located at the node numerically closest to X_i (see figures 1 and 2).

Since this node can be the closest node to a wide interval of different keyword hashes, all associated with different content hashes and different node IDs, an efficient way is needed to index such data. One that is efficient and also facilitates returning search results is by maintaining two separate mappings–one from *filename keyword hashes→content hashes*, and another from *content hashes→node IDs*. The range of each of these mappings is a set, following the logic that different files can share the same filename keyword (therefore, multiple content hashes are associated with the same filename keyword hash), and different node IDs can share the same file (so multiple node IDs are associated with the same content hash). In this case, at the node numerically closest to X_i we create the mapping $X_i \rightarrow \{Y\}$ in the first table and $Y \rightarrow \{Z\}$ in the second.

Now suppose that node A is searching for *Matrix Revolutions.* After entering the filename into the GhostShare client, the same hash function is applied to each of its tokens,

producing the same filename keyword hashes X_1, X_2. For each filename hash X_i, node A sends a message to the node numerically closest to X_i, requesting a list of nodes sharing at least one file with a keyword hashing to X_i, along with their respective content hashes. The closest node can easily check if any node has shared a file with a keyword hash X_i by checking its existence in the domain of its *filename keyword hashes→content hashes*. In this case, for each filename keyword hash X_i, from the first mapping the node closest to X_i. From querying Y in the domain of its *content hashes→node IDs* mapping, the closest node to X_i can determine that Y is shared by node Z. Therefore all nodes numerically closest to X_1, X_2 will return the result $\{Y \rightarrow \{Z\}\}$. Node A can thus ascertain that node Z shares some file with content hash Y that contains keywords *Matrix* and *Revolutions* . Now all that is left is for node A to request from node Z the file with content hash Y, using the traditional Pastry implementation (figure 1). The keyword lookup service achieves index reliability through redundancy and supports substring searching as detailed below.

2.2.1 Substring searching

Lets consider a node A that searches for X' where all tokens in X' are contained in X, then A will still receive the mapping $\{Y \rightarrow \{Z\}\}$ because the filename keyword hashes generated from X' are a subset of $X_1, X_2, ..., X_n$. The proposed mechanism, moreover, distinguishes between files shared on the same filename, but different content. As an example, say that a document is shared in the plaintext and postscript format. Both have the same filename X, and therefore the plaintext version has a content hash of Y'. Therefore for

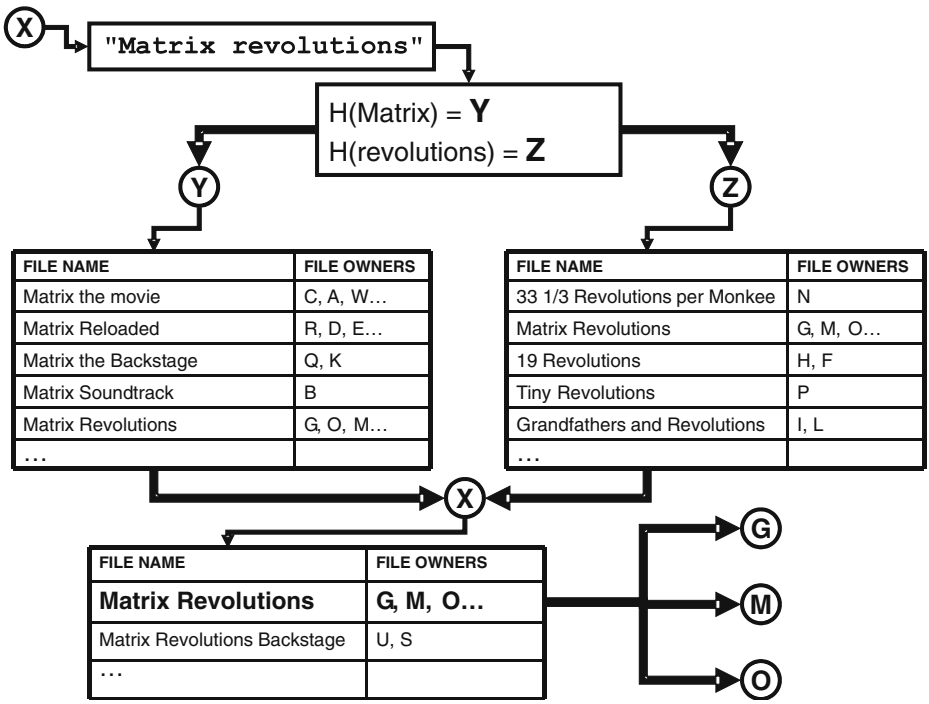

Fig. 1 Figure shows node X looking for "Matrix Revolutions". The keywords "Matrix" and "Revolution" are hashed using SHA1 and sent to index managed by the hash value closest node

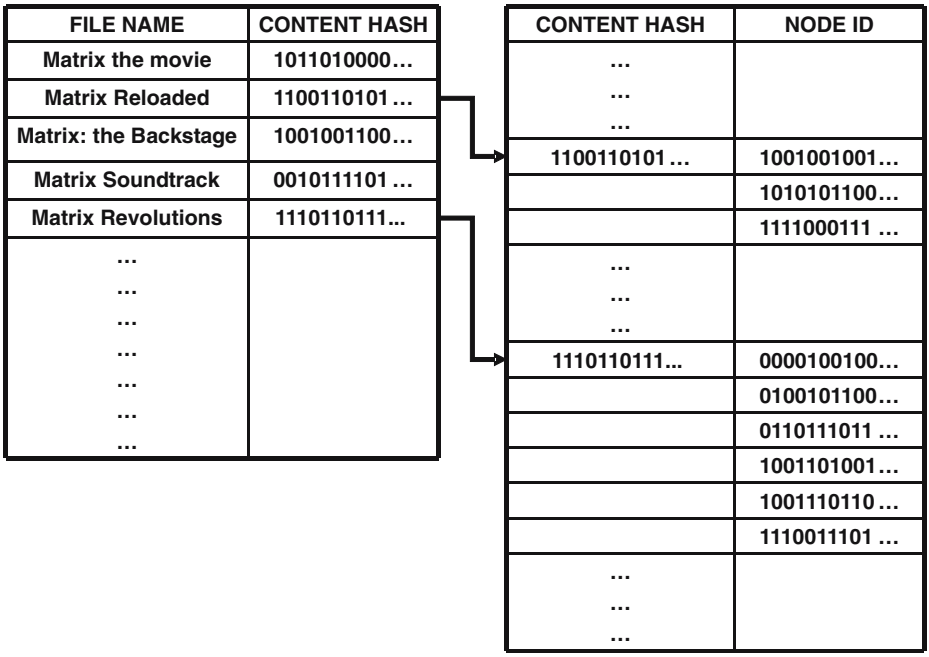

Fig. 2 Figure shows the mapping from the file name to the node I

each filename keyword hash X_i, the node numerically closest to X_i will maintain the mappings $X_i \rightarrow \{Y, Y'\}$ and $\{Y \rightarrow \{Z_1, Z_2, ..., Z_l\}, Y' \rightarrow \{Z'_1, Z'_2, ..., Z'_m\}\}$, where $Z_1, Z_2, ..., Z_l$ are the nodes sharing the plaintext version, and $Z'_1, Z'_2, ..., Z'_m$ are the nodes sharing the postscript version. Using the described above, a nodes searching for X will receive the results $\{Y \rightarrow \{Z_1, Z_2, ..., Z_l\}, Y' \rightarrow \{Z'_1, Z'_2, ..., Z'_m\}\}$—allowing the searching node to differentiate between the two different versions of the file, and which nodes share which version.

2.2.2 Index reliability

Peer to Peer networks are characterized by a very dynamic membership with frequent node arrival and departure [17]. In the event a node disconnects or fails unexpectedly, all *filename keyword hashes→content hashes* and *content hashes→node IDs* mappings that node maintained are lost. We would like to store this data redundantly so that we can avoid this dilemma. An ideal way to do this is to store each $< X_i, Y, Z >$ tuple not only at the node whose ID is closest to X_i, but at the node whose ID is second-closest to X_i. The aforementioned problem is averted since if the node closest to filename keyword hash X_i fails, a search for the closest node to X_i leads to the second-closest node to X_i, which also maintains the required mappings to provide a response with associated content hashes and node IDs. The rules to maintain this redundancy are simple when confronted with arriving or departing nodes: if the closest node to X_i drops, the second-closest node to X_i is now designated the closest node to X_i, and uses its leafset to find a new second-closest node to X_i, forwarding to it the $< X_i, Y, Z >$ tuple. If the second-closest node to X_i drops, the closest node to X_i finds a new second-closest node to X_i in its leafset, and again forwards the tuple. Because these two nodes are immediate neighbors, they should be intimately aware of each other's status, so one should be able to recreate this redundancy soon after

the other fails. When a new immediate neighbor appears in a node's leafset, that node must check whether it is still the closest or second-closest node to the filename keyword hash X_i. If not, that node must simply surrender the tuple $< Xi, Y, Z >$ to the new node to maintain this redundancy. This scheme can be reliably extended to k closest nodes, so long as all k closest nodes can see one another in their leafsets.

2.3 Distributed index load balancing

As described in Section 2.2 each node participating in the lookup service is the responsible index for the keyword hashes closest to it; suppose a given node is closest to a keyword hash X_i which belongs to a very popular file on the network. Every time this file is searched for, a message will travel to this node asking for content hashes associated with this filename keyword hash, and the nodes sharing these files, as described above. If this node has limited bandwidth, such messages could overwhelm the node and deteriorate its performance. Ideally, such requests for a particular filename hash like X_i would not be directed to a single node, but evenly distributed over multiple nodes. To accomplish this goal, we define a parameter d, such that for a chosen d we perform load balancing of a single hash X_i over 2^d nodes. We also make the assumption that no two distinct keyword hashes [5] will collide after their first d bits are omitted. For large values of n (such as the 160-bits of SHA-1) and relatively small values of d (approximately 3 or 4 is reasonable), again by the birthday paradox such a collision is unlikely. Our randomized query mechanism has been designed so that each replica shares a fair amount of load [6]. We evaluate the optimal value of d to achieve a desired query processing load at each node in subsequent section. Using the assumption that Pastry node IDs are uniformly distributed around the space of possible node IDs, it follows that the first d bits of Pastry nodes are uniformly distributed in the range $[0, 2^d)$. Thus, if k nodes share a file with a filename keyword hash X_i, on average $k/2^d$ nodes to permute the first d bits to the value 0, $k/2^d$ nodes to permute the first d bits to the value 1,..., and $k/2^d$ nodes to permute the first d bits to the value $2^d - 1$. Because the bitstring of each permuted hash after the first d bits is identical, the 2^d permutations of each X_i are distributed evenly around the space of possible node IDs, separated by a distance of 2^{160-d}. Now to search for mappings associated with a filename keyword hash X_i, all the inquiring Pastry client must do is randomly permute the first d bits of X_i and contact the resulting node; If this yields no results, X_i is permuted differently and the process repeats. It should be noted that this approach is independent of ring node density (figure 3).

The perfect load balance is achieved if all the nodes generate the permutations evenly, and consequently evenly query the 2^d replicas. GhostShare achieves an even query distribution through a distributed random algorithm. In order to search for a given keyword hash X_i, the requesting node will generate the proper replica permutation substituting the first d bits of the keyword hash with the result of its current clock in /*mu*sec modulo 2^d, as shown in figure 4. The requester will, thereafter, contact the resulting replica to retrieve all content hashes and node IDs associated with X_i.

2.4 Index maintenance

This section describes the maintenance procedures for the proposed load balancing method. We identify two main scenarios:

1. One or more replicas fail to respond to a query (sudden failure). In this case the information stored in the considered node is lost.

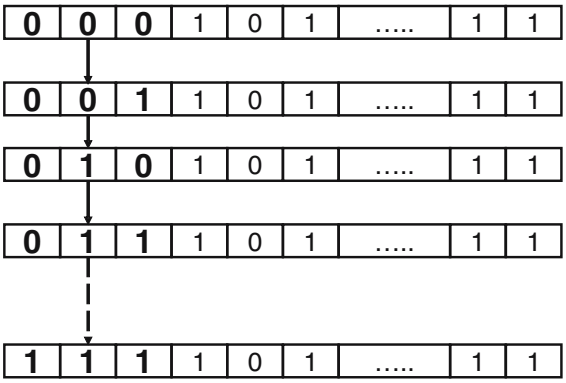

Fig. 3 Figure shows how to permute the first d bits to distribute the load among the 2^d replicas

2. One or more content owner suddenly disappears from the network.

The client sets a timeout for each query. If a replica fails to answer to a query before the timeout expires, that replica is marked temporarily down and a new one is selected. The content owner, in order to limit the presence of invalid records in the keyword index, is also requested to perform a keep-alive procedure with all the replicas. This procedure allows the content owner to discover replicas and eventually dead and republish the content on a new one. A fairly large timeout is set for the keep-alive procedure, failing to complete it in time causes the relative information to be discarded by the replicas. When a node (either replica or content owner) leaves the network intentionally a clean exit procedure is performed. In particular when a replica leaves the network index is handed to the closest neighbor as well when a content owner leaves sends a leave message to all the replicas so they can properly clean the index.

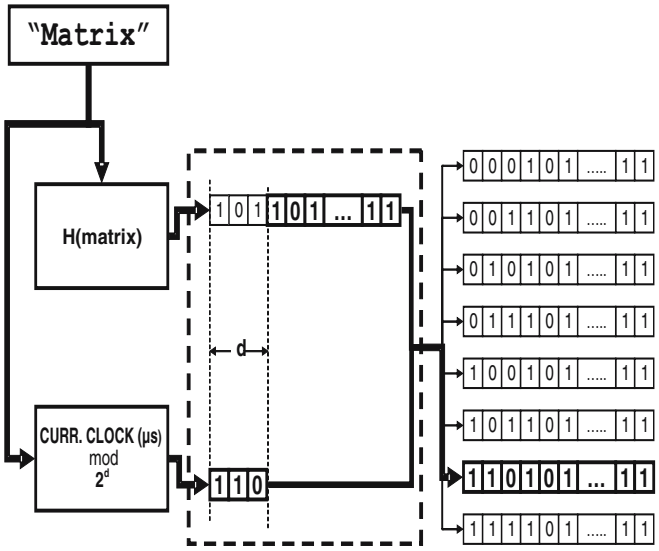

Fig. 4 The figure shows how our algorithm uses the clock to choose a replica that has to comply with the request

3 Analysis of index load balancing

In this section we analyze the index load balancing technique proposed in GhostShare to handle the scalability issue of popular keyword lookups. In particular we derive bounds on the degree of replication of the keyword indices.

Before delving into the details, we briefly describe some notation.

3.1 Notation

N is the number of unique items stored in the system,
K is the per node storage size in terms of index entries,
M is the number of nodes in the system,
r_i is the number of replicas of item i in the system, (for simplicity we assume that r_i is the same for all i),
λ is the arrival rate of nodes,
d the number of bits used for index load balancing purposes.
μ is the failure rate of individual nodes,
Q_{max} is the maximum query processing load of each node,[1]

We have multiple constraints on the system that determine the degree of replication required.

$$\sum_{i=1}^{N} r_i \leq K \cdot M \tag{1}$$

Under the simplifying assumption that r_i is same for all items we can derive a simple upper bound on the degree of replication and hence an upper bound on d, the number of bits to be flipped. From Eq. 1 above we get d bounded by $log(\frac{KM}{N})$ since r is 2^d according to the GhostShare P2P construction.

In order to derive the lower bound on the degree of replication and consequently, the value of d, we need to take into account the arrival and departure distributions of the nodes on the P2P network. We assume a Poisson arrival distribution and inter failure times to be exponentially distributed. This presents a simple model of the an M/M/1 queue and we are interested in the probability that the queue length is above a certain threshold.

Earlier results show that the query processing load Q_{max} is determined by the number of nodes in the P2P network and the degree of each node in the topology of the network in the following way.

$$Q_{max} = D^{log\ M'} \tag{2}$$

where D is the degree and M' is the currently number of available nodes. In our case, given a bound on the maximum query processing load that a node can sustain, we can determine the minimum number of nodes that are *required* to be present.

This value is, consequently, M' as $2^{\frac{log\ Q_{max}}{log\ D}}$. Hence we derive the lower bound on the value of d, the number of bits needed for replication as:

$$d \geq log\left(\frac{K.M'}{N}\right) \tag{3}$$

$$= log\left(\frac{K}{N}\right) + \frac{log(Q_{max})}{log(D)} \tag{4}$$

[1] Q could be different for different nodes

3.2 Summary

Hence to summarize the analytical results, we derived the bounds on d which determines the replication degree as follows:

$$log\left(\frac{K}{N}\right) + \frac{log(Q_{max})}{log(D)} \leq d \leq log\left(\frac{KM}{N}\right) \tag{5}$$

where K, M, N, d and Q_{max} are defined as above. The lower bound on the M' also determines the maximum arrival to failure rate ratio of the nodes in the GhostShare system. Assuming we can model the arrival process of nodes as a Poisson process and the inter failure time as a exponential process. We define ρ to be the fraction $\frac{\lambda}{\mu}$. We also know for an M/M/1 queuing system, the Expected queue length is $\frac{\rho}{\rho-1}$. In other words,

$$\frac{\rho}{\rho - 1} \geq 2^{\frac{log(Qmax)}{log(d)}} \tag{6}$$

Hence, Eqs. 5 and 6 give bounds on the degree of replication determined by the number of bit to be flipped and the bound of the arrival to departure ratio of the nodes in the GhostShare P2P network.

4 Experimental results

In order to evaluate our load balancing mechanism we used both simulations and actual experiments. The simulations have been conducted using a custom pastry simulator while the actual experiments have been deployed using the open DHT service in the *PlanetLab* testbed [12, 13]. The experiments have been designed to study the performance of the load balancing algorithm under different conditions; in particular we varied the query load and number of replicas for a given keyword. Intuitively the system reaches a perfect balancing if given a load of $k \gg 2^d$ users searching for a specific keyword, each replica is contacted by $k/2^d$ peers; to better summarize this concept we used the coefficient of variation (σ/μ) as a metric.

4.1 Simulation experiments

The simulation experiments have been designed to study our load balancing method for large scale networks (50 K nodes). During the experiments we considered both static and dynamic number of replicas.

4.1.1 Experiment A: static number of replicas

The aim of this experiment is to prove the effectiveness of our approach with large scale networks and very high workload; during this first set of experiments the number of replicas for the any given key does not changes through the experiment. In particular we performed several simulations assuming a query rate for a given, very popular, keyword has been set to 50% of the network size per second; the query arrival process distribution is Poisson. Each network node directs the query to the proper replica according the algorithm described in Section 2.3. We measured how many times each replica has been queried and

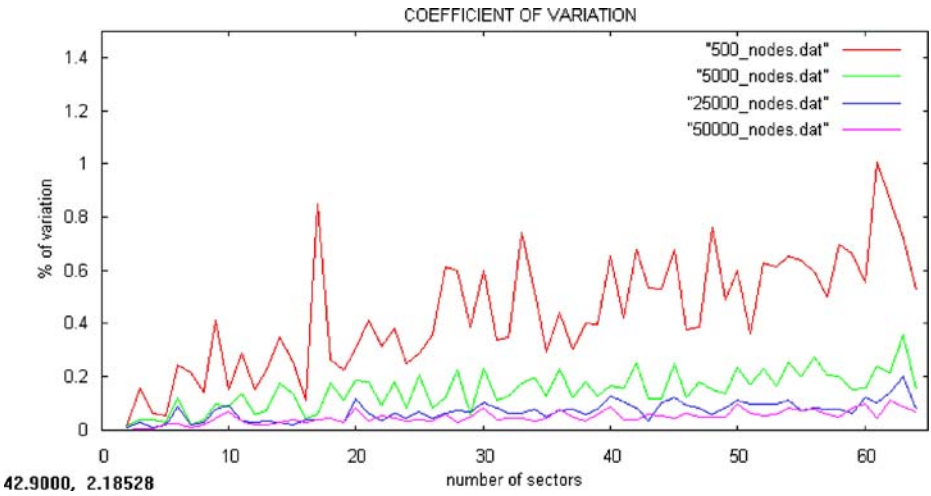

Fig. 5 The percentage is given by $((\sigma/\mu) * 100)$ and we can easily see from the figure that it is very very good (always under 1%). Moreover we show that we can scale easily maintaining good results

computed the coefficient of variation for each replica thereafter. In order to simulate a large spectra of scenarios we varied the number of replicas between 2 and 64 and increased the network size up to 50,000 nodes. For each experiment we selected a simulation time of 1,600 s. The results are summarized by figure 5 shows that in all the considered cases the coefficient of variation is always less than 1%, which proves this method's effectiveness for load balancing. Since each node computes the appropriate replica at any given time using its own clock no coordination between the nodes is required to perform the replica selection. It is worth noticing that the role played by the random nature of the algorithm where independent clocks on different nodes might fall in the same class of equivalence leading to the same replica. Figure 5, moreover, shows a larger variation in small networks where a relative small difference in the number of queries to a particular replica might result bigger coefficient of variation.

4.1.2 Experiment B: dynamic number of replicas

This experiment aims to study how the death of one or more replicas affects our load balancing technique. In particular we considered a network with a steady population of 50,000 nodes. The initial number of replicas for a given key varies with the considered scenario. Nodes are free to join and leave the network at any time. The query rate per key is chosen to be 25,000 per second according with a Poisson arrival distribution. The simulation time has been divided in four slices. At the beginning of each slice a percentage

Table 1 Network size 50,000 nodes; initial replicas: 8; query load 25,000 Qps

Total number of replicas	Number of dead replicas	σ/μ
8	0	0.015096
8	1	0.14707
8	2	0.052353
8	4	0.117597

Table 2 Network size 50,000 nodes; initial replicas: 16; query load 25,000 Qps

Total number of replicas	Number of dead replicas	σ/μ
16	0	0.027220
16	2	0.101759
16	4	0.112532
16	8	0.140332

of the replicas dies up to 50%; the replicas die in a random order. The coefficient of variation is then computed for the surviving replicas. A query directed to a dead replica is handled using the recovery strategy described in Section 2.3. The results are reported in Tables 1, 2 and 3.

4.2 Open DHT experiments

We implemented our mechanism using the Open DHT lookup service that in *PlanetLab* [12, 13]. PlanetLab is a distributed network testbed that offers access to a growing group of linux boxes. At the time of the experiments the group was counting about 300 machines were distributed all over the planet. Open DHT is a Chord based Lookup service developed by the University of California Berkeley and offered for experiments through standard Remote Procedure Call interface (RPC). The PlanetLab environment challenges new protocols and application with all the real Internet issues resulting in a great tool for actual test of new application layer protocols. We implemented our protocol on Open DHT and design a set of experiments suitable to study how the proposed mechanism performs in the real Internet.

We replicated the scenario discussed in Section 4.1.2 but due to the Open DHT constraints in number of available node our network size was 200 nodes. Open DHT, moreover, does not allow nodes to join/leave the Chord ring; we overcame these constraints marking nodes down at the application layer. For each given key the system generated about 150 query per second according the protocol described above. The results are shown in figures 6, 7 and 8 for 8, 16 and 32 replicas respectively. It is important to notice the role played by the network RTT and by the RPC interface; the first introduced a queue of requests at the replicas; while the later introduced queues and an artificial delay in the local client leading to a more unbalanced replicas. The results show that the network RTT and RPC negatively affect the coefficient of variation resulting in the order 8% for both the 8 and the 16 replica case slightly higher values have been reported in the 32 replica case; mostly due to the combined effects of RPC blocking calls and PlanetLab slice scheduling. We are now developing a new version suitable to run on Bamboo [13] directly over *PlanetLab*. The experiments carried in an actual network have arise new questions on the role of the RTT, *PlanetLab* time slicing, and RPC interface.

The experiments shown the effectiveness and efficiency of the proposed mechanism in achieving keyword search, index load balancing and reliability. It is worth noticing that the proposed approach is stateless and with a reducing signaling overhead.

Table 3 Network size 50,000 nodes; initial replicas: 32; query load 25,000 Qps

Total number of replicas	Number of dead replicas	σ/μ
32	0	0.024095
32	4	0.078224
32	8	0.073101
32	16	0.083413

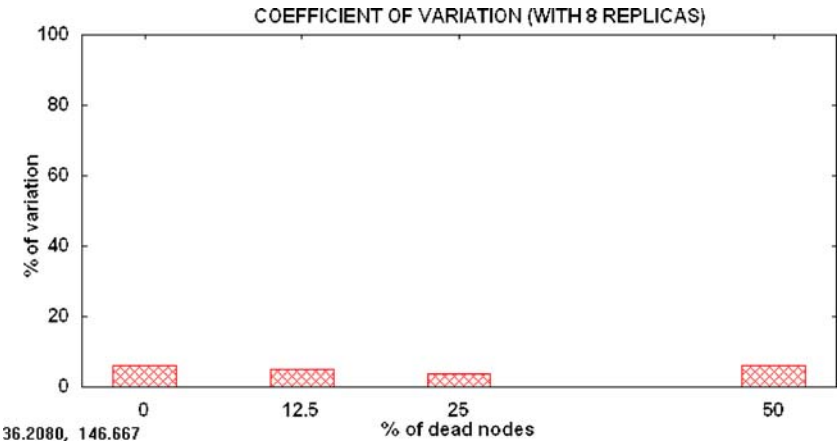

Fig. 6 The percentage is given by $((\sigma/\mu) * 100)$; the introduction of RTT delays and RPC overhead triggers higher values than before; although the results still prove the effectiveness of the proposed method

5 Related work

There are several load-balancing strategies have been proposed which we briefly discuss in this section.

Effective load balancing The system [14] is divided in a set of clusters of nodes and presents some techniques both for intra-cluster load balancing and for inter-cluster balancing. It uses two main techniques to move data from a node to another: migration and replication. Moreover it finds the best trade-off between these two techniques so to preserve both data availability and disk space on peers. Finally, a strategy for self-evolving the cluster s of peers is used to adjust the system to the changes of load of the net. The algorithm considers load as the number of megabytes received, the CPU power of the peer and the CPU power of the cluster to which the peer belongs. The clusters of peers are

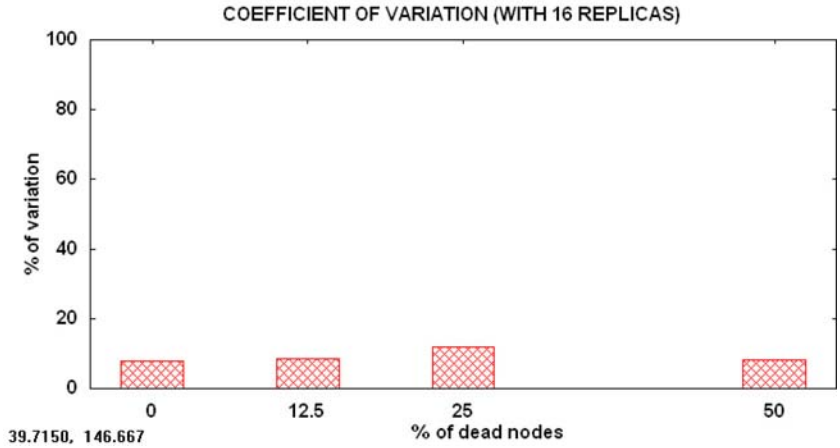

Fig. 7 The percentage is given by $((\sigma/\mu) * 100)$; the introduction of RTT delays and RPC overhead triggers higher values than before; although the results still shows the effectiveness of the proposed method

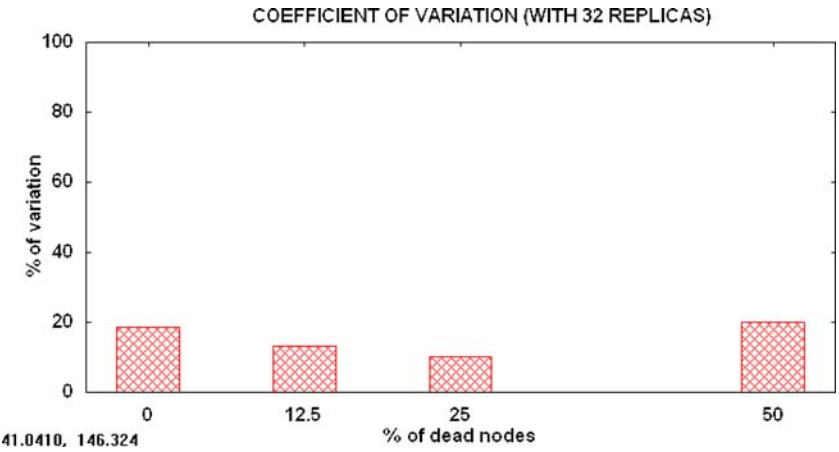

Fig. 8 The percentage is given by $((\sigma/\mu) * 100)$; the introduction of RTT delays and RPC overhead triggers higher values than before; although the results still shows the effectiveness of our method

disjoint and peers belonging to the same LAN are initially put in the same cluster. Each cluster independently chooses its leader, who has the job of coordinating the activities of the cluster and of maintaining information about the categories stored in its own cluster and in the neighboring ones. Decisions about using migration or replication to move data among nodes are taken at run-time by the cluster leaders. Replication increases data availability at the cost of disk space so that a periodic clean-up is needed. Using migration we save disk space but we pay in termsof overhead since we have to re-direct all the queries for data that have been moved. Disk space at the receiving peer is considered too in taking replication/migration decisions: non-hot data are stored at large capacity peers via migration for large-sized data and via replication for small-sized ones. Small sized hot data are replicated at small capacity peers to improve searching performance. Large-sized hot data are stored at large capacity peers via migration if such peers have low probability to leave the system or via replication otherwise.

P-Grid The approach taken [2] by this algorithm to solve load balancing problems consists in using nodes that dynamically change their associated key space independently from their identifier; the routing between peers is based on the associated key space rather than on the peer identifier. Following this approach the partitioning of the data space dynamically adapts to any granularity, until uniform distribution of data items over each partition of the key space is achieved. P-Grid, a particular kind of DHT, decouples the peer identifier from the associated key space and routing, and for this reason has greater flexibility for balancing load (both storage and replication). It provides a decentralized construction algorithm for building such a distributed data structure which adapts the key space partitioning to the data distribution to ensure storage load balancing. Even if the construction algorithm can create a P-Grid perfectly balanced, a maintenance mechanism is necessary in order to cope with changing membership. This approach has several advantages: keeping peer identifiers separated from the associated key space, it has not only great flexibility for load balancing, but it also preserves peer identity. It addresses multifaced load balancing concerns simultaneously in a self-organizing manner without needing general knowledge, or restricting the replication to a previously fixed number.

It maintains structural properties of a DHT without compromising searching efficiency. In P-Grid nodes refer to a common tree substructure to organize their own routing table. This tree must not be balanced, but can have any shape so that it can easily adapt to skewed distributions.

Power of two choices This algorithm [3] uses more than just an hash function to map the items to peers. Between the d hash functions used, it chooses the one that maps the item to the most lightly loaded peer. Using this method we obtain with high probability a maximum load of $(log\ logn/logd) + O(1)$.

Virtual servers One of the difficulties in load balancing in DHTs is that a user has little control over where its indexes are stored. Most DHTs use consistent hashing to map objects to nodes: both objects and nodes are assigned unique IDs in the same identifier space and an object is stored on the node with the closest ID. This associates [7] 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 many IDs and therefore owns a set of noncontiguous regions. Preserving the use of consistent hashing, the load balancer is restricted to moving a load by either remapping objects to different points in the ID space or changing the region (s) associated with a node. Since changing the ID of an object the searching will compromised, the only solution is to change the set of regions associated to a node. To do this, it reassigns an entire region from one node to another, but ensures that the 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. One of the main advantages in using virtual servers is that it doesn't require any modification to the underlying DHT.

6 Conclusion

A new service model is emerging in the home video rental market. Typically, a customer pays a fixed monthly fee to get several movies (e.g., NetFlix). This paper proposes a P2P system suitable for cooperative delivery in home video entertainment applications for future broadband-connected homes. Each user downloads the media in parts that are shared by the community immediately after downloading, even while the file in its entirety may be incomplete. Incentive mechanisms provided to customers to stay on-line. We introduce a novel keyword lookup algorithm and provide simple load balancing techniques to handle popular keyword searches. Using analytical model and experiments, we demonstrate the efficacy of the schemes and conclude that *GhostShare* can indeed achieves a scalable keyword search mechanism at the same time being an anonymous and P2P approach to content delivery.

Acknowledgments We are thankful to Georgui Velev and Amir Nader Teherani, for their suggestions, ideas, critics and support. We also would like to acknowledge the efforts of Paolo Bergamo, Dayana Cucchi, Gionata Ercolani, Matteo Morigi and Daniele Pieri for their help in the experiments and code development.

References

1. Aberer K, Datta A, Hauswirth M (2003) The quest for balancing peer load in structured peer-to-peer systems. Technical Report IC/2003/32, EPFL

2. Aberer K, Datta A, Hauswirth M (2005) Multifaceted simultaneous load balancing in DHT-based P2P systems: A new game with old balls and bins, Self-*Properties in Complex Information Systems, LNCS 3460, Springer
3. Byers J, Considine J, Mitzenmacher M (2004) Geometric generalizations of the power of two choices. In: Proceedings of the 16th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp 54–63
4. Castro M, Druschel P, Kermarrec A-M, Rowstron A (2002) Scribe: a large-scale and decentralized application-level multicast infrastructure. IEEE J Sel Areas Commun (JSAC) 20(8)
5. Jones PE, Eastlake D (2001) US Secure Hash Algorithm 1. Retrieved 2004, July 12, from the Internet Engineering Task Force web site: http://www.ietf.org/rfc/rfc3174
6. Philip J, Erdelsky PJ (2001) The birthday paradox, http://efgh.com/math/birthday.htm
7. Godfrey B, Lakshminarayanan K, Surana S, Karp R, Stoica I (2004) Load balancing in dynamic structured P2P systems. In: Proceedings of IEEE INFOCOM 2004, Hongkong, China
8. Handschuh H, Knudsen LR, Robshaw MJ (2001) Analysis of SHA-1 in encryption mode. In: CT-RSA 2001, vol 2020 of Lecture Notes in Computer Science, pp 70–83
9. http://bitconjurer.org/BitTorrent/
10. http://www.napster.com/
11. http://www.theregister.co.uk/2004/01/27/netflix_the_fly/
12. Karlin S, Paterson L (2003) PlanetLab: a blueprint for introducing disruptive technology into the internet. In: Presented at the joint Princeton ACM/IEEE Computer Society meeting
13. Karp B, Ratnasamy S, Rhea S, Shenker S (2003) Spurring adoption of DHTs with OpenHash, a public DHT service. In: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004)
14. Mondal A, Goda K, Kitsuregawa M (2003) Effective load-balancing of peer-to-peer systems, Data Engineering Workshop
15. Rowstron A, Druschel P (2001) Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp 329–350
16. Rowstron A, Kermarrec A-M, Castro M, Druschel P (2001) Scribe: the design of a large-scale event notification infrastructure. In: Crowcroft J, Hofmann M (eds.), Networked Group Communication, Third International COST264 Workshop (NGC'2001), Vol. 2233 of Lecture Notes in Computer Science, pp 30–43
17. Saroiu S, Gummadi PK, Gribble SD (2002) A measurement study of peer-to-peer file sharing systems. In: Proceedings of multimedia computing and networking 2002 (MMCN'02)

Giovanni Pau is a Research Scientist at the Computer Science Department of the University of California, Los Angeles. He has obtained the Laurea Doctorate in Computer Science and the PhD in Computer Engineering from the University of Bologna (Italy). He is currently serving as North America vice chair for the IEEE COMSOC Multimedia Technical Committee. His research interests include Wireless Multimedia, Peer to Peer and Multimedia Entertainment.

Paola Salomoni is an Associate Professor of Computer Science at the Department of Computer Science of the University of Bologna. In October 1992 she received the Italian Laurea degree (with honors) in Computer Science from the University of Bologna. Her research interests include distributed multimedia systems and services, tools and techniques for E-learning environments, computer entertainment.

Mike Parker is a Master Student at the Computer Science Department of the University of California, Los Angeles. He has obtained the Bachelor of Science degree in Computer Science at the University of California Los Angeles. He is joining Google inc effectively July 17th 2006.

Alok Nandan holds a Master and a PhD in computer science obtained at the University of California, Los Angeles. His research interests include Multimedia peer-to-peer systems, overlay networks and performance evaluation. He is currently working for Microsoft as program manager.