IEEE GLOBECOM 2004
DALLAS • TEXAS
Emerging Technologies, Applications and Services

29 November - 3 December 2004 • Dallas Texas
Hyatt Regency Dallas at Reunion Hotel

IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE WORKSHOPS

www.globecom2004.org

IEEE
IEEE COMMUNICATIONS SOCIETY
ICC GLOBECOM

GhostShare - Reliable and Anonymous P2P Video Distribution

Alok Nandan *, Giovanni Pau *,and Paola Salomoni †
* Department of Computer Science - University of California Los Angeles, CA 90095
e-mail: {alok|gpau}@cs.ucla.edu; ph: (310) 206-3212, fax: (310)825-7578
† Dipartimento di Scienze dell'Informazione
Universita' di Bologna, Italia - 40126
e-mail: salomoni@unibo.it - ph: +39 (0547) 642-813, fax: +39 (0547) 610-054

***Abstract*—**
P2P networks have emerged as a powerful multimedia content distribution mechanism. However, the widespread deployment of P2P networks are hindered by several issues, especially ones that influence end-user satisfaction, including privacy protection. In this paper, we propose *GhostShare*, a P2P network built on the Pastry substrate, to distribute video content. The primary design goals of GhostShare are anonymity and load balancing for participating peers. We present simulation results that prove the effectiveness of GhostShare's load balancing mechanism and provide an analysis of the anonymity scheme.

I. Introduction

File-Sharing applications such as Napster [1], KaZaA [2], and BitTorrent [3] have pushed peer-to-peer (P2P) technology to the forefront of public attention. These programs allow users to share files with one another in a direct fashion, circumventing the centralized distribution model that dominates the Internet. On a different front, the on-line market still uses centralized applications to provide costumers with multimedia content, as it provides control during download and streaming procedures. We believe P2P networks can emerge as a scalable platform supporting not just file sharing, but a wide market of decentralized multimedia entertainment and infotainment services. Several aspects of centralized multimedia content delivery, including scalablility of service and privacy protection, can be addressed using the P2P paradigm.

In this paper we present *GhostShare*, a P2P platform to support video distribution services. GhostShare is based on Pastry [4] substrate. Several mechanisms have been introduced to improve performance by using a load balancing algorithm during content search and multipath content delivery. Redundancy is achieved by two strategies: (i) a novel keyword searching on a Pastry network that has the benefits of redundant keyword mappings and (ii) an algorithm for establishing multiple disjoint paths between source and destination nodes. 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 [5], [6], [7] that frequently ignore the heterogeneity of such networks and reassign loads among nodes without considering the underlying P2P topology.

One of the primary design goals of GhostShare is providing privacy protection. New Internet services provide previously unforeseen opportunities for personalized multimedia experiences. However, a new issue is emerging, related to the protection of personal information that may be collected online. P2P networks pose this privacy threat as well. In particular, two main aspects concerning user privacy have to be considered in P2P distribution services.

Anonymity: The user who downloads a file is revealing her preferences to a network node that offers the resource. In video distribution services a lot information about user preferences could be collected by tracking the costumer activities at the provider side. To prevent this loss of privacy, the user ideally would like to operate in anonymous mode, asking for the resource without declaring her identity.

Secrecy: In P2P networks, the discovery request and the resources themselves move from the origin to the destination node by passing through several participants of the P2P network. To prevent the loss of privacy deriving from this distribution of personal information, the user has to maintain anonymity by the use of cryptography. GhostShare achieves anonymity using a probabilistic publish mechanism. We present a probabilistic analysis of GhostShare's anonymity approach. We provide simulation results to show the effectiveness of our load-balancing algorithm. The remainder of the paper is organized as follows: we first provide an overview of GhostShare in Section II; we describe the keyword search achieving with redundancy and load balancing in section II-A; next, in section II-C we introduce the algorithm for establishing multiple disjoint paths; a general discussion of the routing performance over multiple paths and simulation results are presented in section IV. We provide an analysis of the anonymity procedure in section III-E.

II. System Description

GhostShare is a peer-to-peer system that provides reliable and anonymous lookup and exchange of information. The

This work has been partially supported by the EGRID project funded by the *"Italian Ministry for Research and Education" and the UNESCO/IAEA "Abdus Salam" International Center for Theoretical Physics*; the *STMicroelectronics* under the UC Discovery Grant - CoRe

main features of GhostShare are applicable with no or minimal changes to most peer-to-peer substrate. We provide an implementation and evaluation on Pastry substrate [4]. Pastry provides an efficient and reliable platform to support distributed peer-to-peer applications such as file sharing, global storage, resource lookup, and other services [8]. We assume the reader is familiar with Pastry [4].

A. Keyword Lookup Service

In this section we describe a keyword lookup mechanism on a Pastry network, with the benefits of redundancy and load balancing. The approach is similar to the one proposed in [8], [10]. 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 [11].

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 filename, producing an array of 160-bit hashes for the filename. Henceforth, these will be called token 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 token hashes, $X_1 and X_2$, and a content hash Y. After this is completed, for each filename token 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 token 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 token hash X_i knows that node Z has a file whose filename contains a token 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, it must appear in the leafset of the current node. 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 token 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 token (therefore, multiple content hashes are associated with the same filename token hash), and different node IDs can share the same file (so multiple node

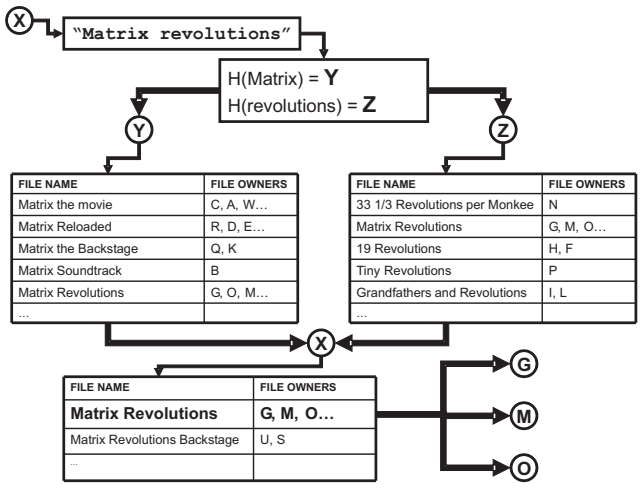

Fig. 1. **Figure shows node X looking for "Matrix Revolutions". The tokens "Matrix" and "Revolution" are hashed using SHA1 and sent to index managed by the hash value closest node.**

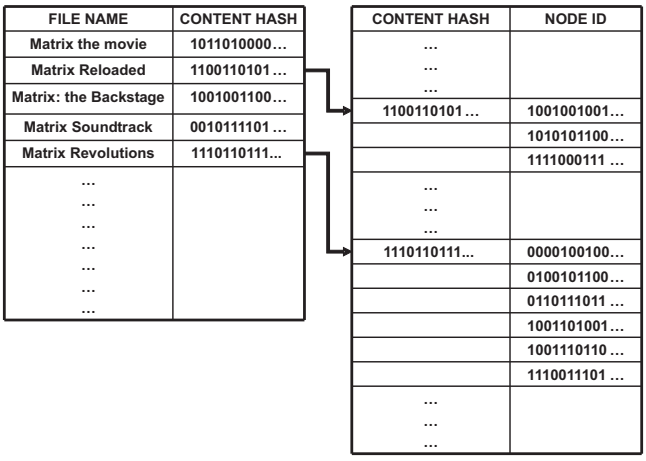

Fig. 2. **Figure shows the mapping from the file name to the node ID.**

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 $MatrixRevolutions$. After entering the filename into the GhostShare client, the same hash function is applied to each of its tokens, producing the same filename token 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 token 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 token hashes → content hashes* mapping. In this case, for each filename token hash X_i, from the first mapping the node closest to X_i finds that some file with content hash Y has a filename token hashing 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 hashing to X_1 and X_2. By the birthday paradox, these are most likely the 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.

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 token hashes generated from X' are a subset of $X_1, X_2, \ldots, X_n$. The proposed mechanism, moreover, distinguishes between files shared on the network having the same filename, but different content. As an example, say that a document is shared in plaintext and postscript format. Both have the same filename X, and therefore produce the same filename token hashes $X_1, X_2, \ldots, X_n$. Now say the plaintext version has a content hash of Y, while the postscript version has a content hash of Y'. Therefore for each filename token hash X_i, the node numerically closest to X_i will maintain the mappings $X_i \rightarrow \{Y, Y'\}$ and $\{Y \rightarrow \{Z_1, Z_2, \ldots, Z_l\}, Y' \rightarrow \{Z'_1, Z'_2, \ldots, Z'_m\}\}$, where $Z_1, Z_2, \ldots, Z_l$ are the nodes sharing the plaintext version, and $Z'_1, Z'_2, \ldots, Z'_m$ are the nodes sharing the postscript version. Using the method described above, a node searching for X will receive the results $\{Y \rightarrow \{Z_1, Z_2, \ldots, Z_l\}, Y' \rightarrow \{Z'_1, Z'_2, \ldots, Z'_m\}\}$—allowing the searching node to differentiate between the two different versions of the file, and which nodes share which version.

2) Index Reliability: Peer to Peer networks are characterized by a very dynamic membership with frequent node arrival and departure[12]. In the event a node disconnects or fails unexpectedly, all *filename token 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 token 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 token hash X_i. If not, that node must simply surrender the tuple $< X_i, Y, Z >$ 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.

B. Distributed Index Load Balancing

As described in section II-A each node participating in the lookup service is the responsible index for the token hashes closest to it; suppose a given node is closest to a token 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 token 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 token hashes [13]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 [14].

Using the assumption that Pastry node IDs are uniformly

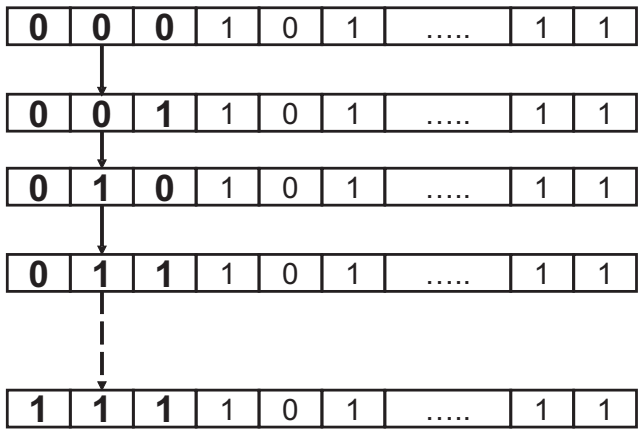

Fig. 3. **Figure shows how to permute the first d bits to distribute the load among the 2^d replicas.**

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 token 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 token 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. 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 token hash X_i the requesting node will generate the proper replica permutation substituting the first d bits of the token hash with the result of its current clock in $/mu$sec modulo (2^d). The requested will contact the resulting replica to retrieve all content hashes and node IDs associated with X_i.

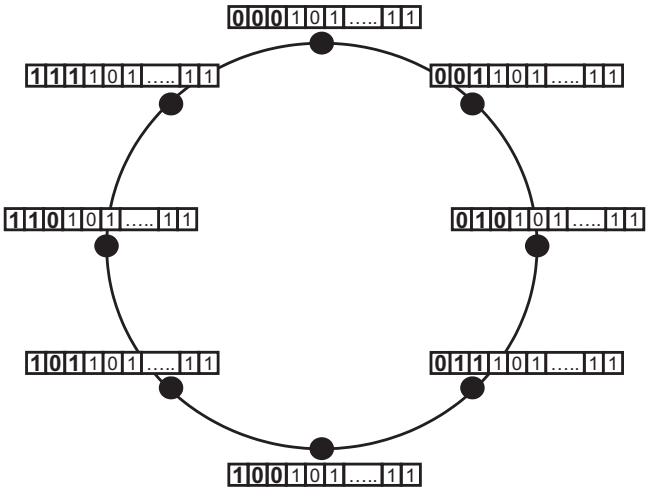

Fig. 4. **Figure shows that replicas are evenly distributed in the nodeId space.**

C. Disjoint Multiple Path

In an overlay network where nodes can join, leave, and fail unpredictably, leading to routing instability, reliability is a key concern. Moreover, if a bottleneck exists along the path between two nodes X and Y, finding an alternate route can drastically improve performance. This is true especially for applications that require large throughput, such as file sharing and global storage [15]. To approach these problems, we extend the Pastry routing protocol to discover several paths between two distinct nodes. To do this, nodes along a path between X and Y maintain state while the path is used, only erasing the state when instructed to destroy the path by X or Y. From the sender's point of view, data is sent in a round robin fashion along each path, fragmenting the data over all available paths. The receiver can then reassemble the file in a similar fashion [16]. In this section we detail the algorithm for establishing disjoint paths and explain how they can be used to improve throughput. In Pastry, the shortest path between a source and destination node might not necessarily be the fastest, because the number of hops on a path is unrelated to the speed of the connection at each hop. Therefore, it is reasonable to assume that there might exist a longer path between the source and destination (i.e., traversing more hops) that is faster, because the connection speed of each interior hop on this new path might be faster. In this section, we present a way to establish multiple paths between a source and destination, thereby bypassing any bandwidth bottleneck that might occur on the shortest path traditionally offered by Pastry. In addition, these multiple paths will be disjoint over the interior nodes, meaning for any two paths, the only shared nodes between them are the source and the destination nodes. Given a source node W and a destination node Z, one approach to creating these disjoint paths is as follows: Create the first path based on the traditional routing techniques of Pastry. This assures you at least one path of length $log_{2^b}\ N$. While establishing this path, each interior node on it sends the hash of its node ID to W. Node W collects these node IDs in a set called the *exclusion set*. Now let X be some node responsible for choosing the next hop of the second path leading to Z, and suppose X has the exclusion set. When X selects Y_i as the next potential hop on the path to Z, node X can immediately check whether a path between W and Z already passes through Y_i by examining whether the hash of Y_i's node ID exists in the exclusion set. If the hash of Y_i exists in the exclusion set, X knows that it should move on and find the next closest node to Z in its routing table. If the hash is not in the exclusion set, then node X can freely pick Y_i as the next hop in the second path. Node X then passes on a copy of the exclusion set to node Y_i, so that Y_i can use it to make the same decisions. After this path is completed between W and Z, each interior node along the second path sends the hash of its node ID to W so that it can be added to the exclusion set. This larger exclusion set is used when forming the third path, guaranteeing that it does not share interior nodes with the first or second path, and so on. Note that we choose to build the exclusion set from *hashes of node IDs as opposed to node IDs themselves*. This is motivated by the fact that any malicious node chosen as a next hop, after receiving the exclusion set, cannot easily determine what nodes comprise the other paths, or preceding hops along the current path.

III. Anonymity

GhostShare provides information exchanging capabilities between its members in an anonymous fashion. Given a node who shares some information (Source), and a node who wants it (Destination) an anonymous transfer would be one where Source does not know Destination's IP and Destination does not know Source's IP. This protects the Source from disclosing what information it shares on the network and Destination's identity remains hidden [17]. Related work on achieving peer-to-peer anonymity is detailed in section V. GhostShare proposes a probabilistic approach to anonymity that integrates in the same system anonymous lookup and message exchange.

A. Representative Concept

In order to keep source and destination undisclosed, in GhostShare, each node performs the Searching, Downloading and Publishing through a *Representative* probabilistically chosen among the network members.
The Representative will pretend to be information owner, the requester, or the destination acting instead on behalf of somebody else. It is important to have a secure Representative selection algorithm.

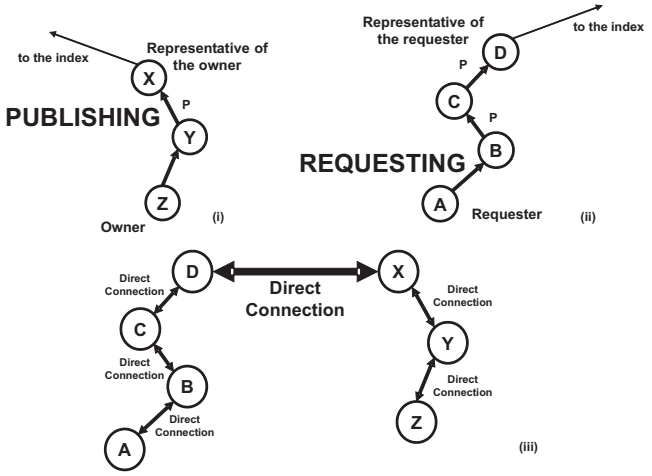

Fig. 5. **Figure shows anonymity mechanisms**

1) Representative Selection: Let us consider a node Z that needs to publish anonymously a set of files. It will first choose a random number RN consisting of K bits (K is assumed to be large) to be used as a session identifier. Next, it will send a message containing RN and the file hashes $\{h(F_1) \cdots h(F_n)\}$ to a randomly selected node in its routing table. It will ask this node to pose as the owner of its files, or as we refer to it, the Representative. The full algorithm for finding a representative is as follows:

- Z randomly selects a node Y from its $RoutingTable|LeafSet|NeighborhoodSet$ and sends to it the message described above.

- Y in turn with some probability $1-p$ will forward that message to a randomly chosen node from its $RoutingTable|LeafSet|NeighborhoodSet$, much like Z did, and repeat this step. With probability p it will advertise itself as the owner of the files belonging to Z.

The process of forwarding $Z's$ representative request will continue until somebody (X in figure 5(i)) accepts the role of $Z's$ representative and advertises itself as the owner of the files. In our case Z forwarded the representative request to Y, which forwarded it to X who chose to be $Z's$ representative. Thus we have the chain $Z \rightarrow W \rightarrow Y \rightarrow X$. When the representative request message passes through Y and reaches X, all of these nodes will extract RN from the message and associate it with the IPs of the nodes next to them on the chain. For example, Y will associate RN with Z as the node whom it received the message from and the node X as the node whom it sent the message to $(Y : RN_Z, X)$. X will have only one entry, namely RN_Y, since it is the final destination. Since X accepted to be the representative, it must advertise $Z's$ files in the network. Content publishing and searching are performed by the representative using the lookup service described in section II-A This approach allows for full anonymity of Z. Since Y and X might choose to be the owner of Z's files with some probability p, a given node can not determine without a reasonable doubt whether Z is the true owner of a given file F, or just a forwarding agent. Say Y wanted to know if Z shared F: it could not determine that, because Z can be just forwarding F (since Z could have chosen with probability $1-p$ to forward the representative request message).

B. Anonymous Download

Let some node A have some file F, which is advertised by D. A node on the network, say Z, wants to download F. After it has searched for it, it finds that D "has" F. It then uses the same technique A used to advertise its files in order to request and download F anonymously. Z will chose some node Y at random from its $RoutingTable|LeafSet|NeighborhoodSet$ and send it a message containing F and RN (discussed above), asking it to request F. Y with some probability p will then chose to forward the message to another randomly selected node, or it will complete the request with probability $1-p$. Say that Y forwards this message to X, which in turn decides with probability $1-p$ to request F from D. After X has requested F it soon starts receiving F from D. It then forwards this to Y, which in turn forwards it to Z (see Fig. 5(iii)). Due to the fact that each node decides with some probability whether to request a file on the behalf of somebody else or not, we can not know for sure whether X is the true node that requested the file, or just a forwarding agent.
Theoretically speaking, the forwarding chain could be infinite. However, as shown, in section III-E the average size of chain is expected to stay a reasonable value.

C. Handling Node Departures and Failures

. Let assume the following scenario: Node Z shares some file through some other node Z_{rep} (Z_{rep} is thus a representative). Let W and Y be nodes on the chain between Z and Z_{rep}. Thus we have the chain Z, W, Y, Z_{rep}. Now let us examine the scenario when Z_{rep} fails or leaves the network cleanly.

1) Z_{rep} Departs Cleanly.: When Z_{rep} decides to leave the network, it just sends a message to Y informing that it is leaving. Y will forward this message to W, which will forward it to Z. Once Z receives this message it will ask somebody else to take ownership of its files using the already described procedure in section 4.1. Effectively, the old path is torn down, and a new path is established to replace it.

2) Z_{rep} or Node On the Path Fails.: If Z_{rep} fails the node next to him that is on the path to Z, or namely Y will detect this failure and forward the fail message to W, W in turn will forward it to Z. When Z receives the fail message it will ask somebody else to take ownership of its files using the already described procedure in section 4.1. If a node in the middle of the chain fails, such as Y, then the nodes next to it on the same chain, here being W and Z_{rep}, will detect this failure. W will forward the fail message to Y, which will forward it to Z which in turn is going to advertise its files again. Z_{rep} can detect Y's failure and can conclude that the path between it and Z has been broken. It will thus remove the RN_Y entry in its table and simply ignore requests for files that it advertised on behalf of Z.

D. Reliability and Performance Discussion

A possible problem is the fact that, if a node on the download path fails then the download path has been broken. A possible solution to that is for Z to advertise its files more than once through multiple paths simultaneously. Then Z will have redundant paths through which it can send the file to its final destination. However, this approach presents a couple of drawbacks:

- If any of Z's forwarders fail, Z will have to re-advertise its files, which is a costly operation.
- We cannot preserve anonymity as well, because the first hops along two paths Z establishes may be able to triangulate Z as the true owner.

Furthermore, another problem is the fact that all the requests for a file F that Z shares pass through the chain between Z and Z_{rep}. Now if a certain node receives many requests the path between Z and Z_{rep} will become quite loaded and thus the nodes along this path will do a lot of forwarding. A way to distribute this load is the following: Z divides the hashes of its files into k disjoint even sets $S_1, S_2, \ldots, S_k$. It also chooses k different nodes $N_1, N_2, \ldots, N_k$ from its $RoutingTable|LeafSet|NeighborhoodSet$. Then, for each i in $1 \ldots k$, Z will advertise the set S_i through N_i. Thus Z will create k advertisement paths and have k representatives, $Z_{rep(1)} \ldots Z_{rep(k)}$. This method has the following advantageous properties:

- If $Z_{rep(i)}$ dies Z can still be reached for the remaining = N N/k files that it shares.
- Downloads for different files will travel through different paths and thus not strain only one chain. Now, for example, Z can download a file using say $Z_{rep(i)}$ and at the same time send a file using $Z_{rep(j)}, i <> j$
- Note that if the failure rate for all nodes on the network is the same then the bandwidth consumed by sending the whole index S = bandwidth for sending k times S/k. Thus by doing this we evenly distribute data flow to and from Z to k different paths.

E. Analysis of the Anonymity Mechanism

In this section we model certain key characteristics that determine the performance of publish-lookup scheme proposed earlier.

One key performance characteristic is the average path length that a file traverses before being published. In other words, the distance between the *owner* and *publisher* of the file. We compute this *Expected Path length* and derive some bounds on the path length in terms of Pastry node hops. Path length, denoted by random variable L henceforth, can be computed by observing that a file can be published by either a node's one-hop neighbor picked at random from the $logN$-sized routing table. Hence, the expected value of L the path length in terms on Pastry node-hops is,

$$\begin{aligned} E[L] &= 1.p + 2(1-p).p + 3(1-p)^2.p + \cdots \\ &= p(1 + 2(1-p) + 3(1-p)^2 + \cdots) \\ &= p(1/p^2) \\ &= 1/p \end{aligned} \tag{1}$$

Here p is the probability that a Pastry node decides to publish the file on the behalf of the owner. This decision to publish is independent of the number of times the publish request has been relayed. The equation(1) above gives us an estimate of the number of hops that the file will traverse before being published. We note that this hop-length is independent of the routing table size and peers on the network. We want to study the deviations from the expectation and magnitude of the deviations. We use the Markov Inequality to derive some loose bounds,

$$\begin{aligned} P[L \geq k] &\leq \frac{E[L]}{k} \\ &= \frac{1}{pk} \end{aligned} \tag{2}$$

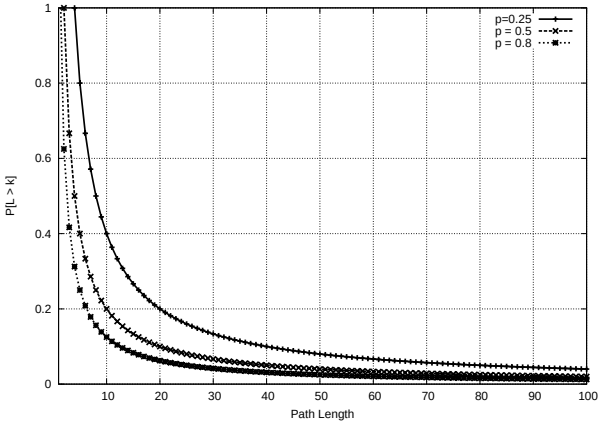

Fig. 6. **Probability of Path Length exceeding k for different values of k. Different plots are for different forwarding probabilities. Refer to the Equation(2).**

The equation(2) gives us a rough estimate of the probability that the path length is greater than some arbitrary value. For

example, consider $k = 10$ and $p = 0.4$: then the probability that the random variable L(the path length) takes on a value ≥ 10 is less than 0.25. Note, here the expected path length, is $1/p = 2.5$.

Another more interesting bound that we can derive using the Chebyshev's Inequality is as follows:

$$P[\ |L - E[L]|\ \geq t\sigma] \leq \frac{1}{t^2} \tag{3}$$

In this case, the variance σ^2 is $(1-p)/p^2$.

IV. Simulation Results

We evaluate the performance of the multiple path routing mechanism and the load balancing achieved by GhostShare using simulations.

A. Multiple Path

We compare the performance that we get using the disjoint multiple path mechanism versus the Pastry routing mechanism while including the need for Pastry to repair its routing table. To obtain a reasonable value of reference, we compare the average path length of traditional Pastry with the length of the minimum path chosen by multipath in the presence of varying rates of the node departures. The path that Pastry calculates is optimal by definition in the presence of no failures, but after the departure of some nodes, after repair, it can became longer than the least remaining multipath (This is because Pastry routing table repairs may be sub-optimal). We ran the simulation using a network of 55000 nodes, choosing random nodes for the exchange of messages. We show that the multiple path mechanism can indeed find paths that have fewer hops than the paths obtained with Pastry schemes. We have considered Pastry (1 path and 1 entry), Pastry with repairing the routing table, Pastry with 3 elements stored in each cell of the routing table (1 path and 3 entries), and multipath Pastry (3 paths and 3 entries). In figure 7 and 8 we observe multipath Pastry provides consistently with a shorter route, in both the presence of 10% and 20% node failure rates. We see in the figures 11 and 12 the probability of obtaining a certain path length in the routing of a message in these systems. We see that the length of the least path obtained with the multipath mechanism is most cases about 3 hops, while with Pastry we obtain paths that reach 6 hops. While increasing the percentage of messages exchanged, the probability distribution of multipath scheme still stays smaller than the traditional Pastry routing schemes as shown in figure 10.

1) Overhead Analysis: We try to evaluate the overhead imposed by the multipath load balancing scheme. We consider the message exchanges. When we build the first path, each node along the route has to send back its node ID to the first node. The first node adds this value to the exclusion set and uses it for the creation of disjoint paths. The same thing happens for the second path, so we have an upper bound of $log_{2^b}\ N$ for the first path and $log_{2^b}\ N$+1 for the second route. The total cost of this operation is about $2 * log_{2^b}\ N + 1$. In the Pastry design we have routing table with $log_{2^b}\ N$ rows and 2^b columns, but only $2^b - 1$ of this cells can be potentially full.

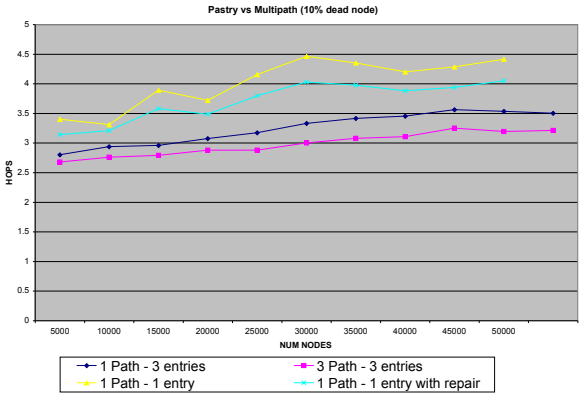

Fig. 7. **Figure shows the path-length using various techniques, vanilla pastry routing, pastry with repair, pastry with 3 elements stored in each cell, and finally multipath. These results are for a node death rate of 10%.**

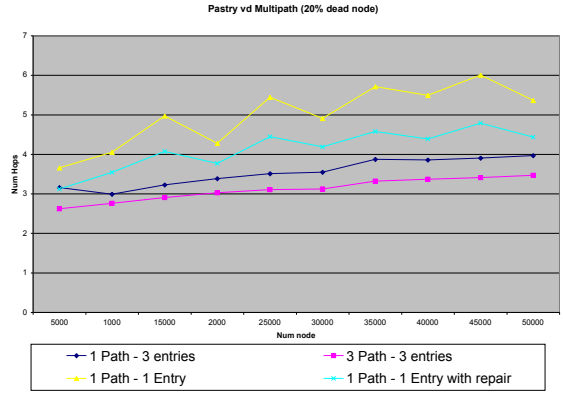

Fig. 8. **Figure shows the path-length using various techniques, vanilla pastry routing, pastry with repair, pastry with 3 elements stored in each cell, and finally multipath. These results are for a node death rate of 20%.**

The total cost of this table in Pastry is $log_{2^b}\ N * 2^b - 1 * K$, where K is the number of bits for storing a NodeId. Using disjoint multipath mechanism we store 3 entries in each cell, so the total cost becomes: $((log_{2^b}\ N) * 2^b - 1) * K) * 3$.

B. Load Balancing

Suppose we have n replicas of the index for a particular keyword hash on the network. Every time a user searches for that keyword, it contacts one of the n nodes storing the index. The load is pefectly balanced if, given k users searching for that token, each replica is contacted by k/n peers. This means we need a mechanism that uniformly distributes the queries to the replicas of the index. When a client asks for a particular token hash, we record the current time in millisecond since the epoch (call it m) and then we calculate the value $m\ mod\ n$ (See Figure 13. We saw that this function

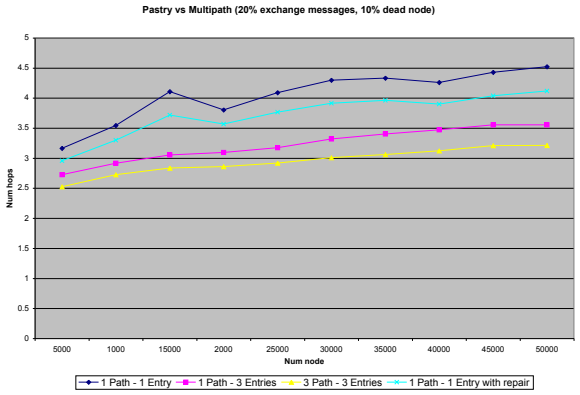

Fig. 9. **With increase in percentage message exchanged to 20% compared to 10% in the previous figures, multipath still performs better.**

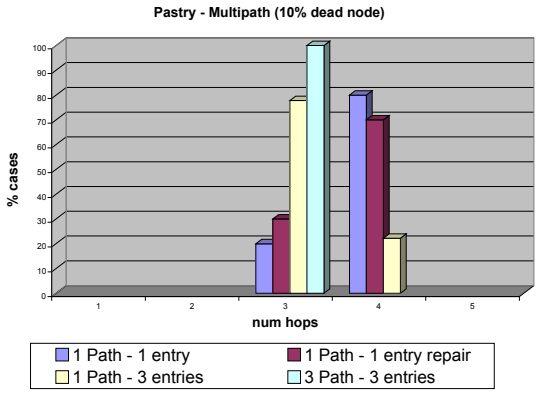

Fig. 11. **Probability distribution of the path length in the various schemes, for 10% node failures.**

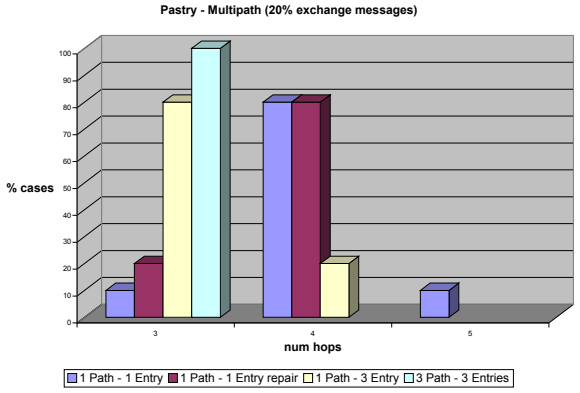

Fig. 10. **Probability distribution of the path length in the various schemes, for 20% message exchanged**

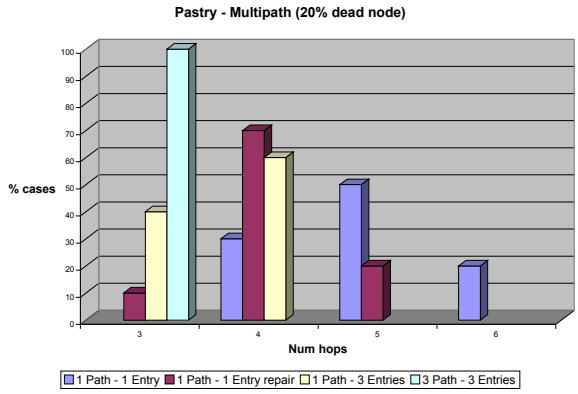

Fig. 12. **Probability distribution of the path length in the various schemes, for 20% node failures.**

provides a highly uniform distribution of the searching clients to the nodes sharing the replicas of the index. To prove this, we modified our Pastry simulator so that 50% of the node simultaneously requested the same index. We than permuted the most significant d bits using the algorithm above, and recorded how many times each replica was contacted. We repeated the test increasing n from 2 to 64 and increasing the network size up to 100,000 nodes. In Figure 14 we show that in any case, the coefficient of variation is less than 1%, which proves this method's effectiveness for load balancing.

V. Related Work

A. Anonymity

Anonymity is one of the primary concerns of the so-called second generation P2P systems. In this section we survey related work on anonymity in P2P systems and earlier work on anonymity implementations in web transactions.

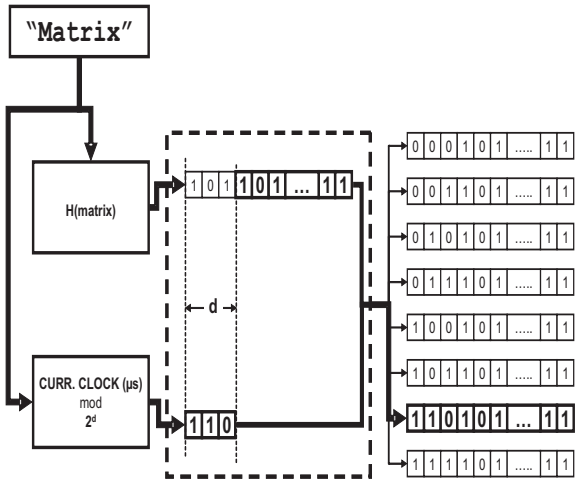

Fig. 13. **The figure shows how our simulator uses the clock to choose a replica that has to comply with the request.**

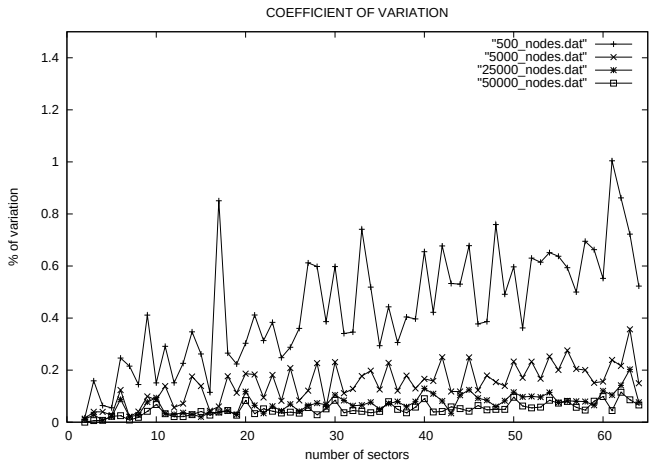

Fig. 14. **The percentage is given by ((σ/μ) * 100) and we can easily see from the figure that it is very very good (always under 1%). Moreover we show that we can scale easily maintaining good results.**

1) Source Rewriting Systems: : Source Rewriting systems try to hide the correspondence between sender and receiver, sub-dividing messages in layers of public-key cryptography and relaying them through a path composed of mix nodes. They remove the timing correlation between arriving packets and outgoing packets. Each mix node in turn decrypts, delays and re-orders messages before forwarding them. They work better when arranged in series and need a constant amount of traffic to maintain anonymity while avoiding long packet delays.

Tarzan [18] uses a decentralized, multi-hop peer-to-peer approach to construct such mixes, called tunnels. All the participants generate traffic and relay traffic for others (each node can act as a server, but there is a limit in the use of IP addresses so a node can't control a large part of the network). Tarzan is not immune to statistical analysis and provides large tunnel intra-hop latencies that have a noticeable impact on end-to-end performance. Other protocols, like Onion Routing [19] and Crowds [17], create random paths for sending messages. Onion Routing creates a circuit in which each node knows only its predecessor and successor, but no other nodes in the circuit. Each node in the path rewrites the source field of the packet with its own id to obscure who is the sender of a message. Additionally packets are encrypted and reordered at each node. Each node reorders the packets, unwraps a layer of encryption with the session key shared with the predecessor node, encrypts it again using the key shared with the successor and forwards them.

Source rewriting is also used in P2P content distribution networks such as Freenet [20] to obfuscate the sender of request messages (and also the data storage). In Freenet, files are stored in each node along the path and replaced using a Least Recently Used replacement scheme (so unpopular files will disappear). Freenet does not allow a node to choose its nodeId. The use of hashes as keys provides a measure of obscurity against casual eavesdropping but is vulnerable to a dictionary attack. There is no anonymity between the user and the first node contacted along a path. Stronger sender anonymity can be achieved by adding mix-style pre-routing of messages. These systems are efficient and scale well, but they do not provide strong anonymity guarantees. Further, source rewriting has high latencies.

2) Broadcast Protocols: : Broadcast protocols (such as P5 [21] and Herbivore [22]) provide *sender* and *receiver* anonymity by transmitting encrypted packets at a constant rate to all participants. When a node has no data packets to send, it sends noise that is propagated through the network in the same manner as valid data packets. This approach provides *strong* anonymity. P5 scales by partitioning the network into anonymizing broadcast groups. P5 achieves anonymity (based on public-key cryptography) and scalability but at the cost of efficiency caused by noise. Herbivore uses a mechanism for strongly anonymous communication (each node tosses a coin in secret and reports the result to the destination node). The destination deciphers the message by taking the XOR of all values it receives. For topology, the network is divided into small groups, called cliques, sender and destination are hiding behind a group (a group for each node is chosen randomly and there is a time limit to the insertion rate of a node in the network). A node's anonymity depends on the number of honest nodes in a clique.

3) Anonymity in Web transactions: : Crowds [17] consist of a dynamic collection of users, called a crowd. These users initiate requests to various web servers (and receive replies from them), and thus the users are the "senders" and the servers are the "receivers". A second approach to achieving anonymous web transactions is to use a mix. As discussed earlier, a mix is actually an enhanced proxy that, in addition to hiding the sender from the receiver, also takes measures to provide sender and receiver unlinkability against a global eavesdropper. It does so by collecting messages of equal length from senders, cryptographically altering them, and forwarding the messages to their recipients in a different order. These techniques make it difficult for an eavesdropper to determine which output messages correspond to which input messages. A natural extension is to interpose a sequence of mixes between the sender and receiver. A sequence of mixes can tolerate colluding mixes, as any single correctly-behaving mix server in the sequence prevents an eavesdropper from linking the sender and receiver. Mixes have been implemented to support many types of communication, for example electronic mail and general synchronous communication. The properties offered by Crowds are different from those offered by mixes. As described above, Crowds provide probable innocence against collaborating crowd members.

B. Load Balancing

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

Effective load balancing [23]: The system 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 clusters of peers is used to adjust the system to the changes of the 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 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 terms of 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 [24]: The approach taken 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 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 preserve key ordering, which is important for range queries, while retaining the search efficiency $O(log\ \pi)$ (in terms of the number of key space partition $abs\pi$, irrespective of key distribution. 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 [25]: This algorithm 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 $loglogn/logd + O(1)$.

Virtual servers [26]: 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 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 be 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.

VI. 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). NetFlix [27]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. This paper proposes an peer-to-peer system suitable for cooperative delivery in home video entertainment applications. In particular GhostShare provides: (i) a *reliable name lookup service* to support advanced content search that includes a built-in index load-balancing, (ii) a *multiple disjoint path routing* to improve performance and reliability, (iii) *anonymous downloading, searching and publishing capabilities* to protect user privacy and (iv) a download mechanism with a built-in content replication system to strengthen content availability. The proposed download mechanism, described in [28], [29], [30], [31], [32], [33], [34], [35], [36], 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 on-line and serve as relays during periods of inactivity can also be built-in. In the ideal scenario, the customer's fee grants him or her an *access key* that allows the costumer 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.

ACKNOWLEDGMENT

This work is part of the project *GhostShare* aiming to build a pastry based Anonymous Secure Distributed Peer-to-Peer System for Ally-Enemy Network Environments.
The authors would like to thank for their humongous work, comments, suggestions, and key experimental support the following University of Bologna and University of California Los Angeles undergraduate students: **Dayana Cucchi, Gionata Ercolani, Matteo Morigi and Daniele Pieri**, *"Corso di Laurea in Scienze dell'Informazione di Cesena" - Universita' di Bologna* ; **Ibrahim Elbouchikhi, Michael G. Parker, Amir N. Teherani, Georgui Velev, Alex Weinstein**, *"Department of Computer Science" University of California Los Angeles*

REFERENCES

[1] http://www.napster.com/.
[2] http://www.kazaa.com/.
[3] http://bitconjurer.org/BitTorrent/.
[4] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
[5] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. The quest for balancing peer load in structured peer-to-peer systems. 2003.
[6] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, November 2001.
[7] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.
[8] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. One ring to rule them all: Service discover and binding in structured peer-to-peer overlay networks. In *SIGOPS European Workshop*, September 2002.
[9] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in distributed hash tables. In Ozalp Babaoglu, Ken Birman, and Keith Marzullo, editors, *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 52–55, June 2002.
[10] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable p2p lookup service for interent applications. In *In Proc. of SIGCOMM'01*, August 2001.
[11] Helena Handschuh, Lars R. Knudsen, and Matthew J. Robshaw. Analysis of sha-1 in encryption mode. In *CT-RSA 2001, vol 2020 of Lecture Notes in Computer Science*, pages 70–83, 2001.
[12] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02)*, January 2002.
[13] Donald E. Eastlake and Paul E. Jones. Us secure hash algorithm 1 (sha1). September 2001.
[14] Philips J. Erdelsky. The birthday paradox. July 2001.
[15] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001.
[16] D. Cucchi, G. Ercolani, M. Morigi, M. G. Parker, G. Pau, D. Pieri, P. Salomoni, A. N. Teherani, and G. Velev. Ghostshare: You name it, you get it! providing peer-to-peer services in an ally-enemy environment. In *UC Los Angeles Technical ReportUCLA/CSD-TR030050*, November 2003.
[17] Michael Reiter and Aviel Rubin. Crowds: Anonymity for web transactions. June 1998.
[18] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. 2002.
[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Usenix Security*, 2004.
[20] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distribuited anonymous information storage and retrieval system. In *In Proc. of Workshop on Design Issues in Anonimity and Unobservability*, pages 311–320, July 2000.
[21] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A protocol for scalable anonymous communication.
[22] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. In *Cornell University Computing and Information Science Technical Report*, February 2003.
[23] Anirban Mondal, Kazuo Goda, and Masaru Kitsuregawa. Effective load-balancing of peer-to-peer systems. 2003.
[24] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Multifaceted simultaneous load balancing in dht-based p2p systems. 2004.
[25] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Geometric generalizations of the power of two choices. 2004.
[26] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *IEEE INFOCOM*, 2004.
[27] http://www.theregister.co.uk/2004/01/27/netflix_the_fly/.
[28] http://www.movielink.com/.
[29] http://www.walmart.com/index.gsp?sourceid=2405972536388928456 0dest=40.
[30] http://www.cinemanow.com/home.aspx.
[31] http://www.dvdemand.com.
[32] http://web.mit.edu/kerberos/www/.
[33] http://video.csupomona.edu/streaming/.
[34] http://www.streamingmedia.com.
[35] http://www.digitalfountain.com/solutions/apps/perfectVideoWiredOrWireless.cfm.
[36] http://www.apple.com/itunes/.