Hierarchical Routing with Soft-State Replicas in TerraDir

Bujor Silaghi, Vijay Gopalakrishnan, Bobby Bhattacharjee, and Pete Keleher
Department of Computer Science, University of Maryland, College Park
{bujor, gvijay, bobby, keleher}@cs.umd.edu

Abstract

Recent work on peer-to-peer systems has demonstrated the ability to deliver low latencies and good load balance when demand for data is relatively uniform. We describe an adaptive replication protocol that delivers low latencies, good load balance even when demand is heavily skewed. The protocol can withstand arbitrary and instantaneous changes in demand distribution. Our approach also addresses classical concerns related to topological constraints of asymmetrical namespaces, such as hierarchical bottlenecks in the context of hierarchical namespaces. The protocol replicates routing state in an ad-hoc manner based on profiled information, is lightweight, scalable, and requires no replica consistency guarantees.

1 Introduction

Peer-to-peer (P2P) systems perform any number of different functions, but their most fundamental task is that of locating data. Recent work [11, 17, 12, 18] has shown the ability to deliver low latencies and good load balance when demand for most items is relatively balanced. The distribution of demand for real data is often skewed, and sometimes time-varying, leading to poor balancing and dropped messages. The situation is even worse with hierarchical namespaces such as TerraDir [15], where the system topology is inherently asymmetrical, resulting in uneven load across servers even with uniformly distributed demand. This paper describes and evaluates a lightweight and adaptive replication protocol that is efficient at redistributing data load in such circumstances, and that can improve query latency and reliability as well.

In this study we will focus on routing load and discriminate between the lookup of an object and its actual retrieval (few of the objects looked up or searched for are effectively retrieved if we are to consider presently deployed P2P applications). So far, load has usually been addressed by caching and replicating data in an end-to-end manner by client applications of P2P lookup services (e.g. CFS [6], PAST [7]). The resulting protocol layering incurs the usual inefficiencies, causes functionality to be duplicated at the application level, and overall is quite heavyweight.

To our knowledge, none of the previous work has addressed routing load as a distinct phenomenon in the context of P2P systems. We believe that lightweight and efficient solutions can be developed more readily if one makes such a distinction. Note that replicating data and replicating routing state for lookup purposes in the overlay P2P network are orthogonal. The service provided by a lookup procedure resolves an object name to one or multiple locations, without having to visit those locations or retrieve the object content.

We evaluate our approach with respect to the following goals in the context of TerraDir [15], a lightweight, hierarchical P2P lookup service: local information and scalability, adaptation versus stabilization, fairness, fault tolerance, and exclusive use of profiled a posteriori information. The main contribution of this study is showing that these goals can be met for hierarchical structures by employing a lightweight adaptive replication model, and that the implementation of such a model in real systems is feasible.

The rest of the paper is structured as follows. In Section 2 we present a hierarchical routing protocol based upon our replication model. The replication protocol is described in Section 3. Section 4 experimentally evaluates the proposed model. We address the context of our approach and related work in the area in Section 5. Section 6 concludes the paper.

2 Hierarchical routing in TerraDir

We describe the assumed data model, the semantics of query processing, a routing procedure on hierarchical namespaces, and finally augment the routing model with replication and caching.

2.1 Data model and query semantics

A *node* is a fully qualified hierarchical name, much like file names in Unix file-systems or host names in the DNS space. Each node has a set of neighboring nodes that includes the parent and children of the node in the namespace. Here we assume that the structure of the namespace is that of a tree with a special node, the root node, as the root of the tree. TerraDir allows arbitrary graph-rooted topologies to be specified.

Nodes export two types of optional application-supplied information: data and meta-data. Node data is the actual contents of a node, while meta-data consists of node annotations most commonly found in the form of attributes (name-value pairs). For a file-system implemented on top of TerraDir or a file-sharing utility, there is a 1-to-1 correspondence between files and nodes. The node's data in this case is the file, and the meta-data are file attributes and searchable keywords annotated to it.

Every node is owned by exactly one server (peer) known as the *owner*. The owner of a node keeps the node's data and meta-data as well as additional state needed by the TerraDir protocol in making routing decisions. Note that the owner of a node is also the server that exports its data. Hash-based peer-to-peer systems virtualize the object namespace and the server that exports the object data will most likely be different than the server that keeps the object's hash key. For such systems the location of data and the location of pointers to data are different.

A lookup query returns the node's name, its meta-data, and mapping information for the requested node. The *mapping* is a set of servers that host the node's data. Given the result of a query, the client application can further request the node's data from one of the servers in the map. Note that getting node data is a two-step process: a node lookup, followed by the actual data retrieval. Complex search queries are decomposed hierarchically into individual lookup queries, the appropriate nodes are resolved, and then the results are aggregated and sent back to the requester. Subsequently, the query initiator may ask for the data of some of the nodes in the result.

Deployed P2P systems make a similar distinction between searching for data and retrieval of the data. This study focuses on lookups and how queries are routed in the network using a lightweight replication mechanism.

2.2 Hierarchical routing

A query is initiated at one of the servers and then routed through one or more servers to its destination.

2.2.1 Routing procedure

The routing algorithm proceeds in a straightforward manner by forwarding queries up and down the hierarchy. Usually, a query for node d originating at the owner of node n will be routed "up" until reaching the first common ancestor of s and d, and then "down" to node d. For instance, if

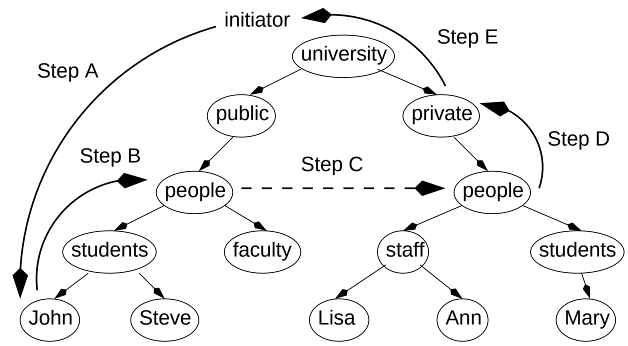

Figure 1. Route for query */university/private*.

$s = /a/t/f/u/$ and $d = /a/t/w$ the resulting path will be $<s, /a/t/f, /a/t, d, s>$. In Fig. 1 we illustrate a more involved routing example using replication and caching (see the following sections). Step D follows a child-parent link induced by the hierarchical topology.

2.2.2 Routing state

To facilitate routing, a server maintains for each owned node its context in the namespace. The node context consists of neighboring nodes and guarantees routing with *incremental progress* towards the destination. With each routing step the query gets closer to the destination by at least one unit in the namespace distance metric.

A server maintains mappings for the neighboring nodes of every node that it owns. The mapping is the association of a node's name and a set of servers hosting the node. For tree namespaces with N nodes on S participating servers we can only bound the number of neighboring maps (links) maintained per node by $O(N)$. However, the cumulative number of links for all nodes is exactly $2(N-1)$, which yields a comfortable mean of 2 links per node. In Fig. 1 neighboring links are shown as straight arrows between parent and children nodes.

2.3 Replication

Straightforward routing on a hierarchical namespace suffers from well-known bottlenecks. Even assuming uniformly distributed queries (both source-wise and destination-wise), servers hosting nodes at the top of the namespace will incur exponentially disproportionate more load than servers hosting leaf nodes. Furthermore, variations in user input may cause other parts of the namespace to be similarly afflicted (e.g. hot-spots). Finally, we do require some form of routing state redundancy to increase the resiliency of the routing procedure and routing state availability. While hierarchical bottlenecks can be addressed by static replication mechanisms [15], the last two arguments call for an adaptive scheme.

We therefore dynamically replicate heavily loaded nodes. We attempt to minimize the amount of state replicated per node, subject to the following constraints:

1. Lookup queries can be resolved by reaching a replica of the node. Such state includes node meta-data, and some mapping for the node. The mapping can be used by the query initiator if it further wishes to retrieve the node data.

2. Routing through a replica needs to be functionally equivalent to routing through the original node. Therefore, a replica will also keep the context of the original node: mapping information for each of its neighboring nodes.

The term *host* will denote hereafter the owner, or one of the replicating servers (in the sense presented here) of a node. In Fig. 1, the owner of */university/public/people* hosts a replica of */university/private/people*. Step C is thus abstract and does not incur query forwardings or network hops. Through replication not only is the owner of */university/public/people* present in other parts of the namespace (*/university/private/people*), but it can also further forward and resolve requests that come in on behalf of the replicated node. This serves a dual purpose: improved query latencies (additional shortcuts), and a mechanism to balance routing load by shedding some of it from the owner of */university/private/people*.

Note that we only replicate routing state and meta-data. Inconsistent routing state (nodes leaving or joining the system) will manifest in less precise forwarding steps. A query could reach a server on behalf of some node even though the server does not host the node any longer. In such cases incremental progress cannot be guaranteed for the current forwarding step. We assume that node meta-data is invariant or else that there are no consistency/freshness requirements for its update/use. Only the owner server of a node is allowed to modify meta-data, and replicas will keep the newest version that they have encountered.

2.4 Caching

A cache entry for a node consists solely of some mapping for that node. A hit in the cache cannot by itself bring query resolution. The query still needs to be forwarded to one of the resolved node's hosting servers (found in the node map). Caches provide only indirect routing functionality: they lack routing context, and act as mere pointers in the namespace. Step B of Fig. 1 corresponds to the shortcut taken with a cache entry.

Caches are ad-hoc state in the sense that there is no correlation between cache contents at different servers. Replacement, eviction and aging is performed locally. Cache entries are replaced using an LRU policy with an entry being touched whenever used in routing. Our caches differ from straightforward caches in that the path "so far" is cached at every step along the query path (*path propagation*); culminating in the entire path being cached at the source when the query completes.

Caches increase the routing state maintained per server to $O(\log S)$ (S is the number of servers) and substantially improve query latency. Caches are able to exploit both temporal and spatial locality in the query stream. Even in the absence of locality, the routing procedure will benefit from caching by taking shortcuts over potentially large portions of the namespace. Routing resiliency is also augmented by the ability to jump over namespace partitions induced by network failures. Path propagation not only brings nodes far apart into the cache (the source caches the destination, and vice-versa), but also nodes from different levels of the namespace tree, and nearby nodes. This mixture of close and far nodes performs significantly better than caching the query endpoints.

Table 1. Server-node relationships.

Node\State	Name	Map	Data	Meta	Context
Owned	√	√	√	√	√
Replicated	√	√		√	√
Neighboring	√	√			
Cached	√	√			

We summarize the various relationships between servers and nodes in Table 1, along with the type of state maintained. Note that cached nodes and neighboring nodes are similar except that cached nodes can be arbitrarily replaced and are not imposed by topological constraints of the namespace. The routing context (last column) refers to maintaining links to neighboring nodes in order to guarantee incremental progress.

3 Replication protocol

The replication protocol addresses replica and mapping management operations: (i) when, what, and where to replicate, (ii) when, and what to de-replicate, (iii) what servers to keep in a map, and how many, (iii) what servers in a map to advertise, and how many, and finally (iii) how to combine two maps for the same node.

3.1 Server load metrics

Replication is used to improve server load balance and routing resiliency in the face of network failures. The first objective is pursued explicitly while the latter follows implicitly from the first: hosting servers for nodes with failed replicas will incur more load after failure than before, and will replicate again to meet new load conditions. Load balance is our first and most important fairness criterion.

We assume that a *normalized load metric* can be defined for all participating servers. A server's load in this metric is valued in the interval $[0, 1]$ with the semantics of the extremes being "no load" and "full capacity load" respectively. The normalization process ensures that system heterogeneity is accounted for. Additionally, the load metric must be:

1. Linearly comparable: given two load values, l_1 and l_2, the value l_1/l_2 should mean that server 1 has l_1/l_2 times more load than server 2.
2. Locally defined: the load must be defined exclusively based on local server information (busy cycles, memory requirements, incoming queue occupancy, etc), and independent of other servers' load condition. Load definition need not be the same for all servers. This accounts for machine heterogeneity.

The replication model is independent of any specific load metric, as long as such metrics respect the above requirements. We evaluate the replication protocol using a simple load measure: fraction of server busy time over a window period τ (e.g. half a second).

A server initiates load balancing sessions by replicating nodes on other servers when its load exceeds a *high-water threshold*, l_{high}. This threshold is a measure of the load-imbalance we are willing to tolerate, and can automatically be set in proportion to the overall system utilization. A server will agree to host new replicas if there is a difference of at least l_{diff} between the load of the requester and its own load.

3.2 Node ranking

When a server's load exceeds the high-water threshold, hosted nodes that comparatively incur more load on the server will be further replicated. Load based node ranking is achieved by identifying the nodes for which processing is performed whenever routing a query.

The criteria for node ranking is given by assigning *node weights* to each node hosted by a server. The weight for each node is proportional to the load incurred by the server on the node's behalf. Simple counter variables can be maintained to implement node weights. With each incoming query the appropriate counter is incremented, and all counters are rescaled periodically to approximate recent demand patterns.

3.3 Replica creation

The protocol for creating new replicas proceeds as follows. Assume source server S_S has load l_S.

1. Replication is triggered when a server's load exceeds the high-water threshold, $l_S > l_{high}$. A server checks its load after each processed query.
2. Among all the servers that it knows about, S_S picks the one with minimum load, S_D. S_S makes this decision based on load information that it has for servers, not the actual load of servers. S_S contacts S_D and learns its actual load, l_D.
3. If $l_S - l_D \geq l_{diff}$, S_S will replicate nodes on S_D. Given a node ranking for server S_S, w_i s.t. $w_1 \geq w_2 \geq \ldots \geq w_n$, the top ranked m nodes will be replicated on S_D, where m is the smallest number s.t. $\sum_{i=1}^{m} w_i / \sum_{i=1}^{n} w_i \geq \frac{1}{2}\frac{l_S - l_D}{l_S}$.
4. S_S and S_D will adjust their loads to $l_S = l_S - \frac{1}{2}(l_S - l_D)$ and $l_D = l_D + \frac{1}{2}(l_S - l_D)$ to reflect the ideal load redistribution targeted after replication. This acts as a hysteresis and will prevent replica thrashing.
5. If $l_S - l_D \geq l_{diff}$ above is not met, then S_S makes another attempt of selecting a destination server, and the protocol continues with step 2. After a few failed attempts, S_S aborts the current replication session and initiates another one after a short delay.

3.4 Controlling the extent of replication

Our second fairness criterion is the amount of replicated state maintained per server. We need to bound the number of replicas, either globally or on a local basis. Otherwise, nodes may continually be replicated in response to fluctuating load, and the system could easily converge to extreme configurations where each node is hosted by every server. We constrain the number of replicas hosted by a server to be proportional to the number of nodes owned by the server.

The *replication factor*, k_{repl}, controls the maximum number of replicas hosted per server relative to the owned nodes. The replication factor need not be the same for all servers. Allowing servers to replicate nodes in proportion to the number of hosted nodes is a locally enforced and, we believe, fair policy. A locally enforced replication factor translates easily to global constraints. The overall number of replicas is bounded by the highest replication factor relative to the number of all nodes in the system. Note that we impose constraints on a per server basis. Nodes can still have as many replicas as globally allowed by the replication factor.

3.5 Replica deletion

To heed the replication factor and at the same time allow the model to continuously adapt to changing demand patterns, node replicas have to be deleted. A hosting server may decide at any time to evict replicas that have not been in use for a long time, i.e. low ranking nodes. Additionally, some replicas may have to be evicted (as dictated by k_{repl}) by a server S_D when some other server S_S requests that some of its nodes be replicated on S_D. In such cases, S_D will delete as many replicas as needed starting with the lowest ranking node and proceeding in increasing order.

Replica deletion is a local process as it involves only the server that hosts the replica. Other servers will learn about deletions in a lazy manner, or may not learn at all about some of the evictions. Inconsistent views are caused by stale mapping configurations and adversely affect routing performance. We do not specify any consistency model for managing mapping configurations. A limited form of control can be exercised by removing stale entries from maps when they are routed through servers. The degree of inconsistency can be further reduced by using inverse-mapping information.

3.6 Inverse-mapping digests

Maps provide a name resolution service by identifying some of the servers that host a node. Resolving node names to hosts is needed every time routing is done on behalf of the node. The inverse function, resolving a server to node names hosted by that server, improves performance and routing quality.

The TerraDir replication protocol benefits from inverse-mapping approximations (*digests* from now on) as follows. Each server generates a digest regarding its hosted nodes. The digest is a Bloom filter [2], and its value is determined by hashing the names of all the nodes hosted by the corresponding server. The only allowed operation on a digest is testing node names against it, and producing a yes/no answer with possible false positives. Digests are used to discover additional shortcuts in the namespace and to maintain up-to-date node maps.

3.6.1 Discovering additional shortcuts

The standard routing algorithm is a minimizing procedure. A server S routing query q always chooses the closest node to q that it knows about, n, and forward the query to one of the servers in n's map. Using inverse-mapping digests, S might indirectly know about some other node, h, that is even closer to q than n. Node h is discovered as follows.

1. Assume S generates all node names that it can infer, and let this set be $Pref$. $Pref$ includes hosted, neighboring, and cached node names, as well as destination name q.

2. By performing prefix extractions, S also includes in $Pref$ ancestor names —all the way to the root— for all these nodes. Each of the names in $Pref$ can be tested against digests of servers that S knows about.

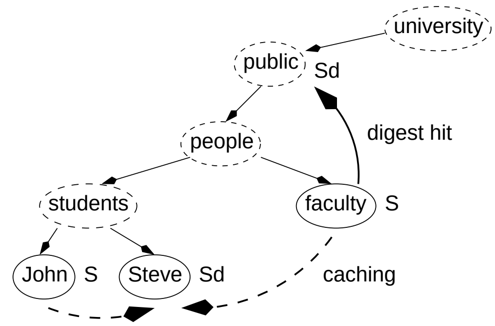

Figure 2. Shortcut taken from server S due to a hit of *university/public* in the digest of S_d.

3. Assume a hit for some name $h_d \in Pref$ occurs in a digest associated with server S_d, and that h_d is closer to q than n. In this case S can optimize the routing by forwarding the query to S_d instead of some server from n's map.

4. By choosing h to be the closest h_d to q, server S is guaranteed to make the best decision it can in terms of namespace distance.

We illustrate the procedure in Fig. 2. Server S hosts nodes */university/public/people/faculty* and */university/public/people/student/John*. Names in $Pref$ are shown using dashed ellipses. In this case, the names of all four such nodes are prefixes of hosted nodes. S also has an entry for */university/public/people/students/Steve* in its cache. The mapping for the cached entry includes one of the node's hosts, S_d, and S_d's digest. S_d also hosts */university/public*, and S gets a hit for node */university/public* in S_d's digest present at S. Thus S can forward the current query to S_d and skip node */university/public/people*.

3.6.2 Pruning node maps

Consider server S that knows about node n by keeping some map of it. For each of the servers in the map, S keeps the corresponding inverse-mapping digest. Thus S can produce a potentially more accurate map for n by testing node n against each of these digests. Servers in n's map whose test fails can safely be eliminated from the map.

False positives associated with digests may preclude S from eliminating one or more servers from a map. Further, not only the node's map but some of the digests used to prune the map may be outdated as well. Despite the fact that node map pruning is a conservative operation, it enables the routing procedure to perform with close-to-perfect accuracy.

3.7 Node mapping management

A node map associates a node name with a (possibly incomplete and inaccurate) list of servers that own or replicate the node. Servers keep mapping information for owned, replicated, neighboring and cached nodes. The following policies govern how replica information is spread in the network, and how servers use replicas for routing queries.

Map size: A node map contains at most k_{map} entries for scalability reasons. The constraint is in effect for maps kept at servers, as well as maps propagated in the network. Maximum map size is orthogonal to how many replicas a node can get, or how many replicas a server hosts.

New replica advertisement: Servers advertise replicas created for their hosted nodes. Each server that has replicated one of its hosted nodes keeps entries for the most recent created replicas (up to k_{map}) in the node's map. These entries are advertised with every outgoing message that includes the node's map. Traffic in excess will quickly be diverted to newly created replicas.

Map merging: Maps are merged whenever a server keeps a map for a node, and an incoming query contains another map for the same node. Map merging is performed such that (i) the above conditions are met, and (ii) the rest of the entries in the resulting map are chosen at random from the choice left. The same two maps may have to be merged twice, once for the resulting map kept at the server, and a second time for the resulting map to be further propagated with the query currently processed.

Disseminating replica information: Map replica configurations for a node are disseminated along query paths whenever information about the node is present in forwarded messages. Additionally, information about newly created replicas is back-propagated at each forwarding step if applicable. For instance, if server S_1 forwards a query to S_2 on behalf of node n, and S_2 has recently created any replicas for n, then S_2 will let S_1 know about such replicas.

Replica selection: Given a node's map present at a server, replica selection is performed by the server whenever a message needs to be forwarded to one of the node's hosts. For lookup messages any data or map replica of the node can be selected, whereas for data retrieval messages only data replicas can be selected. In both cases the destination host is chosen at random from the available choice, i.e. servers in the node's map present at the server.

Map filtering: Inverse-mapping digests are used to filter out stale entries from maps. Map filtering is a best-effort procedure carried out locally whenever a server updates the inverse-mapping information of other servers. Our evaluation is based on a more conservative and thus less efficient approach: filtering is performed at replica selection and map merging, i.e. whenever node maps are used or modified at a server.

4 Evaluation

We evaluate the presented replication protocol in a TerraDir simulated environment. We focus on adaptivity to user input, system load balance and global utilization, stabilization, long-term behavior, and finally scalability.

4.1 Methodology

We consider $4,096$ servers. Service times are exponentially distributed with a mean of $T_s = 20$ milliseconds for each server. The mean query arrival rate is modeled with a Poisson distribution and varies from $\lambda = 2,000$ requests per second to $\lambda = 20,000$ requests per second, globally. Each server has a request queue of size 32 with queries arriving in excess being dropped. The application layer network time is constant at 25 milliseconds. We do not model network contention.

Lookups are initiated uniformly at source servers. Destination nodes are chosen either uniformly at random ($unif$ traces), or with locality according to the Zipf [19] law of popularity vs. ranking ($zipf$ traces). Our study involved both synthetic and real-world TerraDir namespaces:

- As example of a synthetic namespace (N_S) we consider $32,767$ nodes arranged in a perfectly balanced binary tree. Uniform query streams for this namespace are denoted by $unif_S$, while streams with locality are denoted by $zipf_S^{\alpha}$. The order α covers the whole domain of interest: $.75$, 1, 1.25, and 1.5 for heavily skewed requests.

- File-systems are the most common hierarchical structures in use. We consider one of the Coda [14] servers (*barber*) logged throughout one month of activity (January 1993). Files accessed during this month together with their ancestors were included in this namespace (N_C), for a total of $89,382$ nodes. Uniform query streams for N_C are denoted by $unif_C$, while streams with locality are denoted by $zipf_C^{\alpha}$.

Both namespaces considered are mapped uniformly at random on the $4,096$ servers. In preparing the $zipf$ streams, node ranking is established by randomly ordering all the nodes in the namespace. Query streams are combinations of $unif$ and $zipf$ streams. For instance, we may prepend a sequence of $zipf$ streams with a $unif$ stream to allow a "cold" system to compensate for hierarchical bottlenecks and replicate the top nodes in the namespace. Thus we limit the interference between system warmup effects, and those induced by demand distribution. Some experiments are run with locality streams that instantly and at random change node rankings, so that we can quantify how the model adapts to sudden variations in popularity (i.e. shifting hot-spots).

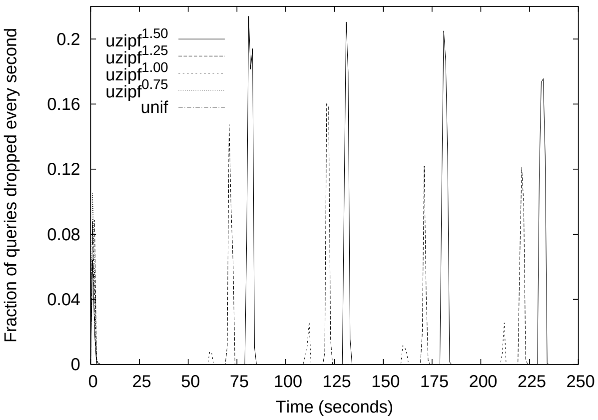

Figure 3. Dropped queries (relative to $\lambda = 2,000$) over time for namespace N_S.

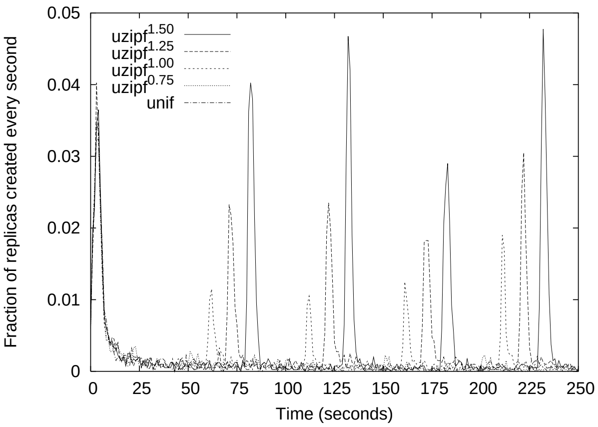

Figure 4. Created replicas (relative to $\lambda = 4,000$) over time for namespace N_C.

4.2 Adaptation

We assess the model's capability to adapt to changing conditions, in this case locality variations in the query stream. For each of two namespace we run two types of streams for 250 seconds: *unif* and $uzipf^{\alpha}$. The latter is given by the sequence $\{unif, zipf^{\alpha}, zipf^{\alpha}, zipf^{\alpha}, zipf^{\alpha}\}$.

Fig. 3 shows the fraction of dropped queries relative to the query insertion rate for namespace N_S. For ease of presentation we allowed for the *unif* component of the $uzipf^{\alpha}$ streams to run longer in increments of 10 seconds, for various Zipf order values. The drops in the first seconds are due to hierarchical stabilization when the system replicates nodes at the top of the namespace hosted by overloaded servers. The spikes of the graph correspond to instantaneous and random changes in node popularity (for $uzipf^{1.25}$ such changes occur at seconds $70, 120, 170$, and

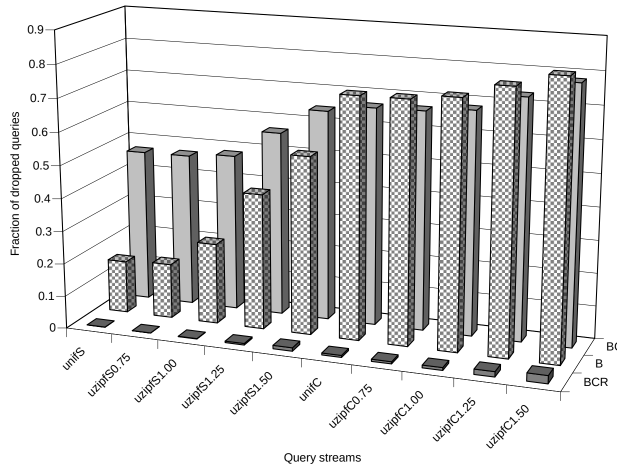

Figure 5. Fraction of dropped queries with combinations of the base system (B), caching (C), and replication (R).

220). Fig. 4 shows the system's reaction for namespace N_C to overloading conditions in terms of the number of replicas created. We doubled the query arrival rate to keep the system at approximately the same utilization. The replication model adapts well to both hierarchical bottlenecks and sudden hot-spot fluctuations. The overall number of query drops is at most 2.5% when randomly changing highly skewed input ($\alpha = 1.5$) four times in a row over a short period of time (250 seconds). A quarter of the 2.5% are simulation side-effects due to hierarchical stabilization. The number of load balancing messages is at least two orders of magnitude less than the number of queries submitted.

Running the same experiments with replication disabled causes a large fraction of queries to be dropped to a point where the system is barely usable. If only caching is used while replication is still disabled, we see further aggravation in performance for namespace N_S, and slight improvements for namespace N_C. These points are made clear by Fig. 5 where we compare the replication protocol with a base system, and one which employs only caching.

4.3 Utilization and load balance

Utilization distribution is our main fairness criterion. We define the computational utilization of a server over a second as the fraction of that second that the server is busy processing queries. We target three utilization factors: $U = 10\%; 20\%; 40\%$, and approximate them with query rates $\lambda = 2,000; 4,000; 8,000$ for namespace N_S, and $\lambda = 4,000; 10,000; 20,000$ for namespace N_C. We use query streams *unif* and $uzipf^{1.00}$ defined previously.

Fig. 6 shows on the left side the mean measured load, and the load on one of the most heavily loaded servers every second, for namespace N_S. Periodical peaks are due to

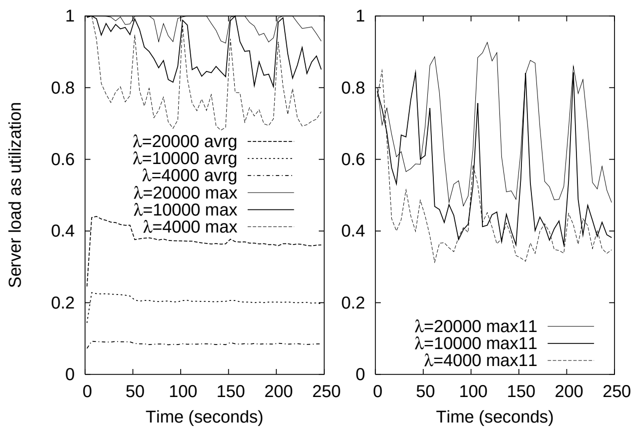

Figure 6. Average and maximum server load for query streams $uzipf_C^{1.00}$ (left). Maximum server load averaged over 11 seconds (right).

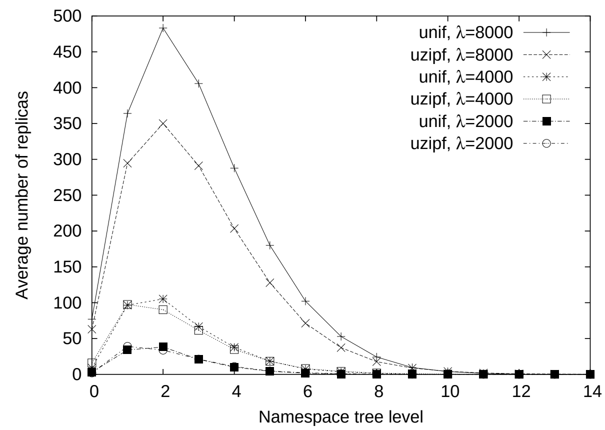

Figure 7. Average number of replicas created for each level of namespace N_S (the root is on level 0) with uniform and Zipf queries.

locality changes in the $uzipf$ query stream. Note that the maximum load tends to go below l_{high} (0.75 in this case) if given enough time. With higher query rates the global mean load is itself approaching the threshold. In such cases it is proportionately harder to bring the maximum load below a constant high-water threshold. Note that servers at full utilization stay there only for a few seconds with each Zipf change, for all query rates shown.

We establish the transiency of highly-loaded server conditions when looking at larger than 1-second intervals. At each second we identify the most heavily loaded servers and average their load over 11 seconds. We show the load thus smoothed in the right side of the figure. The load distribution has improved substantially with the maximum load approaching the mean, notably for higher λ values. Highly-loaded servers experience transient conditions, and by defining load balance over larger intervals we get increasingly better results.

In Fig. 7 we show how the system reacts to hierarchical bottlenecks. For each level of namespace N_S we show the average number of replicas created for nodes on that level, with $unif$ and $uzipf^{1.00}$ query streams, and various query arrival rates. Note that nodes on level 2 tend to have more replicas than their ancestors. Pointers to nodes on level 2 have a high chance of staying in a server's cache. Many of the routes that would normally go all the way up to nodes on levels 0 or 1 are thus using level 2 shortcuts. Nodes on level 3 are less likely to be found in a server's cache since there are more nodes on this level than on level 2, etc.

4.4 Stabilization and long-term behavior

We are interested in establishing long-term behavior characteristics: (i) whether with no changes in input patterns the replication model reaches a quiescent state where very few or no replicas are being created, and (ii) whether routing accuracy and efficiency is preserved with extreme changes in input patterns that will entail many replicas creations and deletions.

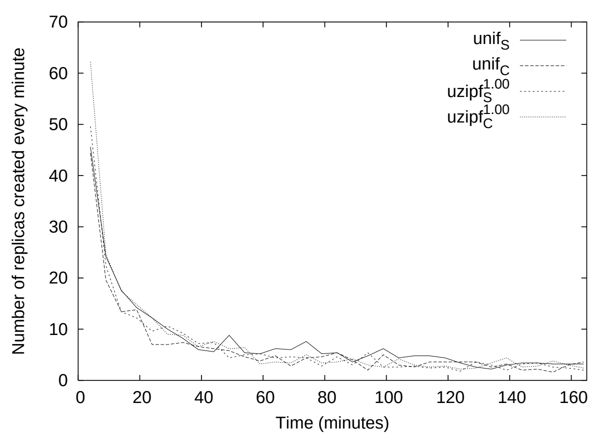

Figure 8. Replicas created over $10,000$ second runs with uniform and Zipf queries.

To asses stabilization we present results from runs with $unif$ and $uzipf^{1.00} = \{unif, zipf^{1.00}\}$ queries for $10,000$ seconds; the uniform component of $uzipf^{1.00}$ lasted for 100 seconds. 20 million queries were run for namespace N_S ($\lambda = 2,000$), and 40 million for N_C ($\lambda = 4,000$).

Fig. 8 shows the number of replicas created every minute. The replication protocol reaches a rate of 2.5 replicas created per minute after $10,000$ seconds. The rate keeps decreasing beyond the time frame shown, with the curve for the whole run resembling an exponentially decaying variable. The replica creation rate is equivalent to one replica

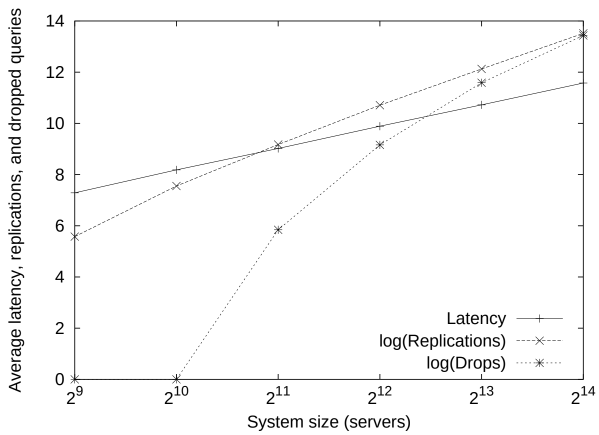

Figure 9. Scalability of query latency, degree of replication, and dropped queries.

created every $48,000$ queries run for namespace N_S, and $96,000$ queries for N_C. The replication protocol stabilizes with time for constant request distributions.

The evaluation conducted so far was based on a replication factor $k_{repl} = 2$, and lead to very few replica deletions. We ran experiments with $k_{repl} = 0.125; 0.25; 0.5$ on query streams $uzipf^{1.50} = \{unif, [zipf^{1.50}]^{99}\}$. Each $zipf^{1.50}$ component lasted for 100 seconds, for an overall of $10,000$ seconds. Low replication factors together with repeated shifts of high-order hot-spots ($\alpha = 1.50$) induce major changes in replica configurations. For space reasons we will only summarize the results. Inverse-mapping digests are good approximations of optimal behavior (i.e. routing with perfectly accurate information, as if given by an oracle). There is enough opportunity to transitively disseminate inverse-mapping information in the network, such that routing accuracy is maintained within the optimal range.

4.5 Scalability

We scale system size exponentially and look at query latency, the number of replica creations events, and the number of dropped queries. Servers range from 2^9 to 2^{14} in incremental powers of 2. The number of nodes per server is kept constant at 8, with the overall number of nodes ranging from 2^{12} to 2^{17}, arranged in a perfectly balanced binary tree. Cache sizes are logarithmic in system size, ranging in increments of 2, from 18 to 28 cache slots per server. k_{repl} is kept constant 2, and k_{map} varies logarithmically with system size, from 2 to 7. Finally, λ is proportional to system size and takes values $250, 500 \ldots 8,000$.

Fig. 9 shows the average query latency, the number of replication events, and the number of dropped queries as a function of system size. The last two are shown on a logarithmic scale. Latency scales logarithmically with system size, the degree of replication scales linearly, while the number of dropped queries scales proportionately and approaches linearity for large systems. The replication protocol is scalable.

5 Related work

The Domain Name System (DNS) [10] is a cornerstone for Internet and one of the most widely deployed directory service to date. Even though DNS servers are required for their namespace resolution functionality, the infrastructure is not provisioned to handle queries for arbitrary resource records. They key to its success lies in a carefully tuned caching scheme that enables queries to be resolved in the local domain. TerraDir caches consist of pointers in the namespace and provide only routing functionality.

Studies [1, 3] show that both spatial and temporal reference locality are present in requests submitted to web servers or proxies, and that such requests follow a Zipf-like distribution. Distributed caching protocols [8] have been motivated by the need to balance the load and relieve hotspots on the World-Wide-Web. Similar Zipf-like patterns were found in traces collected from Gnutella. Caching the results of popular Gnutella queries for a short period of time proves to be effective [16]. Recent work [9, 5] considers static replication in combination with a variant of Gnutella searching using k random walkers. Replicating objects proportionally to their popularity achieves optimal load balance; replicating them proportionally to the square-root of their popularity minimizes the average search latency.

Freenet [4] replicates objects both on insertion and retrieval on the path from the initiator to the target mainly for anonymity and availability purposes. It is not clear how a system like Freenet would react to query locality and hotspots. Chord [17], CAN [11], Pastry [12] and Tapestry [18] are peer-to-peer systems using the common approach of a distributed hash table for location. Assignment of objects to hosts is performed by mapping the object space into a virtual namespace, which is convenient because of the uniform spread of objects. Load balancing is thus automatically achieved for uniformly distributed requests.

CFS [6] is a P2P read-only file system that uses the Chord lookup service to locate files dispersed throughout the network. Data placement granularity is very fine and consists of file blocks. Replication and caching is achieved on a file block basis which is convenient since (i) select portions of a large file may be more popular than others, and (ii) parts of popular files are distributed across different servers. CAN allows for different redundancy schemes: multiple coordinate spaces can be in effect simultaneously, zones can be overloaded by assigning each of them a set of peer servers. Replication is achieved by using multiple hash functions on the same data item. Pastry replicates an object on the k servers whose identifiers are closest to the object key in the namespace. Pastry's locality properties make it likely that among the k replicas of an object, the one that is closest to the requesting client, as given by IP metrics, is reached first.

None of the hash-based schemes feature adaptive replication mechanisms similar to ours. Instead, caches are used to spread popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. CFS for instance uses k-replication similar to the above for data availability, and populates all the caches on the query path with the destination data after the lookup completes. A recent analysis [13] of two popular P2P file sharing systems concludes that the most distinguishing feature of these systems is their heterogeneity. We believe that the adaptive nature of our replication model makes it a first-class candidate for exploiting system heterogeneity.

6 Concluding remarks

We have presented a novel approach to replication in peer-to-peer systems. Our work stems from the observation that routing load, incurred by P2P lookup services, is orthogonal to load incurred by data retrieval, and can be efficiently managed if approached separately. Routing state is replicated in an ad-hoc manner, and the replication protocol can be combined with any data replication mechanism.

The protocol addresses both hierarchical and demand-induced bottlenecks. Adaptivity is based upon information profiled online. This enables it to deliver low latencies and good load balance even when demand is heavily skewed. Further, it withstands arbitrary and instantaneous changes in demand distribution. Our approach is lightweight, scalable, and deals with soft-state in the sense that there is no need for replica consistency models to be specified.

Two lessons have been learned throughout our study. First, simple heuristics for estimating server load, node weights, and replication policies perform well. We did not see a strong enough case for more sophisticated methods. Second, there is enough opportunity to transitively disseminate information in the network by exclusively using in-band means. Queries are the source of the problem (host-spots and hierarchical bottlenecks), and the disruption caused by an individual query can be addressed by piggybacking on query messages limited amounts of information about replica configurations and server loads and digests.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec 1996.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the INFOCOM*, pages 126–134, Mar 1999.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, Jul 2000.

[5] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, Aug 2002.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th SOSP*, Chateau Lake Louise, Banff, Canada, Oct 2001.

[7] P. Druschel and A. Rowstron. PAST: a large-scale persistent peer-to-peer storage utility. In *Proc. of the 8th IEEE HotOS*, Schloss Elmau, Germany, May 2001.

[8] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of the 29th Annual ACM STOC*, pages 654–663, El Paso, TX, May 1997.

[9] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of the ICS*, New York, NY, Jun 2002.

[10] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *Proc. of ACM SIGCOMM*, pages 123–133, Stanford, CA, Aug 1988.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of the ACM SIGCOMM*, San Diego, CA, August 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the* 18^{th} *Intl. Conf. on Distributed Systems Platforms*, Heidelberg, Germany, Nov 2001.

[13] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN*, San Jose, CA, Jan 2002.

[14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[15] B. Silaghi, B. Bhattacharjee, and P. Keleher. Query routing in the TerraDir distributed directory. In *Proc. of SPIE ITCOM*, Boston, MA, August 2002.

[16] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. February 2001.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable P2P lookup service for Internet applications. In *Proc. of SIGCOMM*, San Diego, CA, Aug 2001.

[18] B. Zhao, K. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, April 2001.

[19] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, MA, 1949.