Adaptive Replication in Peer-to-Peer Systems

Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher
Department of Computer Science
University of Maryland, College Park
(gvijay,bujor,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 items is relatively uniform. We describe a lightweight, adaptive, and system-neutral replication protocol, LAR, that delivers low latencies and good load balance even when demand is heavily skewed.

Simulation of LAR in combination with both the Chord and TerraDir systems shows that LAR quickly adapts to non-uniformity in both the underlying system topology and in the input stream. Further, we demonstrate better performance than functionally similar application-layer protocols, using an order of magnitude less network bandwidth.

I. Introduction

Peer-to-peer (P2P) systems perform any number of different functions, but their most fundamental task is that of locating data. Recent work [1], [2], [3], [4] has shown the ability to deliver low latencies and good load balance when demand for most items is relatively equal[1]. However, the distribution of demand for real data items is often skewed, leading to poor load balancing and dropped messages. This paper describes and characterizes a lightweight, adaptive, system-neutral replication protocol (LAR) that is efficient at redistributing load in such circumstances, and which improves query latency and reliability as well.

For purposes of this paper, we define a P2P system as a distributed system where functionally identical servers export data items to each other. The defining characteristic of many such systems is that they are completely decentralized. There is only one class of server, and all decisions, from routing to replication, are local.

While P2P systems provide basic services like data location, P2P *applications* provide high-level functionality (such as file sharing [5], [6], multimedia streaming, event notification [7], etc.) using an underlying P2P system. Much of the complexity of such systems arises because the environment of P2P systems is potentially much different than that of traditional distributed systems, such as those hosted by server farms. A single P2P system instance might simultaneously span many different types of constituent elements, such as dedicated servers, idle workstations, and even non-idle workstations.

Regardless of the underlying system topology, P2P systems need some form of caching and/or replication to achieve good query latencies, load balance, and reliability. The work described in this paper is primarily designed to address the first two: query latency and load balance. A query is merely an instance of data location (a lookup) with a fully qualified name. It turns out that average query latency is relatively easy to minimize, and we do not consider it here further[2]. However, distributing load equitably is more difficult.

For example, many recent systems [1], [4], [2], [3] attempt to balance load by using cryptographic hashes to randomize the mapping between data item names and locations. Under an assumption of uniform demand

[1]Heavyweight caching mechanisms used by P2P applications can cause problems. See Section V-E for details.
[2]Our results show comparable query latencies for both Chord and TerraDir. Unsurprisingly, our adaptive replication helps, sometimes dramatically.

for all data items, the number of items retrieved from each server (referred hereafter to as "destination load") will be balanced. Further, *routing load* incurred by servers in hash-based schemes will be balanced as well.

However, if demand for individual data items is non-uniform, neither routing nor destination load will be balanced, and indeed may be arbitrarily bad. The situation is even worse for hierarchical systems such as TerraDir [8], as the system topology is inherently non-uniform, resulting in uneven routing load across servers.

So far, this problem has usually (e.g. PAST [5], CFS [6]) been addressed in an end-to-end manner by caching at higher levels. However, the resulting protocol layering incurs the usual inefficiencies, and causes functionality to be duplicated in multiple applications. More importantly, our results show that while these schemes can adapt well to hot spots, they perform poorly under heavy load because of their high overhead.

This paper describes a lightweight approach to adaptive replication that does not have these drawbacks. The protocol described here is much more lightweight, can balance load at fine granularities, accommodates servers with differing capacities, and is relatively independent of the underlying P2P structure.

However, the main contribution of this paper is to show that a minimalist approach to replication design is workable, and highly functional. We derived a completely decentralized protocol that relies only on local information, is robust in the face of widely varying input and underlying system organization, adds very little overhead to the underlying system, and can allow individual server loads to be finely tuned.

This latter point is important because of the potential usage scenarios for P2P systems. While P2P systems have been been proposed as the solution to a diverse set of problems, many P2P system will be used to present services to end users. End users are often skeptical of services that consume local resources in order to support anonymous outside users. User acceptance is often predicated on the extent to which end users feel they have fine-grained control over the intrusiveness of the service.

The rest of this paper is structured as follows. Section II describes related work. Section III gives an overview of our model and design goals. Section IV describes the protocol in more detail. Finally, Section V describes our simulation results and Section VI summarizes our findings and concludes the paper.

II. BACKGROUND

This section briefly summarizes related work, and then describes the Chord [1] and TerraDir [8] protocols in some detail. We use Chord and TerraDir as representative P2P system protocols in the simulations described in Section V.

A. Related work

Studies [9], [10] show that both spatial and temporal reference locality are present in requests submitted at web servers or proxies, and that such requests follow a Zipf-like distribution. Distributed caching protocols [11] have been motivated by the need to balance the load and relieve hot-spots on the World-Wide-Web. These approaches target client-server communication models which are different than ours.

Similar Zipf-like patterns were found in traces collected from Gnutella [12], one of the most widely deployed peer-to-peer systems. Caching the results of popular Gnutella queries for a short period of time proves to be effective in this case [13]. Our path propagation is a generalization of this caching scheme.

Recent work [14], [15] considers static replication in combination with a variant of Gnutella searching using k random walkers. The authors show that replicating objects proportionally to their popularity achieves optimal load balance, while replicating them proportionally to the square-root of their popularity minimizes the average search latency. We addressed a static replication approach for TerraDir in previous work [8].

Freenet [16] 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 hot-spots.

Chord [1], CAN [2], Pastry [3] and Tapestry [4] are hash-based peer-to-peer systems. They use the same approach of mapping the object space into a virtual namespace where assignment of objects to hosts is more convenient because of the uniform spread of object mappings. We expand on Chord in the second part of this section.

CAN defines a d-dimensional Cartesian coordinate space on a d-torus. Assuming an evenly partitioned space among the N servers, the path length between any two servers is $O(dN^{1/d})$, with each server maintaining information about d neighbors. 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, while replication is achieved by using multiple hash functions on the same data item.

Pastry is the routing and object location layer used by PAST [5]. Given parameter b, Pastry maintains routing tables of size approximately $O(2^b \log_{2^b} N)$ and performs routing in $O(\log_{2^b} N)$ steps, assuming accurate routing tables and no recent server failures. Replication is attained by storing an object on the k Pastry 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, will be reached first. Tapestry is very similar in spirit to Pastry, both being inspired by the routing scheme introduced by Plaxton et al. [17].

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 [6] 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 great deal of work addresses data replication in the context of distributed database systems. Adaptive replication algorithms change the replication scheme of an object to reflect the read-write patterns and are shown to eventually converge towards the optimal scheme [18]. Concurrency control mechanisms need to be specified for data replication to guarantee replica consistency.

A recent analysis [19] of two popular peer-to-peer 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 would make it a first-class candidate in exploiting system heterogeneity.

B. Hash-based systems

We use Chord to represent the class of hash-based systems, but there is a great deal of other work on systems based on the same idea (e.g. Chord, Pastry, Tapestry, CAN). These systems differ from Chord in important ways, but few of these differences should affect the applicability of our replication approach.

Chord is essentially a fast lookup service based on the notion of *consistent hashing* [11], here implemented via SHA-1. The names of data items exported by the servers are hashed to item keys; server ID's are hashed to server keys. With large numbers of servers, and m-bit keys, the servers will map relatively evenly around an identifier circle whose locations are specified by indices $0 \mathrel{..} 2^m - 1$. Key k's *home* is the first server whose identifier is equal to or larger than k.

The only bit of consistent state required by Chord is a server's successor. If each server reliably knows its successor, routing is guaranteed to complete successfully. However, using only successor pointers would result in $O(n)$ query latency, so the successor pointers are enhanced with a routing table called the finger table.

A finger table has at most m entries. The i^{th} entry at server n contains the identity of the first server that succeeds n by at least 2^{i-1} on the identifier circle. Thus, succeeding finger table entries reach further and further around the circle and a server can efficiently route a packet by forwarding each message to the finger table entry closest, but less than or equal to, the message destination's key. If finger table information is accurate, this approach will usually route packets successfully in at most $O(\log N)$ steps

Note that Chord servers both "export" and "serve" items. Exported items are hashed and inserted into the name space. Served items are those that are mapped to a server by the hash mechanism.

The hash-based approach helps load balance because even if all of the data items exported from one site are in high demand, they will generally be served by different machines. Similarly, routing load is distributed because paths to items exported by the same site are usually quite different.

Just as importantly, the virtualization of the namespace provided by consistent hashing provides a clean, elegant abstraction of routing, with provable bounds for routing latency.

C. Hierarchical systems

The advantages of hash-based systems are clear, but there are also potential disadvantages [20]. First, hash-based virtualization destroys locality. By virtualizing keys, data items from a single site are not usually co-located, meaning that opportunities for enhancing browsing, prefetching, and efficient searching are lost. However, we do not consider these functions in this paper.

Second, hash-based virtualization discards useful application-specific information. The data used by many applications (file systems, auctions, resource discovery) is naturally described using hierarchies. A hierarchy exposes relationships between items near to each other in the hierarchy; virtualization of the namespace discards this information.

We use TerraDir as a representative of non-hash-based systems. Assume a 1-to-1 mapping between data items and servers for the moment. TerraDir assumes that the input data is already organized hierarchically, and routes packets over the resulting tree structure. More specifically, queries are routed from the source to the greatest common prefix of the source and the destination, and then to the destination.

For example, assume that $/a$ is the root of the tree. A query from $/a/b/c/d$ to $/a/b/f/g/h$ is routed "up" to $/a/b$, and then "down" to the destination[3]. If the source is the ancestor (descendent) of the destination, the greatest common prefix is merely the source (destination).

This scheme generalizes to multiple items per server by ensuring that the local item "closest" to the destination is always picked when forwarding messages towards the destination. Note that, unlike in hash-based schemes, an item's home in TerraDir is the server that exported it.

The basic scheme is enhanced by caching item-to-server mapping.

D. Summary

The focus of this paper is the replication protocol, not a comparative evaluation of the two different system-structuring philosophies. However, we here summarize the main advantages and disadvantages of each approach.

Hash-based systems like Chord usually routes messages in $O(\log N)$ steps, and can balance routing load assuming uniform query distribution (items all serve as query sources and destinations with approximately the same frequency).

Hierarchical systems like TerraDir also route messages in approximately $O(\log N)$ steps. Further, preserving the spatial locality inherent in the application label hierarchy allows spatial locality in the query

[3]Our preferred orientation for visualizing the tree is with the root at the top.

stream (browsing, prefetching, etc.) to be handled efficiently. However, performance will degrade if the hierarchy is not approximately balanced, and servers hosting non-leaf items will receive disproportionate amounts of traffic.

Caching and replication are crucial to balancing routing load in hierarchical systems, whereas hash-based systems implicitly balance routing load for uniform query distributions.

III. Goals and Approach

A. Goals

We have two primary goals. First, we will address overload conditions. We assume that each server has a locally configured resource capacity, which indicates the number of queries that can be routed or handled per second, and a queue length, which specifies the number of queries that can be buffered until additional capacity is available. Any arriving traffic that can not be either processed or queued by a server is dropped. The first goal of the replication protocol is to distribute load over replicas such that messages are not lost.

Second, we will attempt to balance load. While P2P systems can deliver functionality similar to that of generic server farms, they differ in that constituent machines are not necessarily dedicated processors. A system constructed from idle resources on user machines might want to satisfy some abstraction of fairness by ensuring that machines all perform nearly the same amount of work. However, this notion can be difficult to define exactly, and even more difficult to implement efficiently.

Instead, we take a slightly different tack. In addition to capacity and queue lengths, each server defines high-water and low-water thresholds. The intent is that the high-water threshold represents a load beyond which a server will shed load with prejudice, i.e. to any other server available. A server whose load is between the high-water and low-water threshold attempts to hand off load to lightly-loaded servers only. Any server whose load is below the low-water threshold does not shed load.

By default, the system sets high-water and low-water thresholds for each server based on fractions of servers' capacities. We assume that these fractions are constant across the system in the simulations here, but the protocol will work unaltered with non-uniform fractions.

We will use the term "load balance" in the rest of this paper to refer to this sense of distributing load.

B. Approach

Our approach is based on adaptivity and local decision-making. Adaptive protocols are necessary to cope efficiently with dynamic query streams, or even static streams that differ from expected input.

Making local decisions is key to scaling the system, as P2P systems can be quite large. For example, the popular KaZaA file-sharing application routinely supports on the order of two million simultaneous users, exporting more than 300 million files. Global decision-making implies distilling information from at least a significant subset of the system, and filtering decisions back to them. Further, there is the issue of consistency. There is a clear tradeoff between the "freshness" of global information summaries and the amount of overhead needed to maintain a given level of freshness. Finally, systems using global information can be too unwieldy to handle dynamic situations, as both information-gathering and decision-making require communication among a large subset of servers.

The choice of local decision-making has far-reaching implications. For one, local decisions might be poor if locally available information is unrepresentative of the rest of the system.

Second, local decision-making makes it difficult or impossible to maintain the consistency of global structures, such as replica sets for individual data items. A *replica set* is just a possibly incomplete enumeration of replicas of a given data item, which are the default unit of replication. Requiring that the "home" of a data item be reliably informed of all new and deleted replicas could be prohibitively costly in a large system.

This difficulty led us to use soft state whenever possible. For example, instead of keeping summaries of all replicas at a data item's home, we allow some types of replicas to be created and deleted remotely without having any communication with other replicas or the home.

Our conclusions are intended to be independent of P2P structure. Hence, we built our replication and caching layer into the Chord simulator, and base our conclusions on simulations over a typical hash-based P2P architecture (Chord), and over a very different tree-based scheme (TerraDir).

IV. The Protocol

This section describes the details of the adaptive replication scheme, LAR. We first cast the description in terms of the TerraDir protocol, and then conclude with a subsection describing differences between the Chord and TerraDir versions.

The routing protocol uses two forms of soft state: caches and replicas. Both operate on the granularity of a single data item. A cache entry consists of a data item label, the item's home, the home's physical address, and a set of known replica locations. Cache entries are replaced using a least recently used (LRU) policy with an entry being touched whenever used in routing. Caches are populated by loading the path "so far" into the cache of each server encountered during the routing process. Both the source and destination cache the entire path. This form of path propagation not only brings remote items into the cache, it also brings in nearby items and a cross-section of items from different levels of the namespace tree. Our experience is that this mixture of close and far items performs significantly better than caching only the query endpoints.

Replicas differ in that (i) they contain the item data, (ii) they contain a great deal more state, and (iii) new replicas are advertised (unreliably) throughout the system. In addition to the item data, the state maintained in a replica includes the physical addresses of the item's home, the neighbors of the home, and any other known replicas.

For purposes of this paper, replication is primarily a tool with which to balance load. Overloaded servers attempt to shed load by creating replicas of their most popular items at other servers. This task can be decomposed into a long list of subtasks:

1) *How to adapt* - What is the mechanism used to redistribute load? How many replicas are created elsewhere? How many items hosted locally have new replicas created? Do the local replicas survive?
2) *When to trigger* - The system must redistribute load relatively quickly in order to handle dynamic query streams. However, reacting too quickly could lead the system to thrash.
3) *Where to place replicas* - The local server has only imprecise, and possibly stale, information about other servers, and in general does not ever know the full membership of the system.
4) *How to choose replicas during routing* - Assume a server has knowledge of a set of replicas for a desired "next hop" in the routing process. Which of the replicas is chosen? Should the selection process attempt to incorporate knowledge of load at replica locations?
5) *How to merge replica sets* - Our decision to allow remote sites to independently create and destroy replicas means that the number of system replicas of a given item is not bounded. Whether information about replica sets is disseminated by new messages or appended to existing messages, the total number of replicas of a given item might be too large to include. Hence, the set of replicas included in a message might be a subset of the replicas known to the message's source. How should this subset be chosen? Further, how large of a set should an item's home maintain? Assuming that the latter set is bounded, the protocol must have a procedure for merging incoming replica set information into its local set.

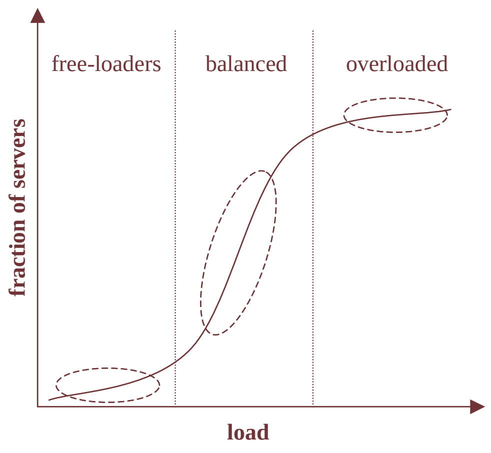

Fig. 1. A CDF of load versus fraction of servers with a uniform query distribution. The majority of servers have acceptable load. We concentrate mainly on alleviating the relative overloading of those in the long high tail, and to a lesser extent on identifying "free-loaders", i.e. servers with little or no load.

6) *How to disseminate replica information* - New replicas are useless unless other servers know of their existence[4]. Hence, information about new replicas must be disseminated. Should such information be pushed eagerly by a separate dissemination sub-protocol, or appended lazily to existing messages?

This set of questions is clearly too large too be answered definitively, so our approach is as follows. Figure 1 shows a cumulative distribution function (CDF) of server loads with a uniform query distribution. The majority of servers are in an acceptable range, but a small subset of server have either very high or very low load (the two tails of the distribution). We will concentrate on moving the relatively few servers in either tail into the "balanced" portion of the load curve.

The system adapts to load irregularities by creating and destroying replicas. Replicas in our system are "soft" in the sense that they can be created and destroyed without any explicit coordination with other replicas or item homes. Hence, idle replicas consume no system resources except memory on the server that hosts them. Therefore, identifying and evicting redundant replicas is not urgent, and can be handled lazily via an LRU replacement scheme. Replica destruction will largely be ignored in the remainder of the paper.

Replica creation, however, is in the critical path. Assume server s_i's most highly requested item is n_x. Load is shed by replicating highly-requested local items on other sites. We do not destroy the local replica. Hence, even if knowledge of the new replica is spread immediately throughout the system, the new load on s_i's replica of n_x will only decrease by a factor of $\frac{1}{R_x+1}$, where R_x is the total number of replicas of n_x in the system prior to new replica creation.

It might seem that the above fraction could be used to calculate exactly how many new replicas are created. However, our desire for local decision-making means that R_x is not known. Further, even completely eliminating the load on n_x might not reduce s_i's overall load to an acceptable level. Finally, we wish to minimize the amount of communication added by the replica mechanism to the system.

Our solution is to shed load by creating new replicas of possibly multiple items hosted by s_i at a single other site. Recall that each server has a defined capacity, and high- and low-water thresholds. All of these metrics are defined in terms of messages sent to (or routed through) a server during a time unit.

Each time a packet is routed through server s_i, s_i checks whether the current load, l_i, indicates that load redistribution is necessary. If necessary, load is redistributed to the source of the message, s_j. The source is

[4]This is not precisely true because routing can be short-circuited whenever a replica is encountered. However, this is a secondary effect.

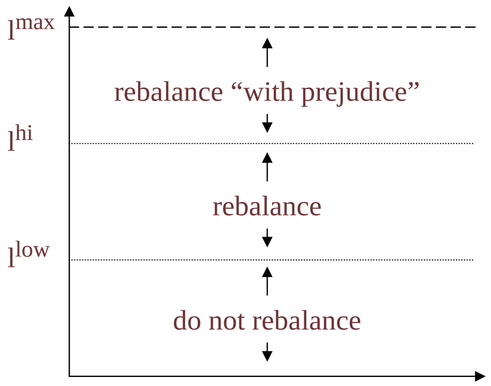

Fig. 2. The server capacity is l^{max}. Load is sometimes re-balanced if greater than l^{low}, and always if greater than l^{hi}.

chosen because it is in some sense "fair" (the source added to the local load), and because load information about the source can easily be added to all queries, allowing downstream servers to have current information about query sources.

Load is redistributed according to a per-node capacity, l_i^{max}, and high- and low-water thresholds, l_i^{hi} and l_i^{low}, as in Figure 2. If $l_i > l_i^{max}$, s_i is overloaded. Assuming that $l_i > l_j$, s_i attempts to create new replicas on s_j s.t. s_i's total local load on the replicated items is greater than or equal to the difference between the loads of the two servers.

Specifically, s_i asks s_j to create replicas of the n most highly loaded items on s_i, such that the sum of the local loads due to these n items is greater than or equal to the difference in loads between the two servers.

If s_i's load is merely high, but not in an overload situation ($l_i^{low} \leq l_i \leq l_i^{hi}$), load is redistributed to s_j only if $l_j \leq l_j^{lo}$. The amount redistributed is calculated as above.

In both cases, there may not be sufficient distinct replicas. Further, replicas are only made if the local load due to an item is non-negligible.

Finally, the protocol needs to disseminate information about replica sets. Rather than introduce extra message traffic, we piggyback replica sets on existing messages containing cache entries. Servers maintain only partial replica set information in order to bound the state required to store and transmit the information. A $2/32$ dissemination policy means that a maximum of two replicas locations are appended to cache insertion messages, while a maximum of 32 replica locations are stored, per data item, at a given server.

The *selection* policy determines which replica is used during routing. We currently choose randomly among all known replicas at each step of the routing process.

The *merge* policy determines how incoming replica locations are merged into the local store of replica locations, assuming both are fully populated. The locations to be retained are currently chosen randomly, as experiments with different preferences did not reveal any advantage.

The *dissemination choice* policy decides which of the locally known replica locations to append to outgoing messages. Random choice works well here as well, but we found a heuristic that slightly improves results. If a server has a local replica and has created others elsewhere, it prefers the replicas it has created elsewhere most recently. Otherwise, the choice is random. The intuition behind this heuristic is that if the existing load is high enough to cause the server to attempt shedding load, it is counter-productive to continuing advertising the server's own replicas. On the other hand, advertising newly created replicas helps to shed load.

Note that we have neglected consistency issues. However, it is highly unlikely that rapidly changing objects will be disseminated with this type of system and we have designed our protocol accordingly. We also do not address servers joining and leaving the system. These actions are handled by the underlying system P2P system and should not affect the applicability of the replication scheme.

To summarize, LAR takes a minimalist approach to replication. Servers periodically compare their load to local maximum and desired loads. High load causes a server to attempt creating a new replica on one of the servers that caused the load (usually the sender of the last message). Since servers append load information to messages that they originate, "downstream" servers have recent information on which to base replication decisions. Information about new replicas is then spread on subsequent messages that contain requests for the same data item.

The replication process only requires a single RPC between the loaded server and a message originator. Further, this RPC contains no data because the originator of a request has already requested it. Even this RPC can be optimized away if the loaded server is also the server that responds to the request. However, we retain it in order to allow the replication process to proceed asynchronously with respect to the lookup protocol.

A. Adapting LAR *to Chord*

The implementation of LAR for Chord differs mainly in that the unit of replication is an entire server, rather than an individual data item. This makes all replication decisions much more heavyweight than in TerraDir, but the procedure is essentially the same. At any point where the local load is higher than even the low threshold, a server replicates itself at the source of the query that initiated the replication event.

V. Simulation Results

Our performance results are based on a heavily modified version of the simulator used in the Chord project, downloaded from `http://www.pdos.lcs.mit.edu/chord/`. The resulting simulator is discrete time and accommodates per-server thresholds and capacities. Each network "hop" takes a single time unit, currently set to 25 milliseconds. Each message contributes identically to loads and congestion.

This section lists the defaults for our experiments. The query distribution is uniform with Poisson arrivals. The average query input rate is 500 queries per second across the entire system. The default "skewed" input is `90-1`, where 90% of the input is skewed to one item, and the remaining 10% randomly distributed. This is a somewhat pathological case for our protocol, and was chosen to show the protocol under extreme stress. Similarly, all query streams arrive as impulses. The system would not lose any messages if input changed slowly.

Simulations run with 1k servers, 32k data items, and $l^{max} = 10/sec$. We ran many experiments with higher capacities, but found no qualitative differences in the results. l^{hi} and l^{low} are set to 0.75 and 0.30 times l^{max}. The length of a server's queue is set to the number of of locally homed items, in this case 32. For example, if an idle server with capacity $l^{max} = 10$ and queue length $q_{max} = 32$ receives 50 queries during one time unit, 8 will be dropped. The default load window size, which controls how quickly the system can adapt, is set to two seconds. The dissemination policy is set to $2/32$.

TerraDir servers each have 20 cache slots, and can accommodate replicas of 64 remote data items (twice as many as the number of items for which the server is home).

Chord has no regular cache slots. Chord servers can accommodate replicas of five other servers, which gives them approximately the same amount of state as consumed by the combination of the TerraDir caching and replication schemes.

Where results are similar, we only show results for the TerraDir model.

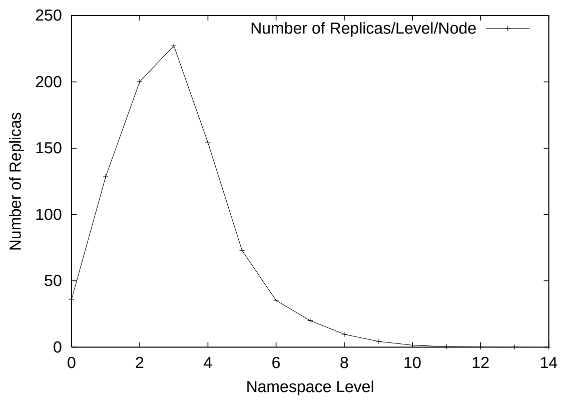

Fig. 3. Average number of replicas created for items at a given level of the TerraDir tree. Level 0 is the root.

A. Static versus Adaptive Replication

This section contrasts the performance of static versus adaptive replication for both Chord and TerraDir. A static distribution for Chord is easily created by replicating all servers evenly.

For TerraDir, calculating the proper static distribution of replicas is non-trivial. Without caches, and assuming that the tree is balanced and that the query distribution is uniform, the load on each item in the tree can easily be calculated analytically. However, caches change the load distribution considerably. Figure 3 shows the number of times each item is replicated in our adaptive scheme, a measure highly correlated with load. We use this result as the “static” replica distribution in the rest of this section.

The explanation for level three having the highest load is as follows. Assume that a cache is initially populated uniformly. Items from high in the tree (level 0 etc.) are quickly evicted, as cached elements slightly lower in the tree are closer to destinations and all fit in the cache. Items from low in the tree are also evicted. They are surely closer to some destinations, but are only touched by a small proportion of the queries. Hence, the caches become populated with items in a middle ground where the whole level fits into the cache, yet each item on the level is used frequently.

Caches populated in this way cause a great deal of load on these middle levels, as cache entries only contain mappings from an item name to a server’s address. They do not contain the data and so can not satisfy queries locally.

Given this distribution, Figures 4 and 5 show message losses versus time for all combinations of static or adaptive replication, and uniform or skewed input. The figures show that adaptivity is crucial in order to handle skewed distributions. Both Chord and TerraDir lose on the order of 90% of messages with static replication and skewed input, but stop losing messages within two and one half minutes when using adaptive replication.

Additionally, the graphs highlight the need of the hierarchical system to populate the system with replicas for items high in the graph. The adaptive experiments start without any replicas in the system, so the “unif. with adapt. rep” line in Figure 5 is showing the process of populating the system with the “best” static distribution discussed above. By contrast, Chord does not need any replication to handle uniform query distributions.

Figure 6 shows the number of message drops in a pathological case of input that changes increasingly rapidly. Each time the focus changes, the system re-populates servers with replicas of the new focus.

B. Load Balancing

LAR has two primary goals: handling overloads and attempting to keep server loads under the low-water

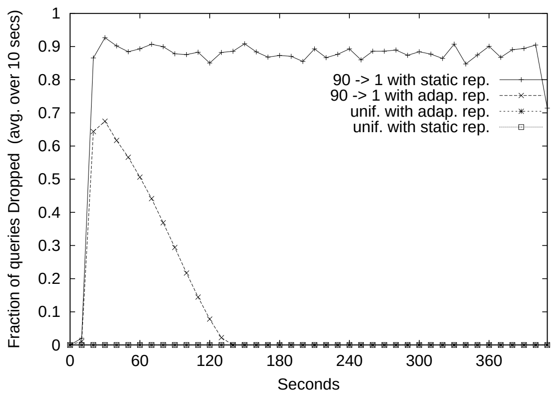

Fig. 4. Message drops versus time for Chord, all combinations of uniform/skewed input and static/adaptive replication.

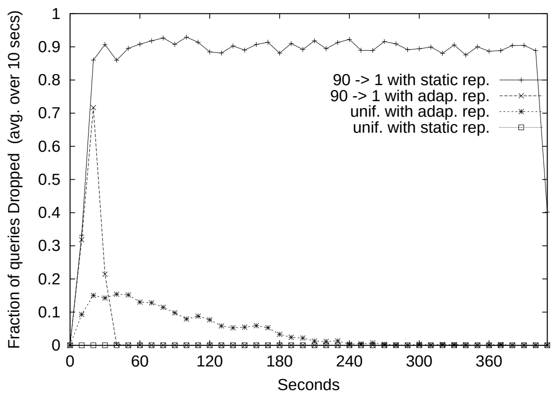

Fig. 5. Message drops versus time for TerraDir, all combinations of uniform/skewed input and static/adaptive replication.

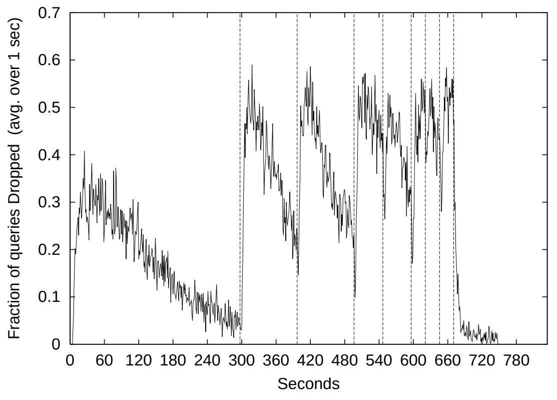

Fig. 6. TerraDir message drops for increasingly rapid changes in the focus of skewed input. Vertical lines indicate when the “hot” data item changes.

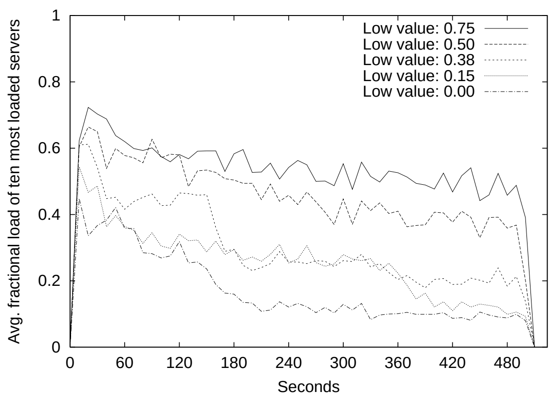

Fig. 7. Average of 10 most loaded servers for TerraDir with skewed input and varying l^{low} values.

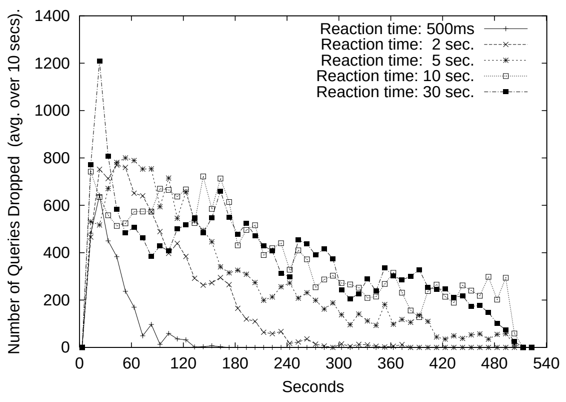

Fig. 8. Drops versus load windows size as a function of time.

threshold, l^{low}, if possible. Figure 7 shows the average of the ten highest server loads for a run with adaptive replication and a skewed distribution on TerraDir. Each line represents a different l^{low} value; in all cases l^{hi} remains at 0.75. In all but the two extreme cases, the averages quickly drop below the low-water threshold. They do not drop further because servers whose loads are less than l^{low} do not attempt to shed load. The slope of the $l^{low} = 0.15$ and $l^{low} = 0.00$ lines flattens considerably as they near 0.10, the mean load in the system.

The overall load in the system varies from 10% with l^{low} at 0.75, to 5.5% with a l^{low} value of 0. The variation is due to increased queuing times when loads are highly loaded.

C. Parameter Sensitivity Study

Load Window Size - Figure 8 shows plots of message drops versus time for TerraDir with adaptive replication and different load window sizes. Smaller windows allow the system to react more quickly, adapting better to swiftly changing conditions at the cost of more replica creations and evictions.

The cost of this adaptivity is shown in Figure 9. One possible conclusion is that the system is relatively insensitive to the size of the load window within a broad range (10-30 seconds). The use of smaller windows can dramatically improve drop rates, but only at the cost of increased protocol traffic.

Dissemination Constants - Table I shows the effect of the dissemination policy on drops and replica events for TerraDir and skewed input. Recall that `x/y` means that x replica locations are appended to outgoing messages about a given item and y are stored locally. Skewed input heavily overloads a single server

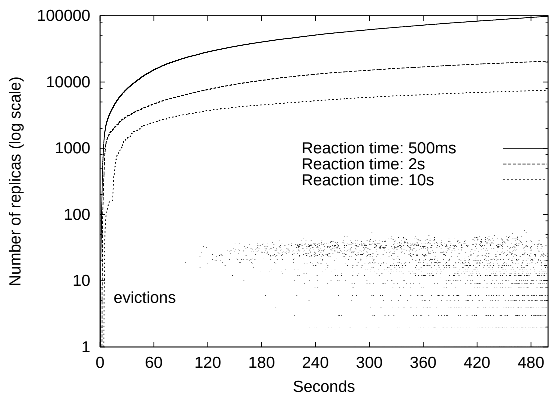

Fig. 9. Cumulative replica creations versus time for a `90-1` distribution under TerraDir, for three different load window sizes. Dots indicate eviction events for a window size of 500 msecs. Larger windows sizes do not cause evictions. Note the log scale.

Policy	Drops	Replicas	Evictions.
0/1	162172	4803	0
1/1	19595	2634	0
1/8	19275	2582	0
2/8	16942	2493	0
2/32	17759	2462	0
16/32	17407	2493	0
32/32	17407	2503	0
64/128	17648	2447	0

TABLE I

DROPS AND REPLICA EVENTS VERSUS DISSEMINATION POLICY.

and places a premium on quickly spreading knowledge of new replicas. Nonetheless, the results show an almost complete lack of sensitivity to these parameters. Though we use `2/32` for the other experiments in this paper, the results show that keeping and propagating only a single other storage location is almost as effective.

D. Scalability

Figure 10 shows that the adaptive replication scheme easily scales through an order of magnitude difference in the size of the system. The input distribution is `90-1`, and the input size is scaled with the size of the system in all cases. The peak in message drops scales only linearly with the size of the system. Message drops are virtually eliminated in a single minute, regardless of system size. This argues well for scaling to larger systems.

E. Caching in CFS and PAST

Neither Pastry nor Chord has any mechanism to deal with non-uniform query distributions. Instead, both PAST [5] and CFS [6], distributed file sharing applications that run on top of Pastry and Chord, respectively, implement their own distributed caching scheme. We will refer to our generalization of the approach used in these applications as `app-cache` in the following.

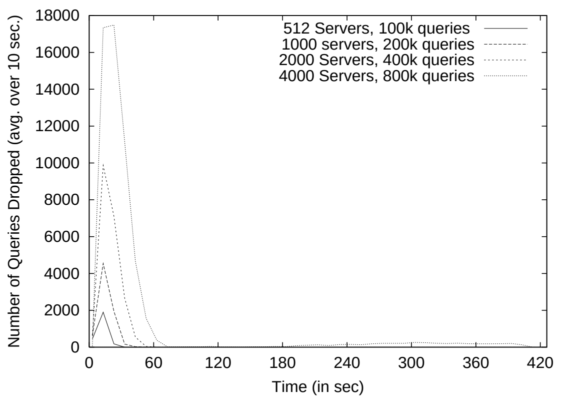

Fig. 10. Message drops versus time for different TerraDir system sizes with skewed input.

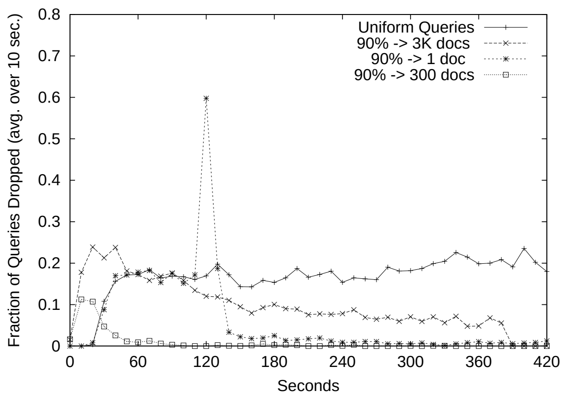

Fig. 11. Drops versus time for `app-cache` with different amounts of query skew.

`app-cache` creates a constant number of replicas "near" each file. Although other sites do not know the precise location of the replicas, they have a constructive algorithm for finding one, i.e. route to near the primary and query the neighbors. However, the main purpose of these replicas is to achieve fault tolerance rather than to improve performance, and they have little effect on load distribution under heavily skewed loads.

`app-cache` deals with skewed loads through the use of caches. Copies of a requested file are placed in the caches of all servers traversed as a query is routed from the source to whichever server finally replies with the file. In a virgin state, this responding server will be the file's home. However, subsequent queries for the file may hit cached copies because the neighborhood of the home becomes increasingly populated with cached copies.

As a result, the system responds quickly to sudden changes in item popularity. `app-cache` is very pro-active in that it distributes $k - 1$ cached copies of every single query target, where k is the average hop count. By contrast, LAR is quite conservative. A server checks its local load after processing each message, but only decides to redistribute load (create replicas) if the local load is above a high threshold. Even then, replicas are only created at one other server.

Figures 11 and 12 show dropped messages versus time for several different input stream distributions. For `app-cache`, each server has 32 cache slots. Since each server also exports 32 items, this means that fully half of a server's storage is devoted to cache space. The distributions range from `90-1`) to uniform.

At the extreme end, `90-1` is the best possible case for `app-cache`. After a warm-up period, the one hot

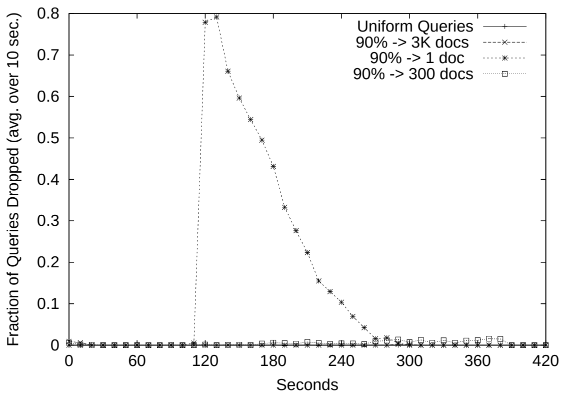

Fig. 12. Drops versus time for LAR with different amounts of query skew.

item is cached on almost all servers in the system. Nevertheless, LAR is still within a factor of two for total losses.

As the hot spot grows larger, `app-cache`'s document caches become less useful while still incurring the same overhead. Recall that the caching performed after a query is served requires revisiting every server on the path, effectively doubling the number of network messages that are sent.

On the other hand, LAR distributes replicas only when an imbalance is detected. For the extreme `90-1` case, replicas are not created quickly enough to avoid heavy losses (although the total number of messages lost is only twice that of `app-cache`). The losses arise because LAR servers cache item-to-server mappings, meaning that later queries often jump straight to the destination. The destination is quickly overloaded and starts dropping messages. `app-cache` fares better because queries being routed towards the destination are satisfied by copies cached by the destination's neighbors. The original destination is quickly shielded from further load. Nevertheless, LAR stops dropping messages within three minutes. This adaption time could be improved by changing the load window size, currently two seconds.

LAR's performance quickly improves as the size of the hot spot increases. By the time the hot spot grows to encompass the entire data set (the uniform input), there are no longer any major load imbalances among servers, and few replicas are created. The number of messages dropped is very low for all but the degenerate case.

The result is that, under LAR, servers are operating at about 23% average capacity for the uniform input. The average query latency is 4.7. By contrast, the average load for `app-cache` is about 58% capacity. The nominal average query latency is 5.7, but this latency is doubled by the caching scheme. Random fluctuations due to the Poisson input stream cause local bottlenecks and a steady state where messages are dropped.

The story for `app-cache` is actually worse than these graphs indicate. For the uniform case, only $1/k$ of the messages carry data for LAR, whereas half the messages carry data for `app-cache`. The size differential between data messages and control messages could vary. For example, CFS serves file blocks while PAST serves whole files. However, it is clear that data messages can easily be an order of magnitude or more larger than control messages, and `app-cache` sends many more than LAR. The difference in drop rates between the two protocols could be much larger if the type and size of messages is accounted for.

The results show the impact of the differing approaches exemplified by the two protocols. `app-cache` scatters cache entries, each containing the entire data item and requiring an RPC, to every server on the routing path for every query served by the system. By contrast, LAR creates replicas at a single site, and only after many queries have been served. Despite this disparity, LAR balances load more effectively.

VI. Summary and Conclusions

In this paper, we have described a new soft-state replication scheme, LAR for peer-to-peer networks. LAR is a replication framework which can be used in conjunction with almost any distributed data access scheme. In this paper, we have applied LAR to a distributed hash-table algorithm (Chord), and to a hierarchical data access protocol (TerraDir).

Compared to previous work, LAR has an order of magnitude lower overhead, and at least comparable performance. More importantly, LAR is adaptive: it can efficiently track changes in the query stream and autonomously organize system resources to best meet current demands.

We have demonstrated the efficacy of LAR using a number of different experiments, all conducted over a detailed packet-level simulation framework. In our experiments, we show that LAR can adapt to several orders of magnitude changes in demand over a few minutes, and can be configured to balance the load of peer-servers within configurable bounds.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content addressable network," in *In Proceedings of the ACM SIGCOMM 2001 Technical Conference*, 2001.

[3] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

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

[5] Antony Rowstron and Peter Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[7] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel, "SCRIBE: The design of a large-scale event notification infrastructure," in *Networked Group Communication*, 2001, pp. 30–43.

[8] Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher, "Query routing in the TerraDir distributed directory," in *Proceedings of SPIE ITCOM'02*, Boston, MA, August 2002.

[9] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana 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, December 1996.

[10] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proceedings of the INFOCOM '99 conference*, March 1999.

[11] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, May 1997, pp. 654–663.

[12] "Gnutella home page," http://gnutella.wego.com.

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

[14] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th ACM International Conference on Supercomputing*, New York, USA, June 2002.

[15] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *The ACM SIGCOMM'02 Conference*, August 2002.

[16] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," 2000.

[17] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 311–320.

[18] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang, "An adaptive data replication algorithm," *ACM Transactions on Database Systems*, vol. 22, no. 2, pp. 255–314, 1997.

[19] 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)*, San Jose, CA, USA, January 2002.

[20] Pete Keleher, Samrat Bhattacharjee, and Bujor Silaghi, “Are virtualized overlay networks too much of a good thing?,” in *The 1st International Workshop on Peer-to-Peer Systems (IPTPS’02)*, 2002.