

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

Peer-to-peer systems can be used to form a low-latency decentralized data delivery system. Structured peer-to-peer systems provide both low latency and excellent load balance with uniform query and data distributions. Under the more common skewed access distributions, however, individual nodes are easily overloaded, resulting in poor global performance and lost messages.

This paper describes a lightweight, adaptive, and system-neutral replication protocol, called LAR, that maintains low access latencies and good load balance even under highly skewed demand. We apply LAR to Chord and show that it has lower overhead and better performance than existing replication strategies.

1. Introduction

Peer-to-peer systems can be used to form a low-latency decentralized data delivery system. Structured P2P systems provide both low latency and excellent load balance with query streams in which all data items are accessed with uniform probability. However, the distribution of demand for real data items is often skewed, leading to poor load balancing and dropped messages. In this paper, we describe and characterize 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 [22, 10], multimedia streaming, event notification [23], 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 participants, 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. A number of efficient algorithms [26, 18, 14, 21, 20, 4, 15, 13] provide low average query latency, and we do not consider it here further.

Distributing load equitably, however, is more difficult. For example, many recent systems [26, 28, 20, 21] attempt to balance load by using cryptographic hashes to randomize the mapping between data item names and locations. Under an assumption of uniform demand 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 these systems 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 [4], as the system topology is inherently non-uniform, resulting in uneven routing load across servers.

This problem has so far been addressed (e.g. PAST [22], CFS [10]) in an end-to-end manner by caching at higher levels. Specifically, data is cached on all nodes on the path from the query destination back to the query source. Routing is not affected and “hot” items are quickly replicated throughout the network. 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 extremely skewed query distributions, they perform poorly under even moderate load because of their high overhead.

We describe a lightweight approach to adaptive replication that does not have these drawbacks. Instead of creating replicas on all nodes on a source-destination path, we rely on server load measurements to precisely choose replication points. Our approach can potentially create replicas for an object on any node in the system, regardless of

whether the original routing protocol would ever direct a query to the replica hosts. We augment the routing process with lightweight “hints” that effectively shortcut the original routing and direct queries towards new replicas (described in Section 3). This protocol incurs much lower overhead, can balance load at fine granularities, accommodates servers with differing capacities, and is relatively independent of the underlying P2P structure.

The main contribution of this paper is to show that a minimalist approach to replication design is workable, and highly functional. We derive 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 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 2 summarizes the related work. Section 3 describes our model and design goals. Section 3.2 describes the protocol in more detail. Finally, Section 4 presents our simulation results and Section 5 summarizes our findings and concludes the paper.

2. Background

This section briefly summarizes related work. We use Chord as the representative P2P system protocol in the simulations described in Section 4.

2.1. Related work

Chord [26], CAN [20], Pastry [21] and Tapestry [28] are prototypical hash-based peer-to-peer systems. There are a number of other recent efforts in this area as well including Viceroy [18], Skipnet [14], Koorde [15], Kelips [13], etc. which provide similar or better latency bounds.

All of these systems 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. In this paper, we use Chord as the example DHT, and apply LAR to a Chord network. The other systems differ from Chord in important ways, but none of them should affect the applicability of our approach.

All the hash-based systems perform well when there is uniformity in query distribution. However studies [2, 5] 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 [16] have been motivated by the need to balance the load and relieve hot-spots on the World-Wide-Web. Similar Zipf-like patterns were found in traces collected from Gnutella [11], one of the most widely deployed P2P systems. Caching the results of popular Gnutella queries for a short period of time proves to be effective in this case [25]. Our path propagation is a generalization of this caching scheme. Stavrou et.al [3] propose the use of a P2P overlay to handle hotspots in the Internet. Ours is different in that we are trying to handle hotspots in the P2P network.

Recent work [17, 9] 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. Freenet [8] 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.

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 [27]. Concurrency control mechanisms need to be specified for data replication to guarantee replica consistency.

A recent analysis [24] 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.

2.2. Existing approaches to Adaptive Load Balancing

Existing load balancing solutions (and products) such as the Cisco Local/Global director [7] or even techniques used in content distribution networks such as Akamai [1] are simply not applicable in our context. This is because these systems require too much coordination and coupling between nodes and often require a centralized coordination point where global knowledge is available.

DHTs (such as Pastry or Chord) do not have any in-built mechanism to deal with non-uniform query distributions. Instead, distributed file sharing applications, like PAST [22] and CFS [10], implement their own distributed replication scheme. Hotspots and dynamic streams are handled by using caches which are used to store popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. CFS [10] for instance replicates data in k of its successive neighbors for data availability, and populates all the caches on the query path with the destination data af-

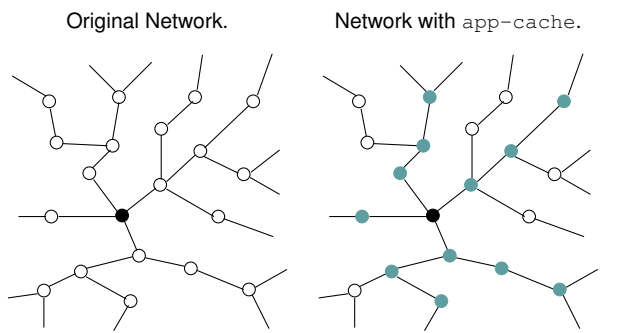


Figure 1. A comparison of the original network and a network with `app-cache`. The darkened node is the “home node” while the shaded nodes are nodes caching the data.

ter the lookup completes. We will refer to our generalization of the approach used in these applications as `app-cache` in the following.

`app-cache` deals with skewed loads through the use of caches. In a virgin state, this responding server will be the file’s home. Copies of the requested file are then placed in the caches of all servers traversed as the query is routed from the source to whichever server finally replies with the file. 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.

Figure 1 depicts a simple network where the darkened dot represents the “home” server. An edge indicate that the server knows about the existence of the other end. Figure 1 also shows how `app-cache` handles dynamic query streams by caching data (represented in shaded dots) in the path of the query. This leads to the neighbors caching copies of the data item and then its neighbors and so on.

3. The LAR Protocol

In this section we describe the LAR protocol. We start off by describing the our goals when we designed the protocol. We also talk about the approaches we adopt to achieve the goals. The we discuss the various aspects of the protocol in detail. Finally, as an example, we describe how we adopt the protocol to Chord.

3.1. Protocol goals and approach

The design of the replication protocol has been motivated by a couple of goals. Firstly, we wish to address overload

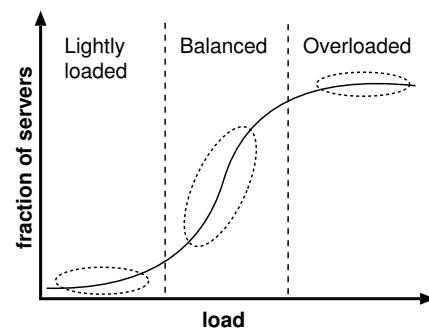


Figure 2. A CDF of load versus fraction of servers with a uniform query distribution. The majority of servers have *acceptable* load. We concentrate mainly on alleviating the relative overloading of those in the long high tail, and to a lesser extent on identifying lightly-loaded servers.

conditions, which are common during flash crowds or if a server hosts a “hot” object. Therefore the goal of the replication protocol is to distribute load over replicas such that requests for hot objects or flash crowds can be handled. The key to achieving this is to employ Adaptive protocols. Adaptive protocols can cope efficiently with dynamic query streams, or even static streams that differ from expected input.

Second, we will attempt to balance load. Figure 2 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 concentrate on moving the relatively few servers in either tail into the “balanced” portion of the load curve. Obviously, the sensitivity, overhead, and effectiveness of the algorithm will depend on exactly how *acceptable* is defined and what mechanisms are used to shed load. Our approach to achieving the above goals is to use load-based replication of data and routing hints and to augment the existing routing mechanism to use the replicas and routing hints.

Our third goal is to base all decisions on locally available information. 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 (as reported by the client). 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 trade-off 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 its implications. For one, local decisions might be poor if locally available information is unrepresentative of the rest of the system. Also, 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.

Finally, our protocol is intended to be independent of P2P structure. We intend our results to be applicable to DHTs [26, 28, 20, 21, 18, 14, 15, 13]; to hierarchical namespaces [4]; and also to unstructured P2P systems like Gnutella [11].

3.2. Protocol description

This section describes the policies that LAR uses for load re-distribution, replica and cache entry creation/deletion, and replica-augmented routing. There are three specific issues that must be addressed:

1) *Load measurement and replica creation*: The system must redistribute load relatively quickly in order to handle dynamic query streams. However, reacting too quickly could lead the system to thrash. We need to specify *when* new replicas are created, on what nodes, and which items a server replicates, and how replicas are discarded.

2) *Routing using cache hints and replicas*: Assume a server has knowledge of a set of replicas for a desired “next hop” in the routing process. The overlay routing algorithm must be augmented such that these replicas are visited instead of only the home node of an item. We need to specify which of the replicas to choose during routing, and (how) should the selection process attempt to incorporate knowledge of load at replica locations.

3) *Replica information dissemination and management*: New replicas are useless unless other servers know of their existence¹. Information about new replicas must be disseminated, whether eagerly by a separate dissemination sub-protocol, or lazily by being appended to existing messages. Allowing remote sites to independently create and destroy replicas means that the number of system replicas of a given item is not bounded. The dissemination policy must determine the amount of replica pointer state that should be kept at each site, the way that new replica pointer information is

¹ This is not precisely true because routing can be short-circuited whenever a replica is encountered. However, this is a secondary effect.

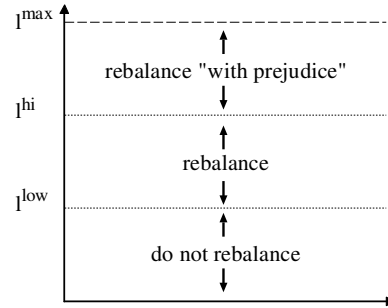


Figure 3. The server capacity is l^{max} . Load is sometimes re-balanced if greater than l^{low} , and always if greater than l^{hi} .

merged with older information, and what state should be appended to outgoing messages (or pushed eagerly).

In the rest of this section, we specify, in detail, how LAR addresses each of these issues.

3.2.1. Load Measurement and Replica Creation

Local Load Measures Our replication scheme is different from the existing schemes like [10] in that we introduce the construct of load in order to perform replication. We assume that each server has a locally configured resource capacity and queue length. We also assume that each server defines a high-load and low-load threshold. By default, the system sets high-load and low-load thresholds for each server based on fractions of servers’ capacities. We also assume that we can keep track of the load due to each object in the server; data and routing indices.

For this work the capacity indicates the number of queries that can be routed or handled per second, and the queue length 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 load metric in the protocol is abstract and in practice can be defined based on any of the factors like CPU load, network load, I/O load or even a combination of these factors. For example, Rabinovich et.al [19] make use of the ready queue, such as the output of the “uptime” command as a measure of computational load. Considering that most P2P clients like KaZaA and Gnutella allow the user to set the maximum upload and the download bandwidth, the application could keep track of the number of bits transferred or received to estimate the network load. Chawathe et.al. [6] suggest that the capacity should be a function of the server’s processing power, access bandwidth, disk speed etc.

As Figure 3 shows, high-load threshold indicates that a server is approaching capacity and should shed load to prevent itself from reaching its maximum capacity and drop requests. The intent of the low threshold is to attempt to bring the load of two servers closer to each other. Intuitively, the

difference in the load of two servers being greater than the low threshold indicates that one of the servers is much less loaded compared to the other and that it can be used to distribute the load. We will use the term “load balance” in the rest of this paper to refer to this sense of distributing load.

Lastly, we assume that the fractions for thresholds are constant across the system in the simulations here, but the protocol will work unaltered with non-uniform fractions. Also note that it is relatively straightforward to incorporate load measures with multiple thresholds into the protocol, or indeed to use completely different load measures. In this paper, we show that the simple two threshold scheme is both robust and efficient.

Replica creation detail Load is redistributed according to a per-node capacity, l_i^{max} , and high- and low-load thresholds, l_i^{hi} and l_i^{low} , as in Figure 3. 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 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. Lastly, creating a replica at the source is often “cheap”, since the source is likely transferring a popular object (which would be replicated).

If $l_i > l_i^{hi}$, S_i is overloaded. S_i attempts to create new replicas on S_j if l_i is greater than l_j by some fixed value K . S_i then 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 $l_i - l_j \geq l_i^{low}$. 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.

3.2.2. Soft State replicas and Replica-augmented routing We use 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. Note that a cache entry does *not* contain the item’s data: it is merely a routing hint that specifies where the data item can be found.

Cache entries are replaced using a Least-Recently-Used (LRU) policy with an entry being touched whenever it is used for 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 in nearby items but also a cross-section of remote items from different regions of the namespace. Our experience is that this mixture of close and far items performs significantly better than caching only the query endpoints. These cached entries effectively “short-cut” routing when encountered by queries. Our results show that adding the cache entries for routing signif-

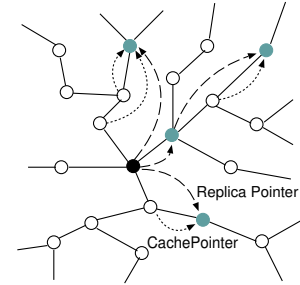


Figure 4. Network with LAR

icantly improves system performance and load balance regardless of input query distribution.

Replicas differ in that (i) they contain the item data, and (ii) new replicas are advertised on the query path. When a replica is created, we install cache state on the path from the new replica to the node that created the replica. Figure 4 shows how LAR creates replicas on the source of the query and adds pointers to the replica in caches along the path in the same network as Figure 1. Also note that a replica can further create replicas as shown in the figure. In this case, pointers are added in the path from the new replica to the original replica only. Also contrast the difference with app-cache protocol in Figure 1.

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. Obviously, cache entries may point to stale replicas since there is no global coordination on when replicas are created or destroyed.

Since cache state includes information about multiple replicas, and during routing, we can choose one of these replicas uniformly at random. Obviously, cache entries can be used to distribute load among the servers that hold replicas of the data item being queried. However, they can also be used to find replicas of *next hop* nodes that are used to route queries (as in Figure 5). Thus, cache entries balance both data transfer load and routing load.

3.2.3. Replica-state Management and Dissemination Once replicas are created, we need to disseminate information about new 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 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 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 policy decides which of the locally known replica locations to append to outgoing messages. Random choice works well here also, 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.

3.2.4. Summary 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 creation of a new replica 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.

In implementation, 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.

3.3. LAR applied to Chord

In this paper, we present results for LAR implemented over Chord; analogous results for a hierarchical namespace (TeraDir) are available in a technical report [12].

When adapting LAR to Chord, the finger list is the default item of replication. We replicate the data item only if the load on the server due to the data item is more than that due to the finger list. However, when we replicate the data item, we also must replicate the finger list in order for the new replica to be seen by other servers.

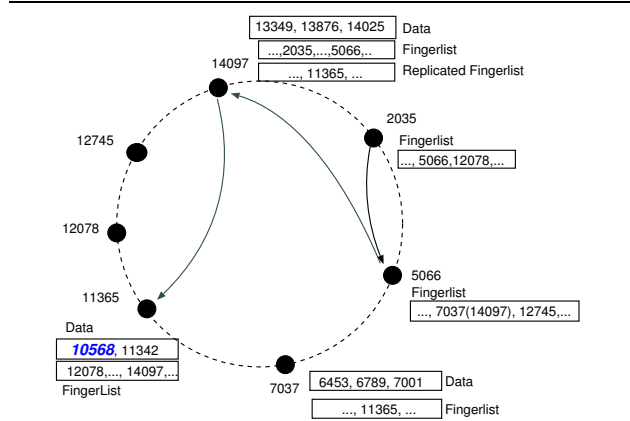


Figure 5. LAR routing and replication in Chord

When disseminating this information, we only know the query’s source and the last sender. We therefore update the cache with this information rather than the full path.

Figure 5 shows an example using chord for a query of item ID 10568, initiated at server 2035. Recall that IDs increase in clockwise direction, and that an item is served by its first successor server. Each server is annotated with its data and fingerlist. In the example, item 10568 is served by server 11365. Server 7037’s fingerlist is replicated at 14097, and this replication is known to server 5066.

Server 2035 chooses 5066 as the next hop because it has the highest ID about which 2035 knows, such that the ID is less than or equal to the item ID. 5066 determines that 7037 is next, but randomly picks 7037’s replica on 14097 instead. Finally, 14097 forwards to 11365, the final destination.

4. Simulation Results

In this section, we present a comprehensive simulation-based evaluation of LAR, and compare its performance to *app-cache*. 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.

Simulation Defaults By default, simulations run with 1k servers, 32767 data items, and the server capacity l^{max} is set to 10 per second. We ran many experiments with higher capacities, but found no qualitative differences in the results. The load thresholds 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 locally homed items, in this case 32. For example, if an idle server with capacity $l^{max} = 10/second$ and queue length $q_{max} = 32$ receives 50 queries over one second, 8 will be dropped (10 will be processed, and 32 will be queued). The default load window size, which controls how quickly

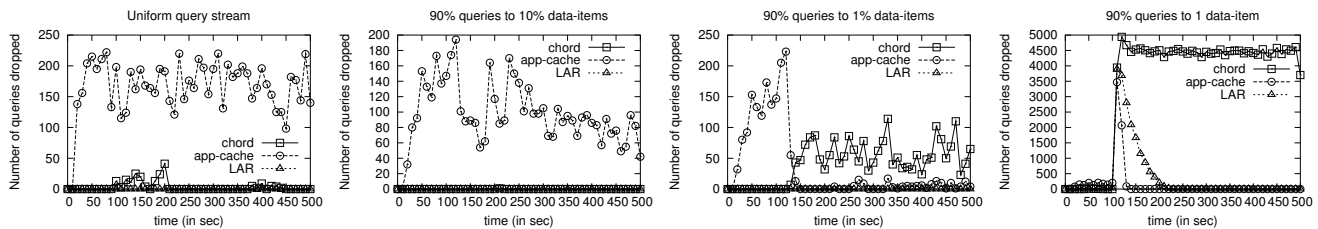


Figure 6. Number of queries dropped over time for different query locality.

the system can adapt, is set to two seconds. Each network “hop” takes 25 milliseconds. The dissemination policy is set to $1/32$. By default, 500 queries are generated per second. The average query path is less than 5 hops, so these default values correspond to an average node load less than 25%.

In the simulations, query sources were selected uniformly at random, and the query inter-arrival had a Poisson distribution. The input query distributions ranged from uniform to heavily skewed. We experimented with extremely heavy skew 90–1, in which 90% of the input is directed to a single item (and the rest 10% uniformly distributed over all items). We also experimented with less skew in which 90% of the inputs were directed to 1% (327) or 10% (3276) items.

By default, each message transfer (whether it is a document, a query or a control message) contributes identically to load and congestion. We chose this default to heavily favor `app-cache`, which creates and transfers many more document replicas. In Section 4.2, we show the effect of document transfers costing 2x, 4x, and 10x over control message transfers. In practice, this cost is likely to be 100 or 1000 times more, so even these values are biased positively towards `app-cache`. Lastly, we note that in the simulations, all messages, including control messages, are dropped when a server is beyond its capacity.

4.1. Effect of Query Distribution

In Figure 6, we show the effect of input distribution on LAR, `app-cache`, and plain Chord. In each figure, we plot the number of dropped messages over time (added over 10 seconds) for the three different schemes. For each experiment, we ran 10 trials with different random number seeds, and these results are from a single representative run. In all experiments, the first 100 seconds of input are uniform, and then the specific input distribution takes effect.

In these results, `app-cache` performs better with skewed query distributions because the hot items quickly get replicated at essentially all nodes in the network. This is because there is no penalty for extra data transfer (since document transfer costs the same as control messages). Note that with even moderate average load (25%), `app-cache` drops messages with a uniform query distribution: this is because the blind replication scheme starts to

Input dist.	Scheme	# q served (250K max)	# replicas		# hints	
			creat.	evict.	creat.	evict.
Unif.	Chord	249.9K	-	-	-	-
	AC	242.4K	1.13M	1.09M	-	-
	LAR	249.9K	5K	0	10.8K	5.9K
90% → 10%	Chord	249.9K	-	-	-	-
	AC	245.7K	994K	962K	-	-
	LAR	249.9K	6.6K	0	12.5K	7.5K
90% → 1%	Chord	248.2K	-	-	-	-
	AC	248.1K	691K	660K	-	-
	LAR	249.9K	10.3K	0	17.3K	12.3K
90% → 1	Chord	72.1K	-	-	-	-
	AC	244.1K	328K	296K	-	-
	LAR	233.4K	2.6K	0	7.2K	2.4K

Table 1. Protocol Overhead of Chord, LAR, and `app-cache(AC)`.

thrash. As we show in later results, this causes severe problems with higher load and higher costs for document transfer.

We should note that plain Chord, while better than `app-cache` for inputs without significant skew, serves only about 10% of input queries for heavily skewed inputs (Figure 6). Note that the y -axis of this figure is significantly higher than the three other figures. Also notice that `app-cache` drops queries for the first 100 seconds of all the runs since the input stream is uniform. When the input shows heavy bias (starting at time=100seconds), `app-cache` stops dropping packets as the hot item is quickly replicated at all servers. At the onset of skew, LAR initially drops a number of queries, but the LAR adaptation reduces drops down to zero as the hot item it replicated and the routing state set up.

4.1.1. Protocol Overheads In Table 1 we show the average overhead of LAR compared to `app-cache`. These results are averages of ten runs. First, note that plain Chord does not create replicas or use routing hints, but loses significant numbers of queries when the input distribution is skewed. For skewed inputs, LAR and `app-cache` both serve over 93% of all queries. With 250K queries, `app-cache` is able to serve more queries for extremely skewed inputs (because the hot item is quickly cached everywhere in the system), but it

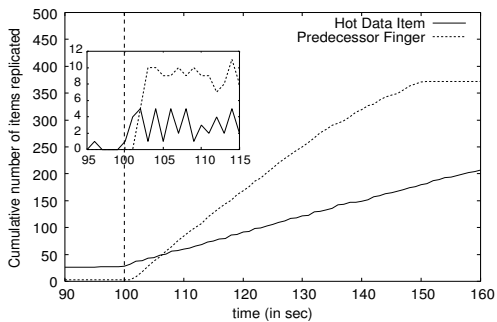


Figure 7. Cumulative number of replicas over time for the hot item and the predecessor’s routing hints, when skewed query distribution begins.

should be clear from Figure 6 that LAR asymptotically approaches 100% service after the adaption takes effect. For inputs with less skew, LAR and even plain Chord outperforms *app-cache*.

The major difference in protocol performance is seen in the replica creation: LAR creates anywhere between 1–3 orders of magnitude less replicas. For example, for uniform queries, *app-cache* creates over a *million* replicas and promptly deletes them! The perils of blind replication are clear; it is relatively easy for *app-cache* to thrash even under moderate load. The lower number of replicas created by LAR directly translates to lower overhead and lower bandwidth usage, since replica creation involves transfer of the document itself. We should note that in LAR, replica creation, in the majority of cases, does *not* involve transferring the document since the source of the query becomes the new replica.

Lastly, the proactive state installed by LAR (the cached routing hints) are orders of magnitude smaller in size than the documents transferred by *app-cache*; moreover, in all cases, the number of hints placed by LAR is 1–2 orders of magnitude lower than the number of replicas created by *app-cache*, and in general, is negligible compared to the number of queries served. Thus, LAR has extremely low overhead, and its judicious replication is stable over a wide range of input distributions.

4.1.2. Dynamics of Replication in Chord Consider the skewed input case. Intuitively, it seems that the successor of the hot data item (which holds the data) would be the one dropping all the queries because of the deluge of queries. However, lookup requests in Chord are routed to the best locally known *predecessor* until the lookup request reaches the actual predecessor of the data item. The query is then resolved and sent to the successor of this node, which is also the successor of the data item. The implication of this is that the first bottleneck in case of skewed inputs is the predecessor of the data item. All the lookup requests need to go to the predecessor before they can be resolved to the successor.

In Figure 7, we show the dynamics of replication of LAR over Chord for a skewed input (90% to 1). The plot shows the cumulative number of replicas created for the hot data item and the number of replicas created for its predecessor between simulation time 90 and 160 secs. In the inset, we show the actual number of replicas created between the time period of 95 seconds and 115 seconds for both the hot item and its predecessor finger. Recall that the input distribution changes from uniform to skew at simulation time 100 seconds (noted by vertical line in the plot). In these plots, the replica creation numbers are computed from the simulation logs once every second, and the servers themselves recompute load once every two seconds. Both from the CDF and the inset plots, it is clear that the predecessor finger gets replicated at a quicker rate than the data item itself is replicated. The routing hints stop replicating at time 150 seconds, while the data is replicated until time 240 seconds (not shown in plot). This is a somewhat non-intuitive phenomenon, and is a direct result of the specifics of how Chord resolves its queries.

4.2. Change in Transfer Costs

scheme	Cost of data transfer vs. control traffic			
	1x	2x	4x	10x
<i>app-cache</i>	4.0K	17.2K	45.5K	106K
LAR	0	0	2	25

Table 2. Transfer Cost: # of queries dropped (250K queries max.)

Table 2 shows the number of dropped queries when the cost of transferring a document increases. In these experiments, 90% of the queries went to 3276 items (10% of the input), and the entire experiment ran for 250K queries. Also, the server capacity l^{max} for these experiments is 20 per sec. The average load on any node in the system is approximately 25%. Since *app-cache* creates an order of magnitude more replicas (and hence transfers correspondingly more data), as document transfer costs increase, it drops close to 50% of the queries in the system. In contrast, LAR is essentially unaffected by these document transfer costs. As mentioned earlier, these experiments are still biased in favor of *app-cache*, and in practice, the document transfer cost is likely to be hundreds of times (or even thousands) more.

4.3. Change in Average System Load

In Table 3, we show how LAR and *app-cache* react when the average system load changes. For these experiments, 90% of the input queries went to 3276 items (10% of the original documents). The figure shows number of unanswered queries over time for average load values of 10%, 20%, 33%, and 50%. With a 90-10 input distribution, LAR

scheme	Average load on server			
	10%	25%	33%	50%
app-cache	0	4K	15.1K	42.7K
LAR	0	0	56	5.4K

Table 3. # of queries dropped (250K queries max) under different loads

is able to serve more than 97% of all queries even with 50% average load. The app-cache scheme is more sensitive to system load and loses about 20% of the input queries when the average system load is 50%.

4.4. Changes in data popularity.

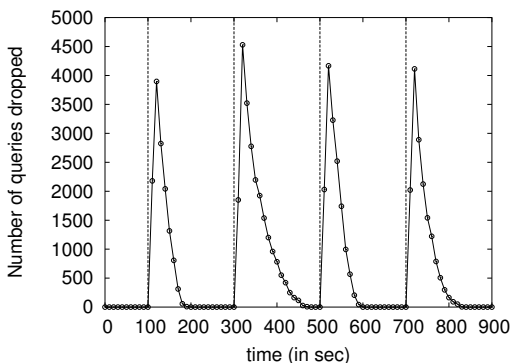


Figure 8. Worst case hotspot adaptivity: Number of queries dropped over 900 sec.

Figure 8 shows how LAR reacts to changes in hotspots over 900 seconds. For these experiments, 90% of the queries went to a single item. There is a change in the hot item every 200 seconds (100K queries) and there are 4 such changes. The first 100 seconds having a uniform query distribution. The vertical lines show time points when the hot item changes. The drops are computed every 10 secs. As is shown in Table 1, this scenario is the worst possible case for LAR. From the plot, we see that LAR is able to adjust relatively quickly to these changes (on the order of 2 minutes), and in all cases the replication is adapted to the change in hot data item. Obviously, in practice, we do not expect such radical shifts in data access patterns to occur over such small intervals, but it is clear that LAR is robust against drastic changes in input distribution over very short timescales.

4.5. Scalability

Figure 9 shows the fraction of dropped queries with different system sizes. In these experiments, 90% of the queries

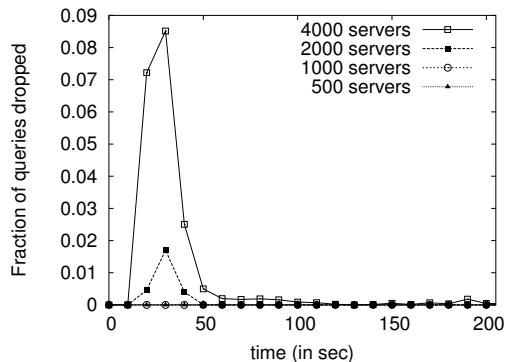


Figure 9. Scalability: fraction of queries dropped for various system sizes.

went to 3276 items (10% of the input). Although the graph is plotted until 205 seconds, the experiments ran for 400 seconds, with no drops at any size after 205 seconds. Note that there are no uniformly distributed queries in the beginning. At each system size, the query stream is adjusted so that the average load on any server is approximately 25%. Since the number of data items are the same for all system sizes and the query rate increases with increase in system size (to maintain 25% load), but the individual server capacities remain the same, the skew in input is heavier for larger system sizes. Thus, larger system sizes drop correspondingly more queries while the adaptation takes effect, but in all cases, LAR is able to control the skew and eventually create sufficient replicas and reduce drops to zero within about two minutes of simulation time.

4.6. Other results

We have conducted several experiments including, comparisons with different *static replication techniques*, using different time windows in which servers compute their load and experiments analyzing the effects of the dissemination constants. However, we cannot present the results here in detail due to space constraints. These results are available in the companion technical report [12].

5. Summary and Conclusions

This paper has described LAR, a new soft-state replication scheme 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).

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 peers within configurable bounds.

References

- [1] Akamai home page. <http://www.akamai.com/>.
- [2] 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, December 1996.
- [3] D. R. Angelos Stavrou and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP 2002)*, Paris, France, November 2002.
- [4] B. Bhattacharjee, P. Keleher, and B. Silaghi. The design of TerraDir. Technical Report CS-TR-4299, University of Maryland, College Park, MD, October 2001.
- [5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM '99 conference*, March 1999.
- [6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *Proceedings of the ACM SIGCOMM '03 Conference*, August 2003.
- [7] Cisco localdirector. <http://www.cisco.com/warp/public/751/lodir/>.
- [8] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system, 2000.
- [9] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *The ACM SIGCOMM'02 Conference*, August 2002.
- [10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.
- [11] Gnutella home page. <http://gnutella.wego.com>.
- [12] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. Technical Report CS-TR-4515, University of Maryland, College Park, MD, July 2003.
- [13] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.
- [14] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.
- [15] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.
- [16] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. 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*, pages 654–663, May 1997.
- [17] Q. Lv, P. Cao, E. Cohen, K. Li, and S. 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.
- [18] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, Monterey, CA, August 2002.
- [19] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *International Conference on Distributed Computing Systems*, pages 101–113, 1999.
- [20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.
- [21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.
- [22] A. Rowstron and P. 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*, Chateau Lake Louise, Banff, Canada, October 2001.
- [23] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.
- [24] S. Saroiu, P. K. Gummadi, and S. 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.
- [25] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability, February 2001.
- [26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. 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.
- [27] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.
- [28] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for wide-area location and routing. Technical Report UCB//CSD-01-1141, U.C.Berkeley, Berkeley, CA, April 2001.