An Adaptive Probabilistic Replication Method for Unstructured P2P Networks

Dimitrios Tsoumakos and Nick Roussopoulos

Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Abstract. We present *APRE*, a replication method for unstructured Peer-to-Peer overlays. The goal of our method is to achieve real-time replication of even the most sparsely located content relative to demand. *APRE* adaptively *expands* or *contracts* the replica set of an object in order to improve the sharing process and achieve a low load distribution among the providers. To achieve that, it utilizes search knowledge to identify possible replication targets inside query-intensive areas of the overlay. We present detailed simulation results where *APRE* exhibits both efficiency and robustness over the number of requesters and the respective request rates. The scheme proves particularly useful in the event of flash crowds, managing to quickly adapt to sudden surges in load.

1 Introduction

Peer-to-Peer (hence P2P) computing represents the notion of sharing resources available at the edges of the Internet. Its success can still be largely attributed to file-sharing applications which enable users worldwide to exchange locally maintained content. A basic requirement for every P2P system is fault-tolerance. Since the primary objective is resource location and sharing, we require that this basic operation takes place in a reliable manner. In a variety of situations, the distributed and dynamic nature of the target environments stress the system's ability to operate smoothly. For example, the demand for certain content can become overwhelming for the peers serving these objects, forcing them to reject connections. *Flash crowds*, regularly documented surges in the popularity of certain content, are also known to cause severe congestion and degradation of service [1]. Failing or departing nodes further reduce the availability of various content. Consequently, resources become scarce, servers get overloaded and throughput can diminish due to high workloads.

Data replication techniques are commonly utilized in order to remedy these situations. Replicating critical or frequently accessed system resources is a well-known technique utilized in many areas of computer science (distributed systems, databases, file-systems, etc) in order to achieve reliability, fault-tolerance and increased performance. Resources such as content, location of replicas, routing indices, topology information etc, are cached/replicated by multiple nodes, alleviating single points of contact in routing and sharing of data. This has the additional benefit of reducing the average distance to the objects. Replication can be performed in a variety of manners: Mirroring, Content Distribution Networks (CDNs [2, 3]), web caching [4], etc.

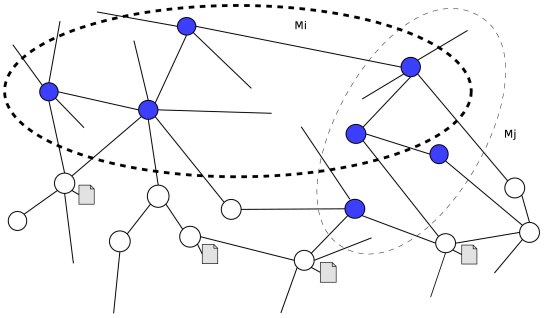


Fig. 1. Part of the overlay network of our model. Dark nodes inside the bold dotted ellipse represent $\mathcal{M}_i$, while those inside the thin dotted ellipse represent $\mathcal{M}_j$. Peers with a file attached also serve objects i or j

However, these approaches often require full control and provide static replication. Static replication schemes require a priori knowledge of the popularity/workload distribution in order to compute the amount of replicas needed. In large scale unstructured P2P networks, peers usually operate on local knowledge, having variable network connectivity patterns and no control over the induced topology or workload. Data availability and efficient sharing dictate replication in this challenging environment. Structured P2P systems (DHTs) provide with the state necessary to accurately identify the paths that requests take. However, such information is not available in unstructured overlays. File-sharing applications implicitly handle replication through object downloads, while some force their users to maintain the new replicas for the benefit of others. Yet, this does not tackle the issue of real-time replication responsive to workload for unstructured environments.

In this work we present *APRE* (Adaptive Probabilistic REplication), a replication method based on soft-state probabilistic routing indices. Our approach focuses on providing an adaptive solution to the problem of availability together with minimizing the instances of server overloads and serious service degradation. We intend for our system to "expand" and "contract" its resources according to the workload as perceived locally. New replicas are created in areas of high demand in the overlay, thus disposing of the need of advertising them. Moreover, this will be done in a completely decentralized manner, with minimal communication overhead and using absolutely affordable memory space per node.

1.1 Our Framework and Overview of APRE

We assume a *pure* Peer-to-Peer model, with no imposed hierarchy over the set of participating peers. All of them may serve and request various objects. Each peer locally maintains its own collection of objects, as well as a local view of the system. Ignoring physical connectivity and topology from our talk, we generally expect peers to be aware of their one-hop neighbors in the overlay, while maintaining any other protocol-specific information (e.g., search indices, routing tables, etc). The system is expected to exhibit

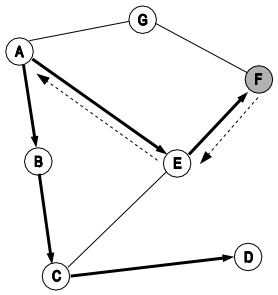
a dynamic behavior, with peers entering and leaving at will and also inserting/removing objects from their repositories. The overlay structure will also be affected, since nodes are not guaranteed to connect to the same neighbors each time.

As a motivating example, assume an unstructured P2P system, where peers share and request replicated resources. Objects are assumed to be requested regularly, e.g., results of a live sports meeting, weather maps, security updates, real time aggregated statistics, software, etc. There exist some nodes (similar to the web servers or mirror sites in the Internet) that provide with fresh content, but their connectivity or availability varies, as happens with all other network nodes. Peers that are interested in retrieving the newest version of the content conduct searches for it in order to locate a fresh or closer replica.

Figure 1 gives a graphic representation of our system. For each object i, there exists a set of peers called the *server set* $\mathcal{S}_i = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ that serve the specific object. These are the nodes that, at a given time, are online, store object i and are willing to share it. A subset of $\mathcal{S}_i$, the *mirror set* $\mathcal{M}_i \subseteq \mathcal{S}_i$ (the shaded peers) represents the set of peers that, if online, *always* serve i. This does not imply that all peers in $\mathcal{M}_i$ will always be online, their connectivity in the overlay will remain the same, or that they will never refuse connections. But we can assume, without loss of generality, that these nodes will be mostly available. Our assumption is not unrealistic: Imagine that these servers can represent mirror sites/authority nodes that provide with up-to-date content. Nevertheless, they are not guaranteed to be always on-line, nor do they provide similar services. Apart from the mirror set, other peers that already host or have recently retrieved an object can serve requests for it (nodes with files attached to them in Figure 1). A server set comprises of these nodes plus the corresponding mirror set.

Naturally, peers may belong to server or mirror sets for multiple objects. While this is a symmetric environment, it is clear that nodes will exhibit different sharing abilities. A variety of parameters, including storage and CPU capability, popularity of stored objects, system workload, connectivity, etc, contribute to this fact. Some of these factors remain more or less static over time (e.g., processing power or the maximum available bandwidth of a host), while others change dynamically. In this work, we focus on two of these parameters, namely workload and object popularity as they are manifested through the request rate λ. It is obvious that servers of popular (or temporally popular) items receive a larger number of requests, which can possibly affect their sharing ability as well as the system's behavior.

Given this general framework, our goal is to design and implement a replication protocol that will provide efficient sharing of objects (in terms of providing low load operation), scalability and bandwidth-efficiency. *APRE* is a distributed protocol that automatically adjusts the replication ratio of every shared item according to the current demand for it. By utilizing inexpensive routing indices during searches, loaded servers are able to identify "hot" areas inside the unstructured overlay with a customizable push phase. Chosen nodes receive copies thus sharing part of the load. Under-utilized servers become freed and can host other content. The rationale behind *APRE* is the tight coupling between replication and the lookup protocol which controls how searches get disseminated in the overlay. By combining the Adaptive Probabilistic Search (*APS*) state with *APRE*, we are able to identify in real-time "hot" or "cold" paths and avoid the



Indices	Initially	After walkers finish	After the updates
A→B	30	20	20
B→C	30	20	20
C→D	30	20	20
A→E	30	20	40
E→F	30	20	40
A→G	30	30	30

Fig. 2. Node A searches for an object stored at node F using APS (pessimistic) with two walkers. The table shows how index values change. X→Y denotes the index value stored at node X for neighbor Y relative to the requested object.

need of advertising constantly created replicas. Furthermore, we show that this method provides a very robust replication with minimum change in the server set per replication cycle.

2 Probabilistic Resource Location

2.1 Probabilistic Search

We now briefly describe the *APS* [5] search method, which is the basis for our replication scheme. In *APS*, each node keeps a local index consisting of one entry for each object it has requested per neighbor. The value of this entry reflects the relative probability of this node's neighbor to be chosen as the next hop in a future request for the specific object. Searching is based on the deployment of k independent walkers and probabilistic forwarding. Each intermediate node forwards the query to one of its neighbors with probability given by its local index. Index values are updated using feedback from the walkers. If a walker succeeds (fails), the relative probabilities of the nodes on the walker's path are increased (decreased). The update procedure takes the reverse path back to the requester and can take place either after a walker miss (*optimistic* update approach), or after a hit (*pessimistic* update approach). Figure 2 shows an example.

APS exhibits many plausible characteristics as a result of its *learning* feature. Every node on the deployed walkers updates its indices according to search results, so peers eventually share, refine and adjust their search knowledge with time. Walkers are directed towards objects or redirected if a miss or an object deletion occurs. *APS* is also bandwidth-efficient: It induces zero overhead over the network at join/leave/update operations and displays a high degree of robustness in topology changes.

2.2 Utilizing Search indices

One interesting observation is that the values of the stored indices are refined as more searches take place, enabling the network to build a useful soft-state. After some queries

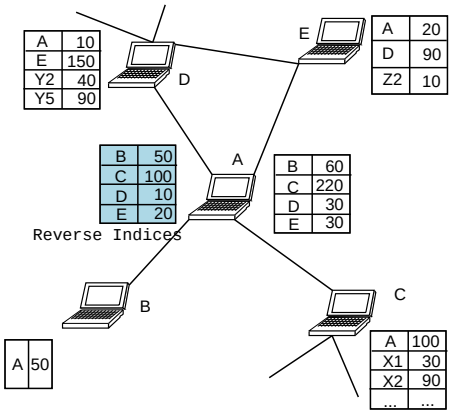


Fig. 3. Graphic explanation of the reverse indices. The filled table represents the reverse index values stored at node A, which coincide with the APS index values that nodes B,C,D,E store regarding A

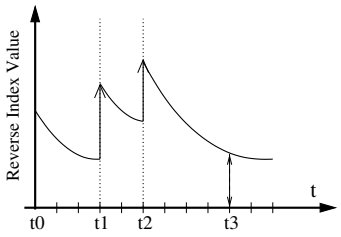


Fig. 4. Example of exponential decay in the reverse index values

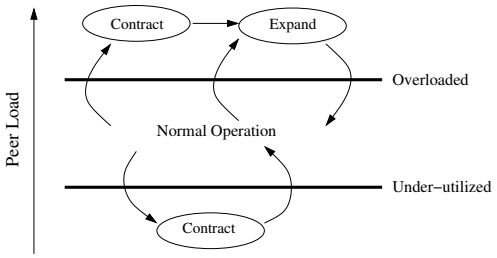


Fig. 5. State transitions in our system

take place, paths with large index values connect the requesters to the content providers and identify query-intensive areas inside the overlay.

APS keeps an index value for each neighbor. Extending this, each peer P also maintains the index values that its neighbors hold relative to P. If $X \rightarrow P$ denotes the index value stored at node X concerning neighbor P for a particular object, then peer P must know $X \rightarrow P$, for each neighbor X. These values can be made known to P during the search phase: Whenever a search is conducted and X forwards to P, it piggybacks $X \rightarrow P$. We call these new stored values the *reverse indices*, to distinguish them from the indices used by *APS* in searches (see Figure 3).

Reverse indices are used by nodes in order to forward messages along high demand paths in an unstructured overlay. These messages can be forwarded either to the top-k or probabilistically selected neighbors on a hop-by-hop basis. They are discarded either when their TTL value reaches zero or if they are received by a node more than once due to a cycle. Reverse indices get updated during searches, but this is not enough: There may be peers that have searched for an object and built large index values in the past,

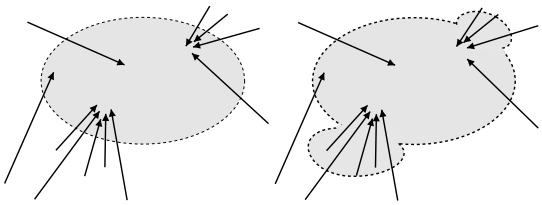


Fig. 6. The shaded oval represents a server set for a specific object. Our system expands by creating replicas inside two areas where demand (depicted by arrows) is high.

Algorithm 1 *Expand*

1: **if** Replica i at node s reaches its limit **then**
2: $P \leftarrow FindPossibleServers(i)$; $\{P \cap \mathcal{S}_i = \emptyset\}$
3: $Activate(i)$ at $Y \subseteq P$ {Replicate at a subset of the nodes in the high-demand area}
4: **end if**

but are no longer interested in it. If searches are no longer routed through those peers, the respective reverse index values will not be updated and will remain high.

To correct this situation, we add an *aging* factor to the reverse indices, which forces their values to decrease with time. Peers need to keep track of the time that a reverse index was last updated in order to acquire its correct value before using it. When a peer receives a search message, it sets the corresponding reverse index to the piggybacked value and its last modified field to the time of receipt. We describe this process in Figure 4. The value of the index decreases exponentially, while two searches at times t_1, t_2 reset its value. A push message received at time t_3 will use the value as shown in the figure. The last modified value is also reset when a reverse index is used, since a peer computes its current value before using it.

3 Our "Expand-Contract" Technique

Our main goal is to provide a completely decentralized mechanism through which the system will adaptively expand its replica size when demand is increased and will shrink when demand will fall. *APRE* is based on two basic operations: *Expand* and *Contract*.

The high-level behavior of our system can be described using a simple model (Figure 5): In normal mode, nodes can adequately serve requests and also retrieve objects. As load increases due to incoming requests, some reach their self-imposed limits. By invoking the *Expand* process, we aim at bringing the node status back to normal and lower the average load for a specific object through the creation of more replicas. Normal operation through the distribution of load will not be necessarily achieved in a single step. Consider, for example, that a peer initiating *Expand* may receive requests for multiple objects. Expanding with respect to one of them will probably lower its load, but will not necessarily bring its level back to normal. As load decreases, nodes can free up space (and the respective resources) and thus share a bigger portion of the workload.

Conversely, consider that one or more subsets of $\mathcal{S}_i$ have recently received very few requests for object i. This practically means that an amount of their storage space is

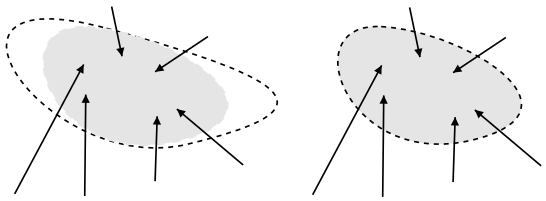


Fig. 7. Due to low demand in certain regions of the server set (depicted as white areas inside the dotted line), our system contracts its replica set

Algorithm 2 *Contract*

```
1: if (Replica i at node s is under-utilized) or (s receives Activate(j)) then
2:     i ← ChooseObject(); {i is among the candidates for eviction}
3:     Deactivate(i);
4:     if (s received an Activate(j)) then
5:         Activate(j);
6:     end if
7: end if
```

under-utilized. They could remove i to free up space or replace it with another object of higher demand. We have to stress here the point that the system will not force any peer to store or serve an object until this becomes necessary. Peers with available storage can play that role. *Contract* will also be invoked when a peer is called to join $\mathcal{S}_i$ but cannot do so without exceeding its limits (e.g., available storage). Note that peers can still choose to reject a certain action, e.g., refuse to remove an object in order to serve a new one.

Algorithm 1 describes the high-level operation of the *Expand* process. It is invoked by peers receiving more requests than those that they are willing to accept. Overloaded peers have to identify the set P, i.e., candidate nodes for replication inside query intensive areas. A subset Y of these nodes is selected and, upon their agreement, the new replicas are transfered (*Activate*). Figure 6 shows an example of our system expanding in response to increased demand for a specific object. On the left, we see some initial server set (gray oval) and the demand for i (arrows from various parts of the network). Servers in two areas are overloaded with requests, thus forcing extra replicas in those two areas to be activated. $\mathcal{S}_i$ expands, as we see on the right part of the picture, in response to the current demand for object i.

Algorithm 2 describes our *Contract* process. It is invoked by a peer that either receives a low amount of requests for the object(s) it serves or is requested to serve a more popular one but cannot do so without freeing up some space. In any case, peers stop serving the object(s) that fall into these categories (*Deactivate*). Function *ChooseObject* decides at each point which object should be deactivated at nodes that have decided to serve a new object (i.e., received an *Activate*) but have reached their storage capacities. Natural choices are to have the new replica replace the least recently requested or the least popular one. Figure 7 shows that two areas of the server set (the areas inside the dotted line) do not receive any requests for object i. This leads to the contraction of $\mathcal{S}_i$

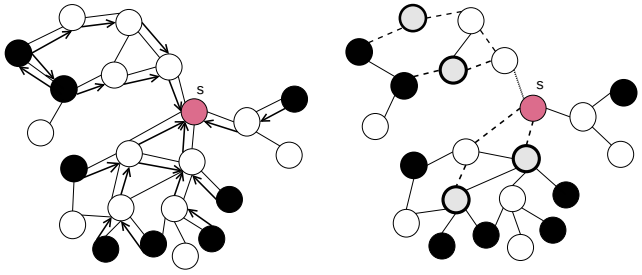


Fig. 8. After searches for an object at s take place, reverse index values are updated and a push phase creates new replicas inside areas of high demand (dotted links)

which is now the gray oval on the right part of the figure. Our goal is to achieve a system behavior that resembles the buffer management techniques in databases: Viewing the P2P network as a large buffer, we want to decide (in a distributed and dynamic manner) the ratios of objects in the buffer according to user-specified queries (i.e., workload).

3.1 Protocol Implementation

In this section we describe the actual implementation of the *APRE* protocol as described by the *Expand* and *Contract* algorithms. We assume that servers measure load and perform replication on a per-object basis, at the same level of granularity with lookup and reverse indices.

The conditions of line 1 in Algorithms 1 and 2 describe when *Expand* or *Contract* are initiated. We believe that each peer can independently choose when to initiate an expansion or when to deactivate a replica. Therefore, there is no need for any message exchange between servers. We assume that each server s defines the maximum number of requests that each object i can accept per time unit $Limit^{up}_{s,i}$. If it receives less than $Limit^{down}_{s,i}$ requests for object i, this replica is deactivated/deleted from the node's cache without any further communication. Obviously, the total maximum capacity for server s is equal to $\sum_i Limit^{up}_{s,i}$, where i refers to every object that s serves.

In order to discover candidate new servers to host replicas of i (i.e. locate subset Y), whenever the local load for object i (measured in requests per time unit) $\lambda^s_i(t)$ exceeds the limit $Limit^{up}_{s,i}$, the respective server s issues a special message which is forwarded to k neighbors with the k highest reverse index values. The push message also contains the amount of overload $D_i(t) = \lambda^s_i(t) - Limit^{up}_{s,i}$. Each node that receives this message, independently decides whether to join S_i according to our implemented replication policy. This phase continues with each intermediate node forwarding this message to k neighbors in a similar fashion until either its TTL value reaches zero or a duplicate reception is detected. Figure 8 shows an example of our scheme at work: Black nodes represent requesters of the item held at node s. *APS* searches are depicted by arrows. In the push phase, paths with high index values are visited (links with dotted lines). The new shaded nodes with bold outline represent possible replicas created.

Each node on the path independently decides whether it will join S_i according to our replication policy. Currently, we have implemented three: *FurthestFirst*, *ClosestFirst* and *Uniform*. In *FurthestFirst*, the probability of a node joining S_i increases with the message distance, while the opposite occurs in *ClosestFirst*. In *Uniform*, all nodes are given the same probability. After subset Y has been identified, replicas are transmitted and activated.

In order for *APRE* to adapt to various workloads and avoid system oscillation at the same time[1], we introduce a *scaled* replication policy: We regulate the number of replicas activated per push phase according to the amount of overload for object i, $D_i(t)$, as observed by the server initiating the push at time t. To achieve that, we define a set of intervals $\{d_1, d_2, \ldots, d_m\}$ that group the different values of D_i. Each interval $d_k : \{(l_k, u_k), \{p_{k_1}, p_{k_2}, \ldots, p_{k_{TTL}}\}\}$ is defined by an upper and lower value and TTL probability values, one for each hop distance. For the interval limits, we require that $l_1 < u_1 = l_2 < u_2 \ldots < u_m$. When a server receives a push message, it joins $\mathcal{S}_i$ with probability p_{k_δ}, if $l_k < D_i \leq u_k$ and the message has travelled δ hops. Probability values increase as D falls into higher number intervals (i.e., $p_{k_\delta} < p_{(k+1)_\delta}$). Thus, a heavily overloaded server will create more replicas than a less overloaded one and marginally overloaded peers will not alter $\mathcal{S}_i$ significantly. We note here that each server locally estimates $\lambda_i(t)$, the number of requests for object i per time unit.

4 Results

We test the effectiveness of *APRE* using a message-level simulator written in C. Object requests occur at rate λ_r with exponentially distributed inter-arrival times. At each run, we randomly choose a *single* node that plays the role of the initial $\mathcal{M}_i \equiv \mathcal{S}_i$ set and a number of requesters, also uniformly at random. Results are averaged over several tens of runs. We utilize 10,000-node *random* graphs with average node degrees around 4 (similar to gnutella snapshots [6]) created with the *BRITE* [7] topology generator.

To evaluate the replication scheme, we utilize the following metrics: The average load Λ observed as the number of received requests per second in $\mathcal{S}_i$. To measure the disparity of the load distribution, we measure its standard deviation σ_Λ. A high value for σ_Λ indicates that load is very unevenly balanced across $\mathcal{S}_i$. Besides the size of the server set, we also keep track of the number of replica activations/de-activations. Frequent changes in $\mathcal{S}_i$ incur huge overheads in terms of messages and bytes transferred.

APRE Parameters: We assume that $(Limit_s^{up}, Limit_s^{down}) = (18, 3)$ requests/sec, for each server s. When *Expand* is initiated, peers forward to 2 neighbors with the largest reverse index values. We utilize a scheme with 3 distinct intervals for values of D: $[0-5], (5-20]$ and $(20-\infty)$. While we experimented with more fine-grained granularities, the results did not vary considerably. Finally, we assume no item can be replicated at more than 40% of the network nodes[2].

We compare our method against a random replication scheme as well as path-replication as applied by Freenet [8] (and in effect by *lar* [9]). In the first case, we

[1] replicas with perceived load a little above or below the limits, frequently entering and leaving $\mathcal{S}_i$

[2] This external condition simulates the natural limitations in storage that exist in most systems

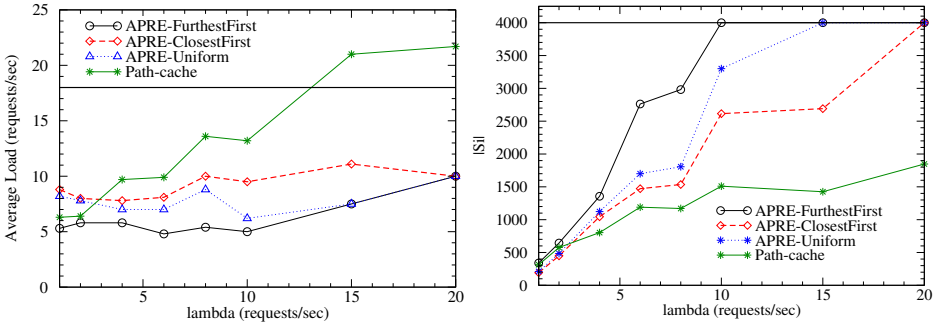


Fig. 9. Variation in Λ and $|\mathcal{S}_i|$ over increasing λ_r values

randomly create the same number of replicas as our method. In path replication (hence *path-cache*), each time a server is overloaded we replicate the object along the reverse path to the requester. In every case, the *APS* method is used for lookups, while in *path-cache* replicas are also deactivated using our *Contract* scheme. Obviously, by varying the push method and the replication probabilities, *APRE* can behave either as *path-cache*, random, or in between with a variable rate of replica creation per workload. This allows for full customization according to the system's primary objects, namely low load (more replicas) or space (replicas only where necessary).

4.1 Basic performance comparison

For our default setting, we assume 2000 random requesters as we vary their request rate. Figure 9 presents the variation in Λ and $|\mathcal{S}_i|$.

APRE effectively expands $\mathcal{S}_i$ in order to accommodate increased demand and manages to keep all servers into the *Normal Operation* zone, well below $Limit^{up}$ (identified by the bold horizontal line). Our first observation is that *FurthestFirst* achieves lower Λ values by creating more replicas than *ClosestFirst*. Downstream paths during the "push" phase contact an increasing number of nodes as their distance from the initiator increases, thus giving *FurthestFirst* an increased probability of replication. *Uniform* behaves in-between, creating replicas equally at all distances. *Path-cache* exhibits a steeper increase in Λ and fails to keep its value within the acceptable region for large λ_r. Choosing only successful walks to replicate along, quickly "saturates" the frequently used paths with replicas. Increased demand merely forces the algorithm to utilize a few more paths, which is the reason why this method fails to increase the replica set to meet the limits.

It is interesting to note that *APRE* exhibits small σ_Λ values, ranging from 3.3 to 11. It increases to 14.9 only when $\lambda_r = 20/sec$ (see Figure 10). These values are either smaller or at most comparable to Λ, a good indication of load balancing. On the other hand, randomly placing the same number of replicas yields significantly worse load distributions, with σ_Λ values roughly twice as large. This is a clear indication of the need for correct placement inside structureless multi-path overlays. Finally, *path-cache* behaves in-between, with larger deviation values than *APRE* that converge as load in-

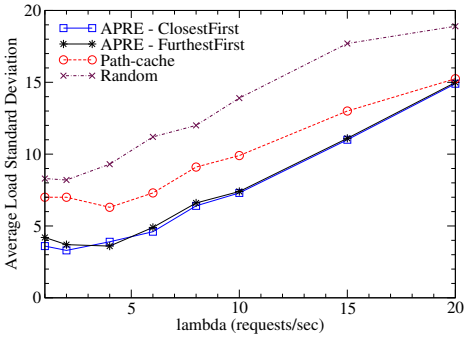


Fig. 10. Variation of σ_Λ vs. variable λ_r

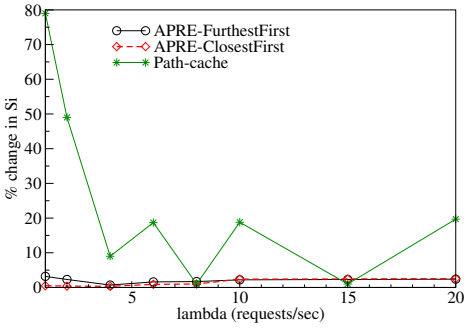


Fig. 11. Percentage of change in $|\mathcal{S}_i|$

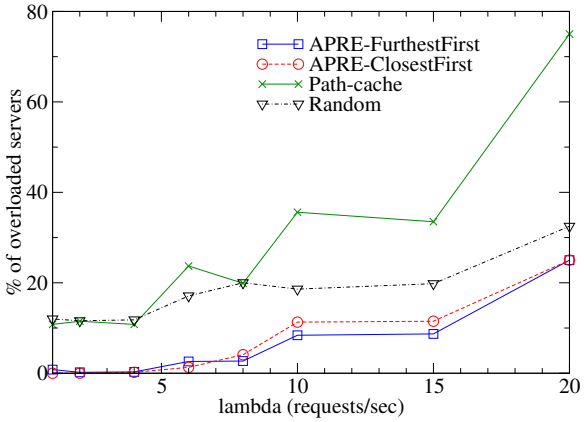


Fig. 12. Ratio of overloaded servers vs. variable λ_r

creases. This happens since both methods base their replication on paths connecting requesters and servers.

Moreover, we show that *APRE* achieves a much more robust replication. The stability of the server population constitutes an important metric to the evaluation of a replication scheme. This is measured by the average ratio of new replicas entering the server set per replication phase over the size of the server set. This quantity approximates the amount of marginally under-utilized replicas in the overlay: Receiving few requests, they get deactivated. Server overloads force them to get re-activated, producing an oscillating effect. Obviously, this is a highly undesirable situation: network and local resources are burdened by a multiplicative factor, since replicas need both control messages and data transfer for reactivation. Figure 11 shows that *APRE* is particularly robust, altering at most 3% of $\mathcal{S}_i$ per push phase, while *Path-cache* oscillates and performs poorly in most runs. altering a large percentage of the server set. The variability in the amount of oscillation is due to the effect we described before: An increase in the demand is not always followed by an increase in the number of replicas. In these

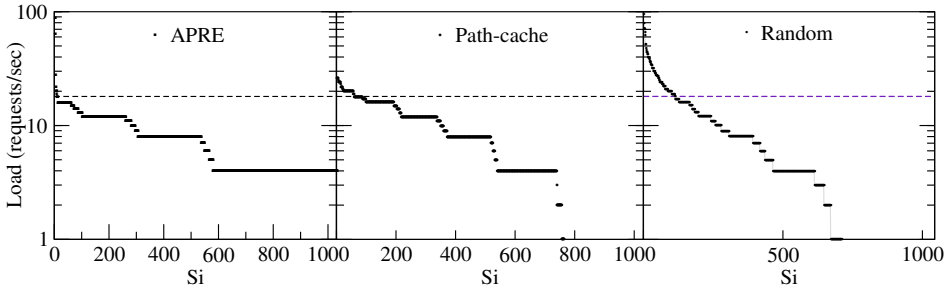


Fig. 13. $\mathcal{S}_i$ load distribution for $\lambda_r = 4/sec$

situations, the existing ones receive the extra amount of requests (assisted by the *APS* scheme), thus reducing the marginally idle servers.

Figure 12 displays the average percentage of overloaded servers at any time for all three methods. Our technique clearly outperforms the two competing methods: For $\lambda_r < 10/sec$, less than 4% of servers are overloaded, while about 10% and 25% are documented as overloaded for the largest demand. *Random*, having the same number of servers, exhibits twice as many overloaded nodes. Even though the learning feature of *APS* helps in redirecting queries, yet the load cannot be evenly distributed. *Path-cache* shows the worst performance (at least 3 times larger ratio of overloaded peers than *APRE*), reaching 75% at the highest λ_r value. Replicating closer to requesters creates, as we saw, more service points, thus marginally reducing the number of overloaded instances for *FurthestFirst* (*Uniform* exhibits the same curve).

Figure 13 shows the load distribution of every server s at a random point in time ($\lambda_r = 4/sec$). Servers are sorted in decreasing order of load. Our method exhibits a less steep curve and, more importantly, has only 3 servers above $Limit^{up}$, compared to 54 for *path-cache*. Random replication causes even more unbalanced load.

The same experiment is repeated with 5,000 requesters, which constitute 50% of the overlay (see Figures 14 and 15). *APRE* again keeps the system within its limits, except for the two cases where even the largest replica set cannot achieve that (75k and 100k total queries per second). Still, our method shows remarkable stability in the $\mathcal{S}_i$ population for both strategies, while *Path-cache* exhibits even worse performance than before.

Finally, Figure 16 shows how Λ and $|\mathcal{S}_i|$ vary with time, using *ClosestFirst*. For all values of λ_r, *APRE* manages to bring Λ to a steady state in few time steps, a state which is hence maintained with almost no deviation. The same is true for the size of $\mathcal{S}_i$, with the exception that for high total demand, it takes longer to reach the steady state. This is due to the fact that there is a limit to the maximum amount of replication per push phase for our method (as there is for *path-cache*) that causes the delay in reaching the constant values.

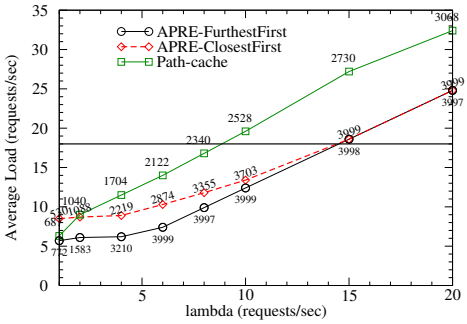


Fig. 14. Variation in the average load vs. variable λ_r (5000 requesters)

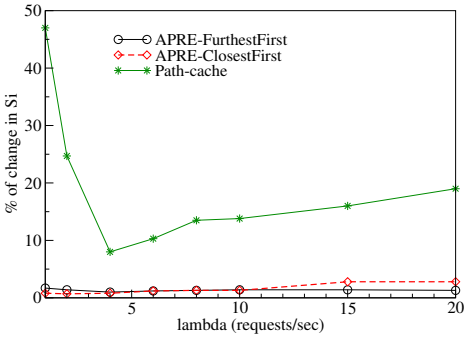
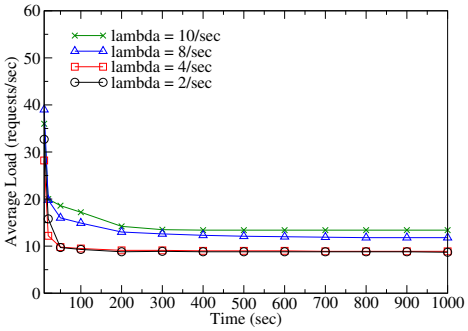


Fig. 15. Percentage of change in $\mathcal{S}_i$ and σ_Λ vs. variable λ_r (5000 requesters)



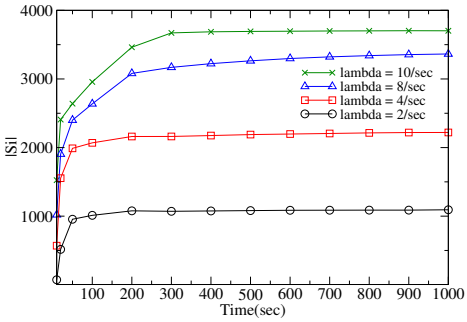


Fig. 16. Λ and $|\mathcal{S}_i|$ over time for 5000 requesters and multiple λ_r values

4.2 Flash Crowds

Thus far we established our basic premise, that replication along high demand paths in the overlay proves an effective and highly robust solution in a variety of metrics and workloads. Although our method does not explicitly offer load-balancing, it achieves a well-proportionate load distribution. We also showed that our method is advantageous to randomly replicating inside the network or merely choosing a single path and fully replicating along it. In the first case, few replicas receive the majority of requests, while in the second case the composition of the replica sets changes very frequently. Our method outperforms both alternatives by keeping fewer peers over the sharing limit and showing less disparity in the distribution of load among servers.

In the next experiment, we examine the behavior of our method when we experience a sudden surge in the workload. This is often referred to as a *flash crowd*, an unexpected rise in requests towards specific item(s), typically due to some newsworthy event that just took place. Flash crowds have been regularly documented in web traffic history (e.g., September 11th) and are known to cause severe congestion at the network layer. Requests may never reach content servers while others do so with significant delays, caused by packet loss and retransmission attempts. Content holders are unable to handle

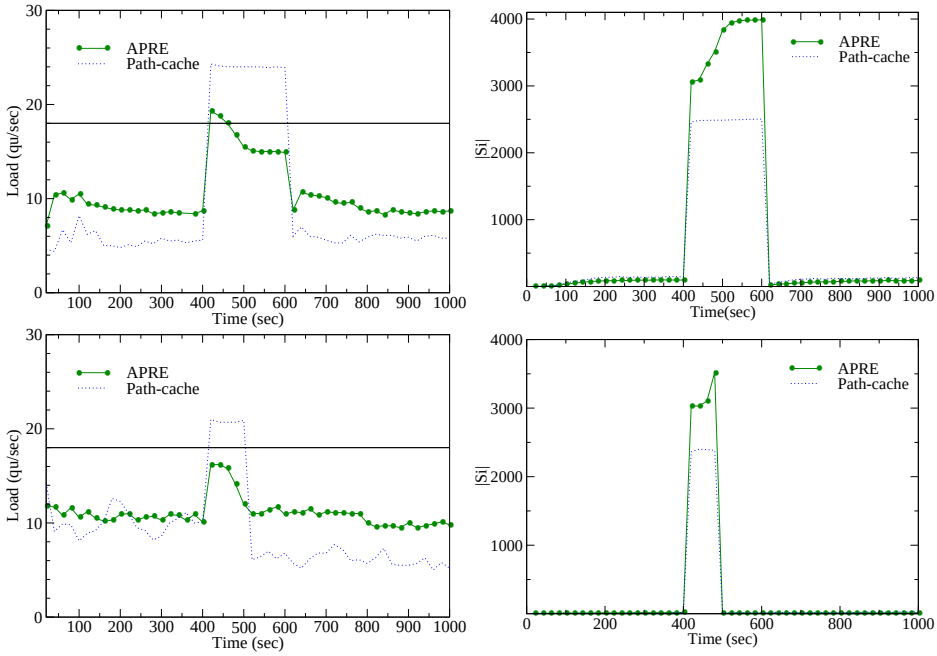


Fig. 17. Effect of flash crowds in Λ and $\mathcal{S}_i$ in two different settings

the volume of requests while end-users experience long delays and failures in their queries.

To simulate a similar situation, we start our system with 500 requesters querying at rate $\lambda_r = 2/sec$. At time t=401sec, 10 times as many requesters start querying for this item at rate $\lambda_r = 12/sec$. The parameters return to their initial values at time t=601sec. On average, the total demand during the flash-crowd period increases by a factor of over 70. Note that this is the worst case scenario, when simultaneously both requesters and rates increase. We present the variations in Λ and $|\mathcal{S}_i|$ in the first 2 graphs of Figure 17.

APRE promptly manages to meet the surge in requests by increasing the replication ratio by a factor of 30. Excluding a very short window due to our mechanism's response, our method succeeds in keeping the load factor below the limit (with $\sigma_\Lambda < 10$) and steady through time. At both moments of load change, replicas are activated and de-activated rapidly to meet the extra requests or reduced traffic. While *path-cache* shows similar response speed, it creates more servers in the low-workload period and less than the minimum number required to keep content providers from overloading during the surge.

The next two figures show how the same two metrics vary in a more challenging flash-crowd setting. Here, we initially set 500 requesters with $\lambda_r = 1/sec$, while for time $t \in (400, 480]$, 5000 requesters query at rate $\lambda_r = 10/sec$. On average, the workload inside the overlay increases by a factor of 120. Our results show that, even for shorter and steeper changes, *APRE* very successfully adapts to the surge in requests. On average, $\mathcal{S}_i$

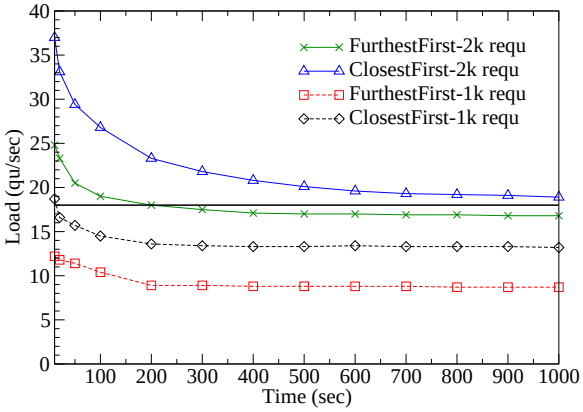


Fig. 18. Average load for 1k and 2k requesters in power-law topologies ($\lambda_r = 6/sec$)

is expanded by a factor of 175 in order to reduce and balance load (our results document an average $\sigma_\Lambda \simeq 8.6$).

4.3 Other Experiments

We test our method on a set of 4,000-node power-law graphs created with Inet-3.0 generator [10]. These graphs have an average degree of $d = 4.3$ (maximum degree equals to 855), while over 30% of the nodes have only one neighbor. Figure 18 shows how Λ varies with time for both replication strategies used in *APRE* using 1000 or 2000 nodes as requesters.

These topologies noticeably affect performance compared to our previous tests. Even for average-range λ_r values, Λ moves close to the overload line, while expansion shows diminished ability to extend $\mathcal{S}_i$. This is consistent with results documented in previous work [5]. The tested topologies offer fewer paths between servers and clients, while a large percentage of the nodes only have one neighbor. This also explains why *FurthestFirst* outperforms *ClosestFirst*. Favoring replication close to the requesters quickly saturates available nodes due to lack of alternate paths. Nevertheless, its is worth to notice that our method still manages to keep Λ at lower levels. Even at the 2k-*ClosestFirst* run, where $\Lambda > Limit^{up}$, 14% of the servers are overloaded compared to 20% by *path-cache*.

We must note here that the replication protocol is not always responsible for overloaded servers. In many occasions, the amount of demand or the overlay connectivity cannot allow for more extensive or balanced replication. As we experiment with more densely connected graphs, *APRE* performs inside the load limits where it failed to do so over more sparse overlays.

In the accompanying technical report [11], we also present results on our method's behavior over variable maximum replication ratios and different values for the load limits, as well as load-balancing analysis based on a different metric.

5 Related Work

Replication is a well-known technique utilized to achieve high availability and fault-tolerance in large-scale systems. While applied to a variety of contexts, we focus in the area of distributed (P2P) systems.

Structured overlays (DHTs) balance routing between network nodes, due to the nature of the hashing functions used. Moreover, in systems like CFS [12] and PAST [13], each item (or chunk of it) is replicated on a set number of network nodes. DHTs take advantage of the routing structure, which in effect allows for almost-deterministic paths between two nodes, thus identifying "hot" areas easily. Nevertheless, DHTs are not optimized for skewed access patterns and direct such traffic to few nodes responsible for popular content.

DHash [14] is a replication method applied on Chord [15]. The protocol allows for r copies to be stored at the r immediate successors of the initial copy's home. In [16], the authors propose the storage of at most R replicas for an object. Their location is determined by a hash function, allowing requesters to pro-actively redirect their queries. The work in [17] proposes replicating one hop closer to requester nodes as soon as peers are overloaded.

Lar [9] is a DHT-based approach similar to *APRE*, in that it adapts in response to current workload. Overloaded peers replicate at the query initiator and create routing hints on the reverse path. Hints contain some other locations that the content has been previously replicated, so queries are randomly redirected during routing. The method takes advantage of the DHT substrate in order to place the hints. Our scheme does not attempt to re-route queries or shed load to the initiator, but rather places replicas inside forwarding-intensive areas using multiple paths. Moreover, the state kept is accessible at any time, not only at the time of the query arrival.

HotRoD [18] presents a load-balancing approach for DHTs handling range queries in RDBMSs. It is based on a locality-preserving DHT and replication of overloaded arcs (consecutive modes on the DHT ring). [19] employes minimization function that combined high availability and low load to replicate video content inside a DHT. The approach requires knowledge of peer availabilities, workload and data popularity. In [20], the authors show that load-balancing based on periodic load statistics suffer from oscillation. By directing queries towards the the maximum capacity replica store, they show that both heterogeneity and oscillation issues are tackled. This method, nevertheless, assumes prior contact of an authority server which provides with a list of existing replicas. Moreover, replicas regularly advertise their maximum capacities to the network.

There has also been considerable amount of work on flash crowd avoidance. In [21], overloaded servers redirect future requests to mirror nodes to which content has been pushed. This approach does not tackle the issue of which node to replicate to. PROOFS [22] explicitly constructs a randomized overlay to locate content under heavy load conditions or unwilling participants. In effect, the method relies on the combination of custom overlay creation and a gossiping lookup scheme to locate objects and does not involve replication. Finally, the work in [23] discusses static replication in unstructured networks, given content popularity and random walks as a lookup method.

6 Conclusions

In this paper we presented our adaptive replication scheme for unstructured Peer-to-Peer systems based on probabilistic soft state. *APRE* aims at providing a direct response to workload changes, by creating server points in needy areas or releasing redundant servers in areas of low demand. Our approach couples lookup indices together with an aging mechanism in order to identify query intensive areas inside the overlay. Peers then individually decide on the time and extent of replication, based on local workload computation.

In this work, we show that it is important to couple replication with the search protocol in unstructured systems. Random replication performs poorly with informed lookup schemes, unless extra state is added to enhance searches. Applying *APRE* over a scheme such as *APS* mitigates this problem. *APS*-indices store local, per-object state to direct queries to objects. While peers only keep metadata about their neighbors, this information can be used to identify, hop-by-hop, where the queries are coming from. Moreover, our scheme is highly customizable allowing control of both the size and the location (as defined through reverse-indices) of replication.

Through thorough simulations, we show that *APRE* is extremely robust in eliminating server overloads while minimizing the communication overhead and balancing the load. Specifically, we show that replicating along the reverse path is an extreme case of our protocol. By effectively discovering all reverse paths, *APRE* manages to distribute content proportional to demand in a variety of overlays and workloads. Finally, we show that our method succeeds in creating a very stable server set with minimal amount of oscillation.

7 Acknowledgments

This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number DAAD19-01-1-0494

References

1. Jung, J., Krishnamurthy, B., Rabinovich, M.: Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In: WWW. (2002)
2. Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., Weihl, B.: Globally Distributed Content Delivery. IEEE Internet Computing (2002)
3. Freedman, M., Freudenthal, E., Mazires, D.: Democratizing Content Publication with Coral. In: NSDI. (2004)
4. http://www.squid-cache.org/: Squid Web Proxy Cache
5. Tsoumakos, D., Roussopoulos, N.: Adaptive Probabilistic Search for Peer-to-Peer Networks. In: 3rd IEEE Intl Conference on P2P Computing. (2003)
6. Ripeanu, M., Foster, I.: Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In: IPTPS. (2002)
7. Medina, A., Lakhina, A., Matta, I., Byers, J.: BRITE: An Approach to Universal Topology Generation. In: MASCOTS. (2001)

8. Clarke, I., Sandberg, O., Wiley, B., Hong, T.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. Lecture Notes in Computer Science (2001)
9. Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., Keleher, P.: Adaptive replication in peer-to-peer systems. In: ICDCS. (2004)
10. Jin, C., Chen, Q., Jamin, S.: Inet: Internet Topology Generator. Technical Report CSE-TR443-00, Department of EECS, University of Michigan
11. Tsoumakos, D., Roussopoulos, N.: APRE: A Replication Method for Unstructured P2P Networks. Technical Report CS-TR-4817, University of Maryland (2006)
12. Dabek, F., Kaashoek, M., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: SOSP. (2001)
13. Rowstron, A., Druschel, P.: Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In: SOSP. (2001)
14. Cates, J.: Robust and efficient data management for a distributed hash table Master's thesis, Massachusetts Institute of Technology, May 2003.
15. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: SIGCOMM. (2001)
16. Waldvogel, M., Hurley, P., Bauer, D.: Dynamic replica management in distributed hash tables. Technical Report RZ–3502, IBM (2003)
17. Stading, T., Maniatis, P., Baker, M.: Peer-to-peer caching schemes to address flash crowds. In: IPTPS. (2002)
18. Theoni Pitoura, Nikolai Ntarmos, and Peter Triantafillou: Replication, Load Balancing and Efficient Range Query Processing in DHTs. In: EDBT. (2006)
19. Poon, W., Lee, J., Chiu, D.: Comparison of Data Replication Strategies for Peer-to-Peer Video Streaming. In: ICICS. (2005)
20. Roussopoulos, M., Baker, M.: Practical load balancing for content requests in peer-to-peer networks. Technical Report cs.NI/0209023, Stanford University (2003)
21. Felber, P., Kaldewey, T., Weiss, S.: Proactive hot spot avoidance for web server dependability
22. Stavrou, A., Rubenstein, D., Sahu, S.: A lightweight, robust p2p system to handle flash crowds. In: ICNP. (2002)
23. Lv, C., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and Replication in Unstructured Peer-to-Peer Networks. In: ICS. (2002)