Web Search Results Caching Service for Structured P2P Networks

Erika Rosas[a,b], Nicolas Hidalgo[a,b], Mauricio Marin[a,b], Veronica Gil-Costa[a]

[a] *Yahoo! Labs Santiago, Chile*
[b] *DIINF, University of Santiago, Chile*

Abstract

This paper proposes a two-level P2P caching strategy for Web search queries. The design is suitable for a fully distributed service platform based on managed peer boxes (set-top-box or DSL/cable modem) located at the edge of the network, where both boxes and access bandwidth to those boxes are controlled and managed by an ISP provider. Our solution significantly reduces user query traffic going outside of the ISP provider to get query results from the respective Web search engine. Web users are usually very reactive to worldwide events which cause highly dynamic query traffic patterns leading to load imbalance across peers. Our solution contains a strategy to quickly ease imbalance on peers and spread communication flow among participating peers. Each peer maintains a local result cache used to keep the answers for queries originated in the peer itself and queries for which the peer is responsible for by contacting on-demand the Web search engine. When query traffic is predominantly routed to a few responsible peers our strategy replicates the role of "being responsible for" to neighboring peers so that they can absorb query traffic. This is a fairly slow and adaptive process that we call mid-term load balancing. To achieve a short-term fair distribution of queries we introduce in each peer a location cache which keeps pointers to peers that have already requested the same queries in the recent past. This lets these peers share their query answers with newly requesting peers. This process is fast as these popular queries are usually cached in the first DHT hop of a requesting peer which quickly tends to redistribute load among more and more peers.

Keywords: Web Search Engines, Caching Services, Load Balancing, P2P networks.

1. Introduction

Web search engines are systems devised to cope with a highly dynamic and demanding query rates, potentially of the order of many hundred thousand queries per second. For redundancy and fault tolerance, large search engines operate on multiple, geographically distributed data centers. Intensity of query traffic is featured by the unpredictable behavior of users who are usually very reactive to worldwide events [1]. Web search queries typically follow a Zipf-like distribution of data [2] [3] which produce load balancing problems over the query result cache servers.

This paper proposes implementing a result cache service for Web queries under the concept of a nano datacenter infrastructure implemented over a P2P network composed by

small peers. The nano datacenter architecture uses ISP-controlled home gateways, like the ubiquitous set-top-boxes or STBs [4], to provide computing and storage services, and adopts a managed peer-to-peer model to form a distributed infrastructure [5].

We organize the STBs following a Distributed Hash Table (DHT) model which generally provides better routing/searching performance with much smaller overheads when compared against unstructured P2P overlays [6]. DHTs provide the opportunity of building scalable, decentralized and self-organizing systems. These systems balance storage, transparently tolerate peer failures, and provide efficient routing of queries. However, they suffer from load balancing problems under the presence of highly unbalanced query distributions. The emergence of hot-spots (popular queries) and flash crowds can lead to saturation of peers the degrading performance of a results caching service.

Caching schemes for DHTs and P2P networks have been proposed for applications such as Web caching and streaming, and has been a subject that has deserved research. In our context, the main difference with systems studied in previous works is that a cache service for Web search engine queries must be devised to be efficient under the following requirements: (1) query traffic can be very dynamic both in intensity and trending topics, and it can be subjected to sudden peaks occurring at unpredictable time instants; (2) cached queries are not expected to remain valid for a long time as current search engines supporting real time search may include new documents within a few minutes from publication in the Web; and (3) it is necessary to transfer the whole query answer to the requesting user as opposed to pointers to answer objects for later retrieval.

To address these requirements, a portion of each peer main memory is used as a result cache to hold query answers retrieved from one or more Web search engines. These queries are assumed to be distributed uniformly at random on N peers by means of a hash function on the query terms. They correspond to queries submitted by users in the recent past. Thus, any given peer is responsible for a fraction of these queries and (on request) it must contact the appropriate Web search engine to get the current answers for them. Users may submit queries at any peer and each peer must respond with a query answer to each respective user.

The contribution of the proposed P2P result caching are mainly the following:

- A strategy that quickly reacts upon a sudden and drastic increase in query traffic where possibly query contents are highly skewed to a relative small number of different terms.
- A strategy to replicate the role of overloaded peers which provides load fairness among the peer replicas using a small amount of communication.

We propose using the two strategies in combination to efficiently cope with user query traffic on a P2P result caching system for Web search queries.

We evaluate our proposal by using a query log from a commercial Web search engine and compare it against state of the art baseline strategies. The results show that our approach (1) improves load balancing among participant peers, (2) increases cache hits, (3) reduces the amount of traffic generated within the peers community and towards the Web search engines, and (4) significantly reduces the communication volume associated with the replication of highly popular queries.

An early version of this paper was presented in [7]. In this paper we extend [7] to significantly improve the performance of the strategy devised to react upon sudden query traffic peaks and present a more comprehensive experimental performance study.

The rest of this paper is organized as follows. Section 2 surveys the related work and present some background concepts about our solution. Section 3 presents the architecture, model, and the proposed solution. In Section 4 we present a performance evaluation study conducted through process oriented discrete simulation. Finally, Section 5 presents concluding remarks.

2. Background and Related Work

In the following we briefly explain DHTs in order to make our paper self-contained. Then, we present related work on caching and replication in the context of DHTs.

2.1. Distributed Hash Tables

A Distributed Hash Table (DHT) is a self-organized structured substrate built over P2P overlay networks which provides data availability and persistence through replication. Every peer in such systems is associated with a unique identifier which defines the peer's position in the structure and the range of keys it is responsible for. Data is identified by a key in the same identifier space and can be located within a logarithmic number of routing hops.

DHT overlays maintain a strong topology, for example a ring in the case of Chord [8], Pastry [9] or Tapestry [10]. We design our solution in the context of Pastry [9] but it can be applied other DHT realizations.

Every peer in Pastry is assigned a unique nodeID in a space of 128-bits identifiers generated using a cryptographic hash SHA-1. The neighbors of a peer in Pastry are stored in the *leafset* that contains the L numerically closest peers, $L/2$ clockwise and $L/2$ counterclockwise. This set can handle replicas to improve fault tolerance in an environment where peers join and leave the network without warning. In order to maintain these two sets of peers, a "keep alive" message is used periodically to detect failed peers.

The Pastry routing algorithm is prefix based and it routes a message to the numerically closest peer of a given key k, and in this paper we call this peer the *responsible* of k. The Pastry routing table stores on the n^{th} row the IP address of peers whose nodeIDs share the first n digits with that of the present peer. The algorithm forwards the messages to a peer from its routing table that shares at least one more digit with the key k than the current peer. If no such a peer can be found and the current peer does not know any other peer numerically closer to k, then the current peer is the responsible of k and the routing ends.

In the case of Pastry, the *leafset* and the *routing table* compose the *out-links* of a peer.

2.2. Caching and Replication in DHTs

Caching and replication in DHTs are active areas of research. P2P cache schemes have been proposed mostly for web pages (Squirrel [11], BuddyWeb [12], Backslash [13]), storage systems (Aren [14], [15]) and video streaming applications [16].

Approaches for cache mainly focus on optimizing one metric, such as hit rate, latency, or communication.

Kangasharju et al. [17] propose a caching P2P community, as our work does. Their key assumption is that traffic is cheaper within the community than transferring content from external nodes. They propose an algorithm called Top-K MFR that stands for most frequently requested objects. Each node tracks the files for which it has received a request and retrieves from the outside only the most requested ones. Their model assumes that cached objects are large in space, like videos, and that the size of space for caching is relatively small, namely there is space to hold in the order of dozen objects. This is why selecting the objects stored in cache is their main focus. Our solution, on the other hand, is designed to handle many small-sized objects which occupy space in the order of a few KBs.

Despotovic et al. [18] aim at improving caching with respect to the total traffic originated within the DHT search. They propose to replicate an object i when a high demand for i is detected by sending a reference of i to the previous hops that forwarded the queries for i in the past. A threshold is defined in order to trigger the replication of references to i. Unlike our work, [18] replicates references to objects, while our solution builds on the need to send the full object (query results) to the requesting peer.

Tigelaar et al. [15] explores search result caching in a P2P model for information retrieval. They have found that a small size cache offers performance comparable to an unbounded size cache. Moreover, throughout simulation they have shown than in distributed scenarios, caching can greatly reduce query load. Their results also show that the Least Recently Used (LRU) policy is the best approach in the P2P setting.

PCache [19] is a proactive caching strategy based on popularity of objects. Replicas are stored in the nearest overlay nodes to the peer that is responsible for an object. It periodically estimates the popularity value of a content by using an aggregation protocol among the peers, which can be costly in a large scale network.

Some of the authors of this paper have proposed solutions for results caching outside the P2P model environment [20] [21] [22]. Their work have focused on caching results for Web Search Engines deployed on clusters of processors. We have used one of the terms presented in [20] to name one of the structures proposed in our paper: Location Cache. The Location Cache in [20] reduces the amount of hardware involved in the solution of search engine queries by storing the nodes which provided the results for a query in the backend. With this structure they can also select the search nodes most likely to provide a good approximated answer to queries so that queries are prevented from hitting all search nodes (processors) when operating under near saturation query loads.

Replication has been done mainly in two ways: *multiple hash strategy* and *path replication*. The multiple hash strategy is described in [23][24][25]. However, since DHTs exhibit path convergence, multiple hash strategies are less adapted to efficiently handle highly popular objects than path replication.

Most DHT based strategies select a group of neighbor peers to replicate the data (e.g., the leafset in Pastry [9] contains neighboring peers). This technique uses uniform replication in a fixed number of peers. This is done to achieve high availability and fault tolerance. In the path replication techniques, the goal is to have other peers to answer queries instead of

the responsible peer. These replicas could be even closer to the source than the responsible peer.

Stading et al. [13] use a local cache diffusion method which pushes the replicas of a document (object) to one hop closer to the source of the last requests. This flood creates a *bubble* which grows in relation to the intensity of the flood, until no peer on the perimeter of the bubble observes high request rates. However, details on how this decision is taken are not given. They also propose a directory diffusion, but this is not adapted to hot-spots.

Bianchi et al. [26] compute a popularity value, maintaining a counter for each object and a query counter. When a peer is overloaded, it replicates the most popular objects in the most frequent last peer along the path towards the destination peer.

The solutions in [13] and [26] suffer from the *bubble* problem where a responsible peer stops answering queries for a popular key.

Yamamoto et al. [27] propose a path adaptive replication method which determines the probability of replication according to a predetermined replication ratio and storage capacity. Even though it considers the storage availability of peers, it ignores their current load.

Ramasubramanian et al. [28] replicate objects based on the Zipf parameter α in order to minimize the resource consumption using local measurements and limited aggregation. However, flash crowds can deviate the requests from the Zipf distribution. A popular object is replicated by levels in all the peers that share one less digit with the key than the responsible peer. This work reduces latency of popular objects, but it creates a high number of replicas in the system which depends on the size of the network.

Silvestre et al. [14] propose a replication scheme called Aren, which takes into account content popularity and QoS metrics in content distribution systems for edge networks. Their strategy uses a hierarchical network for a cloud environment and relies on the use of a coordinator to optimise scheduling and replication. The use of edge networks to build the system is similar to the presented in this work, however, our P2P model is fully distributed.

In order to cope with load balancing issues in structured P2P networks, not only replication has been used. Most of the solutions focus on providing a uniform distribution of the range of keys for peers. Works like [29] for example, assume uniform workload and balance the namespace. The virtual server strategy is also widely used, where a single physical node may have multiple virtual nodes in the logical structure. This model enables resource aware load balancing, since peers may vary the number of virtual nodes depending on their capacity. A drawback of this type of solution is that a physical node has to handle multiple routing tasks for maintenance and forwarding [30][31][32] .

Solutions that modify a node's identifier to redistribute the load are also ignoring the skew distribution of queries. If the identifier of a node changes the partition assigned is also moved between adjacent nodes. The same occurs if a node leaves and join at a different place in the ring, like in [33] and [34], then the responsible peer has changed but not the load. Low loaded peers can exchange place in with highly loaded peers [35] [36], but the responsibility stay at a single peer . Moreover, security is affected when modifying the nodes identifier. Ledlie et al. [37] proposed the paradigm of k-choices to overcome this issue. With this technique many random choices of nodeIds are used in order to have a set of variable

Ids and avoid arbitrarily change of identifier. Karger and Ruhl [38] propose that each node chooses $\log n$ places in the ring, and takes the responsibility for only one of them. This scheme has been used together with the virtual server strategy.

We have focused on replication since our scenario we need to cope with hot spots and skewed distribution of queries, more than consider the distribution of stored content in the network.

3. Proposed P2P Caching Service

3.1. Architecture

We propose to build a Caching Service (CS) to reduce query traffic towards Web search engines by using a P2P overlay composed of peers within a single geographic region and operating as a *nano datacenter* infrastructure. The idea is to create a fully distributed service platform based on managed boxes (set-top-box or DSL/cable modem) located at the edge of the network, where both boxes and access bandwidth to those boxes are controlled and managed by a given entity (e.g., by Telco, virtual operator or service provider) [39]. Figure 1 presents the proposed architecture. The set-top-boxes act like peers on the overlay. By utilizing virtualization technology the application running on a box can be completely transparent to the end user. The advantages of this approach rely on their low-energy consumption, self-scalability, and high availability. Additionally, a nano datacenter does not suffer from free-riding, peer dynamics, and lack of awareness of underlying network conditions [5]. Importantly, they provide access to a very large amount of resources. As an example, in USA there are more than 160 million set-top-boxes (STBs) which can potentially act as small servers [4].

For a P2P CS to be of practical interest the issue of low latency to get query answers is critical. On the other hand, structured P2P networks suffer from high latency because during search it is necessary to visit several peers to find the peer that contains the target object. In order to ease this problem, like in [11], we consider that peers remain within a geographic region.

In our architecture, user web browsers issue query requests to the P2P CS overlay. Peers (set-top boxes) route queries among peers in order to find cached information, namely answers of queries. Like in a standard DHT, terms of queries are hashed by using a cryptographic hash, such as SHA-1, and routed through the overlay to the peer which shares the closest ID with the hashed query. If the P2P CS finds a valid cached query answer it responds directly to the requesting user. Otherwise, the query is sent to the Web search engine to get the answer, cache it on the P2P CS and respond to the user. The main goal is to alleviate traffic towards search engines and reduce the costs for ISPs by reducing traffic outside their networks.

3.2. TLC+: Two-Level Caching

We assume a classical DHT overlay composed by N physical nodes (or peers) and K object-keys (query terms) mapped onto a ring. Objects (query terms plus the respective query answer) stored in a DHT such as Pastry or Chord, have a responsible peer in the

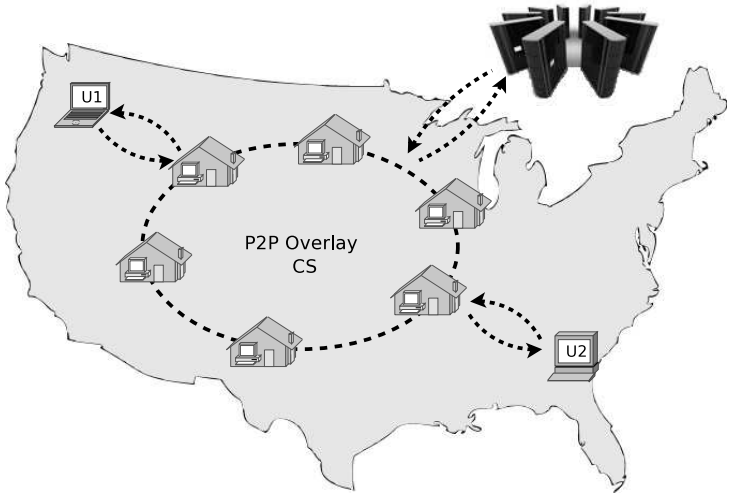


Figure 1: Architecture

network that is the peer with the closest ID to the key of the object. Thus, any given peer is responsible for a fraction of these queries and (on request) it must contact the appropriate Web search engine to get the current answers for them. Below we use query answer and query result to mean the same, namely a valid html page containing the results of executing the respective query a the Web search engine which can be directly sent to the user Web browser.

Participant peers have the ability to *receive*, *forward*, and *answer* a given query. In our model we assume that answering a query (i.e., sending a query answer to a requesting peer) consumes the highest amount of resources considering that query answers are expected to be of the order of tens of KBs. On the other hand, *receive* and *forward* operations are in the order of hundreds of bytes. Requesting an object to the Web search engine is assumed to be costly to the ISP since the communication traffic leaves outside its network.

Peers of the network create a collaborative cache which acts as a distributed CS. The set-top box provider defines a fixed storage space for caching query answers from the Web search engine for all peers.

The proposed P2P caching strategy is composed of two levels. The first one is devised to quickly react upon a sudden and drastic increase in query traffic where possibly query contents are highly skewed to a relatively small number of different terms. In this case, each peer p_i that submits a query q and receives the answer can potentially share it with other peers p_k that are requesting q during the same period of time. This quickly eases communication traffic on the peer r that is responsible for contacting the search engine and sending back the answer of query q to the requesting peer p_i. Namely, if peer p_1 requests the answer for a query q to its responsible peer r, and soon after the peers p_2 and p_3 request the answer for the same query q (in that order), then the proposed strategy causes a query answer transfer from r to p_1, then a transfer from p_1 to p_2, and finally a transfer from p_2 to p_3. Notice that the number of sequential transfers cannot be excessively large and parallelism

can be exploited for a new peer p_4 as peer p_1 can transfer the query answer to p_4 in parallel with the transfer from p_2 to p_3. Below we treat separately the sequential solution from the parallel one. The former is suitable for moderate query traffic and uses minimum memory space whereas the later is suitable for high query traffic but it requires more memory space. We call this part short-term load balancing.

The second level, called mid-term load balancing, is in charge of replicating the role played by a peer r responsible of a query q in neighboring peers when traffic intensity exceeds a threshold value. In this case, peers that are neighbors to peer r in the sense of the DHT protocol, are also allowed to contact the Web search engine to get answers for the query q and pass them back to the requesting peers. The decision on what neighbors to replicate a query q is made by considering the intensity of query traffic arriving from each neighbor to peer r. Neighbors routing more instances of query q to peer r are more likely to get a replica of q, that is they are more likely to become a responsible peer for q.

3.3. Algorithmic details

Each peer has a result cache (RCache) and a special purpose cache called location cache (LCache). The RCache is used to store answers to queries and their respective query terms in a normalized format. This cache stores answers of queries for which the peer is responsible, and answers of queries delivered to users that submitted queries in the peer. Each cache entry has a state flag indicating whether the respective query answer is being retrieved from some other place such as a peer or a Web search engine. Also, in each entry there is a timeout flag indicating the time instant from which the respective query answer is no longer valid. RCache entries are administered with the LRU replacement policy.

The LCache stores data used to support short-term scheduling. Each entry stores terms of a query, an IP address of a peer that contains (or is to contain soon) the answer for the query, and an expiration counter used to limit the number of times the entry is used to route incoming queries with the same terms. These cache entries are also administered with the LRU replacement policy though their overall size is much smaller than in the RCache as they do not store query answers.

In the following we describe the combined operation of RCache and LCache in each peer through an example.

Assume that a peer d is responsible for a query q and that a peer a_1 requests the answer for q. Peer a_1 does not find an entry for q in either its RCache or its LCache so it must get the query answer from peer d. The DHT protocol establishes that to go from a_1 to d it is necessary to pass through peers b and c. In consequence, peer a_1 stores q in its RCache with flag state indicating "answer being retrieved" and it sends q to peer b. Since b is the first DHT hop from a_1, an entry (q, a_1, m_b) is stored in the LCache of peer b where $m_b = m$ is the initial value for the entry expiration counter. Then peer b sends q to peer c, and as peer c does not find a valid entry for q in either its RCache or its LCache, it sends q to peer d. At this point, peer d searches for a valid entry for q in its RCache which, upon a cache miss, it triggers a request for q to the Web search engine. In the mean time, another peer a_2 generates a request for the same query q by sending q to peer b since b is the first DHT hop from a_2 as well. This causes a cache hit in the LCache of peer b which changes the

entry associated with q from (q, a_1, m_b) to $(q, a_2, m_b - 1)$. Then peer b sends the pair (q, c) to peer a_1 where peer c address is included for cases in which peer a_1 contains an invalid entry for q in its RCache and q is not in its LCache either, so the query is sent from peer a_1 directly to peer c to reach peer d if necessary. In either case, an entry (q, a_2, m_a) is stored in the LCache of peer a_1 where $m_a = m_a - 1$ or $m_a = m$ depending on whether there is or not an entry for q already stored in the LCache. On the other hand, once peer d contains a valid answer for q in its RCache, it initiates a data transfer from peer d to peer a_1 to make it possible for peer a_1 to contain a valid answer for q stored in its RCache and respond to the requesting user. In turn, this triggers a data transfer from peer a_1 to peer a_2 so that a_2 can respond to its user and store the answer for q in its RCache.

In this way, the aim of the combined operation of the RCache and LCache objects in each peer is to quickly prevent from the potential saturation of peer d when it suddenly arises an extremely popular query q. Basically the main idea is to take advantage of the fact that it is necessary to transfer query answers to requesting peers which are re-used as replicas by other requesting peers. When the query traffic on peer d becomes beyond a threshold value, it shares the responsibility for q on one or more neighboring peers, such as peer c, to reduce load on d. This peer c is then allowed to contact the search engine to let it hold a valid replica of the answer for query q. Overall, the LRU replacement policy takes care of cache entries that are not longer useful in the RCache and LCache objects.

Algorithms 1 and 2 describe the steps followed by the proposed two-level scheme upon the occurrence of events triggered by the reception of query messages in each peer.

3.4. Replication

Popular objects (or hot objects) may produce overload for the responsible peer which may not be able to handle the amount of requests of a hot object. Our solution attempts to solve this issue by dynamically increasing the number of peers that are responsible for an object. The key idea is to spread the load among the replicas taking into account the current load and capacities of the peers. Unlike previous works, we do not overload the saturated peer by sending the cached objects but sending an small message to authorize the peer to request the object directly from the Web Search Engine.

We define C_i the maximum capacity of a peer i, as the requests per unit of time it is able to *answer*, for example, 100 requests per second. When the number of queries received for a peer i exceeds C_i we say the peer is saturated. Each peer keeps track of the number of times an object k was accessed during a time interval. We call this value f_k, the frequency of k. The load of a peer is the total number of times it has sent a cached object to another peer, which also corresponds to the summation of the frequencies of all the objects it had in its cache in that interval.

In order to take replication decision the maximum capacity C_i and its current load L_i is sent periodically by piggybacking this information in the routing and keep-alive messages. Additionally, each peer stores the query frequency f_{ij} coming from its in-links for each stored object.

Algorithm 3 presents our replication approach. We select a set of *in-links* to replicate considering that (1) the summation of incoming frequencies for q has to be higher than the

Algorithm 1: Event handler in each Peer

```
1  Event: A new query q arrives from a user connected to
2         this peer
3  begin
4      if RCache.find( query= q ) then
5          if RCache.isValid( query= q ) then
6              return RCache.answerQuery( query= q )
7      else if IsResponsible( q ) then
8          r = getAnswerFromSearchEngine( );
9          RCache.insert( query= q, flag="retrieved", answer= r )
10         return r
11     if LCache.find( query= q ) then
12         p = LCache.getPeerIP( query= q )
13     else
14         p = DHTgetNextPeerIP( thisPeer( ) )
15     RCache.insert( query= q, flag="retrieving")
16     send( peer= p, query= q, source= thisPeer( ) )

17 Event: An answer for query q arrives from another peer
18 begin
19     r = getAnswer( q )
20     RCache.update( query= q, flag="retrieved",
21                    answer= r )
22     foreach query e waiting for r do
23         send( peer= sourcePeer( e ), query= e, answer= r )
24     return r
```

Algorithm 2: Event handler in each Peer (cont.)

```
1  Event: A query q arrives from another peer
2  begin
3      if RCache.find( query= q ) then
4          if RCache.isValid( query= q ) then
5              r = RCache.answerQuery( query= q )
6              send( peer= sourcePeer( q ), query= q, answer= r )
7          else if containNextPeer( q ) then
8              send( peer= nextPeer( q ), query= q )
9      else if IsResponsible( q ) then
10         r = getAnswerFromSearchEngine( );
11         RCache.insert( query= q, flag="retrieved", answer= r )
12         send( peer= sourcePeer( q ), query= q, answer= r )
13     else if LCache.find( query= q ) then
14         p = LCache.getPeerIP( query= q )
15         LCache.update( query= q, peerIP= sourcePeer( q ) )
16         c = DHTgetNextPeerIP( thisPeer( ) )
17         send( peer= p, query= q, nextPeer= c )
18     else
19         if firstHopDHT( sourcePeer( q ) ) then
20             LCache.insert( query= q, peerIP= sourcePeer( q ) )
21         if containNextPeer( q ) then
22             send( peer= extraPeer( q ), query= q )
23         else
24             p = DHTgetNextPeerIP( thisPeer( ) )
25             send( peer= p, query= q )
```

load it wants to alleviate, and (2) the total available capacity of the involved peers is enough to answer the number of requests. Notice that in case all *in-links* have not enough available capacity, they have to be replicated further into their own *in-links*.

Algorithm 3: Replication Algorithm

input : Object q: most frequently accessed query in the peer RCache
input : Overload β

```
1 sort (inLinks): high to low f_ij
2 while β > 0 do
3     P_i ← Next in-link
4     Replicate responsability for q in P_i
5     if f_ij < (C_i − L_i) then
6         β = β − f_ij
7     else
8         β = β − (C_i − L_i)
```

3.5. Parallel LCache transfers

For each query in the LCache we can reduce latency when a sudden peak in query traffic takes place. In this case, the number of requesting peers per unit time increases drastically. In such a situation letting sequential transfers of a query answer from one peer to the another can degrade performance since the last requesting peers should experience long latencies. We propose a solution for this case which consists on parallelizing query answer transfers.

For a highly popular query in the LCache we can keep more than one peer ID, say D peer IDs. Thus, for a suddenly popular query q, the first requesting peer p_1 would have to get the query answer from the responsible peer and place its ID in the LCache entry associated with q. The second peer p_2 gets the answer from peer p_1. After this transfer, both peer p_1 and p_2 are in condition to transfer the q answer to a third peer p_3. Assuming that a transfer takes τ units of time, on average, peer p_3 should expect to receive the q answer no earlier than $2 \cdot \tau$ either from p_1 or p_2. For a continuous stream of newly requesting peers, the number of peers available for transfers increases significantly. For instance, for $D = 2$ the number of available peers follows a Fibonacci sequence for the number of available peers able to deliver at τ, $2 \cdot \tau$, ..., $n \cdot \tau$ units of time respectively.

On the other hand, keeping D unbounded does not increase performance after a threshold value since the peer arrival rate λ limits the total number of peers that can be served in parallel. In terms of a $G/G/\infty$ queuing model of this problem, the average number of peers being served at any time is given by $\lambda \cdot \tau$ and thereby, at steady state, it does not make sense to have $D > \lambda \cdot \tau$ in the LCache for q. This indicates the basis of our approach.

We let grow and shrink the value of D in accordance with the observed arrival rate λ. We keep a priority queue to retrieve the peer ID which is able to serve a transfer in the least time.

Algorithm 4 presents our strategy for selecting the next peer to let parallel transfer at the lowest cost in accordance with the current value of D.

Algorithm 4: GetPeer

```
1  LCache request: A new query q arrives from peer R
2  begin
3      t= current_time( )
4      λ= update_rate( t )
5      n = heap.size( )
6      if n = 0 then
7          heap.insert( new( R ), t + τ )
8          return NULL
9      D = λ · τ
10     p = heap.get( )
11     t_p = p.timeout( )
12     if t_p > t then
13         t= t_p
14     if n < D then
15         heap.insert( new( R ), t + τ )
16     if n + 1 < D then
17         heap.insert( p )
18     return p.ID
```

4. Evaluation

We have used a process-driven simulation to conduct our evaluation using the simulation library libcppsim [40]. Libcppsim is a general purpose library written in C++ where each process is implemented by a co-rutine that can be locked and unlocked during simulation. We have built a simulator that implements a transport layer, a P2P overlay and our caching proposal. Pastry [9] has been used as the overlay in our experimentation. We have also simulate the Web Search Engine process, which sends the answers to the responsible peers and other users that may exist outside the P2P community.

In our previous work [7] we used an event-driven simulation over Peersim [41]. However we were not able to have a dynamic query rate with this simulator. In the present work, we therefore use our own simulator to create a more realistic environment with different query rates to simulate flash crowds.

From the state of the art we have chosen two efficient strategies for DHTs to be compared against our approach. The first one, that we call *Leafset*, replicates objects of an overloaded peer in its closest neighbors as suggested in [9] to prevent from saturation. The second approach, that we call *Bubble*, replicates objects of an overloaded peer in all of its in-links

Table 1: Default Configuration

Parameter	Value
Network size (num. peers)	1,000
RCache Size (num. entries)	100
Local Cache Size	50
LCache Size (num. entries)	50

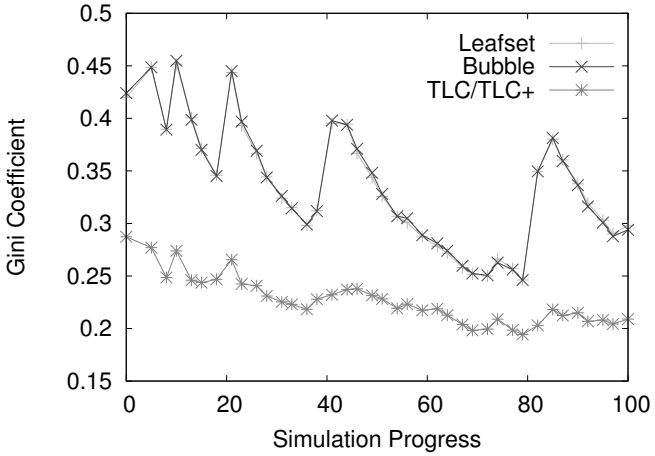


Figure 2: Gini coefficient measuring load balance

[18, 13]. In order to make a fair comparison, we also included a Local Cache to both approaches. These two approaches represent the main ideas from the state of the art.

The approach first presented in [7] is named TLC in figures below and the extended version proposed in this paper is called TLC+. TLC+ has an extension in the LCache mechanism to support parallel transfers.

The default configuration of our simulation experiments is presented in Table 1. The Simulation is divided in simulation time windows of 100 units of time. In our default configuration the query rate is 1000 queries per unit of time, and every 500 units of time we increase the query rate to simulate a flash crowd introducing 5000 queries for 5 units of time.

We have considered the freshness of the query answers by assuming that queries are assigned a time to live (TTL) value by the Web search engine (WSE) after which the cached object become stale. The TTL assigned by the WSE is independent of our cache solution. As in [42], we use the incremental approach to estimate the TTL value for objects cached by the responsible peers. Notice that the WSE does not reveal the TTL values assigned to query answers and thereby peers can only predict them.

Queries were extracted from a US-Canada user query log of a Yahoo! Web search engine. In the following we present our performance results.

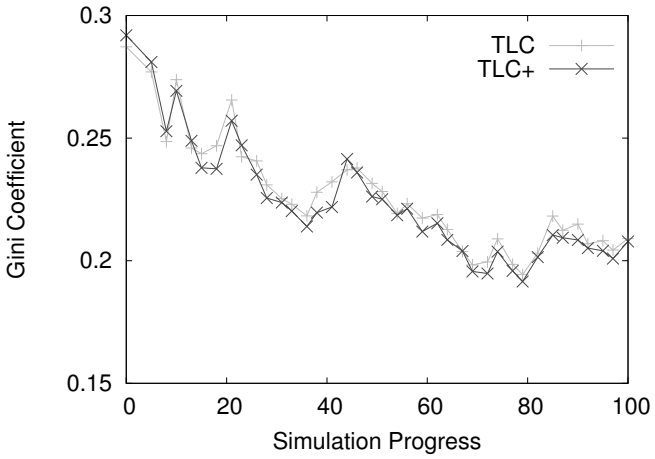


Figure 3: TLC versus TLC+

4.1. Load Balancing

In order to measure load balance among peers we use a metric based on Lorenz curves called the Gini coefficient, which is a metric commonly used on other fields like economics and ecology.

If all peers have the same load, the Lorenz curve is a straight diagonal line, called the line of equality or the perfect load balancing. If there is any imbalance, then the Lorenz curve falls below the line of uniformity. The total amount of load imbalance can be summarized by the Gini coefficient G, which is defined as the relative mean difference, i.e. the mean of the difference between every possible pair of peers, divided by their mean load. For n peers the Gini coefficient is calculated by (1), where l_i correspond to the load of the peer i and μ is the mean load. G values go from 0, when all peers have load equal to 1, when every individual, except one has a load of zero. Therefore, as G comes closer to 0, load imbalances are reduced, whereas, as G comes closer to 1, load imbalances are increased.

$$G = \frac{\sum_{i=1}^{n}(2i - n - 1) \cdot l_i}{n^2 \cdot \mu} \tag{1}$$

Notice that a Gini value equal to 0 is extremely rare in a P2P network. In [23] Pitoura et al. estimate that Gini values under 0.5 represent a fair load imbalance among the peers.

Figure 2 compares the load balance conditions of the network by measuring the Gini coefficient. We compare TLC+, TLC, Leafset and Bubble. The results show that our approach achieves better load balance than the other approaches. This is explained by our short-term scheduling. This strategy improves the hit rate of the local cache, since there is one node strategically chosen that knows its location. We reach an average improvement of 31% during the simulation as compared to the other techniques.

Figure 3 shows the difference between TLC+ and TLC. Both achieve very similar results. There is a 1% of improvement of TLC+ compared to TLC.

Flash crowds can be clearly seen in the gini coefficient curves of the other approaches

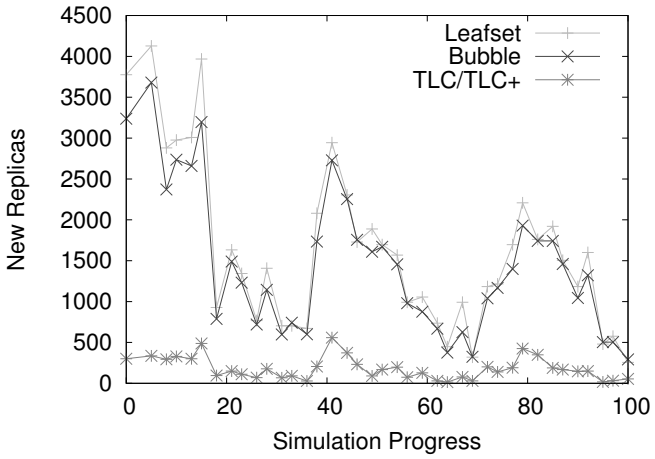


Figure 4: Number of Replicas

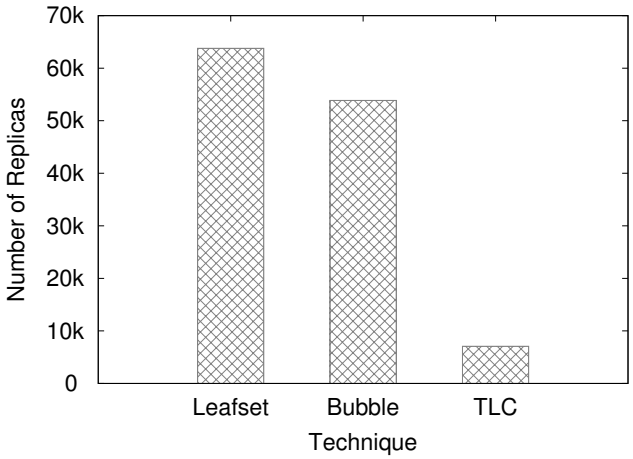


Figure 5: Number of Responsibility Replicas

since the unbalance increases drastically. In our approach, on the other hand, flash crowds do not impact performance significantly.

4.2. Number of Replicas

Another important metric is the number of replicas created in order to cope with peaks in load. The replication of an object in another peer implies the transfer of the whole object and when a peer is already overloaded this can degrade even more the system performance. TLC/TLC+ only replicates the role of the peer of being the responsible for an object, which requires a low cost message but increases traffic outside the network.

Figure 4 compares the number of replicas created during the evolution of the simulation. The number of replicas for TLC are significantly smaller than the number of replicas in other approaches. This improvement can be explained by our short-term scheduling. The LCache greatly reduces the load on the responsible peers by redirecting requests to peers that have

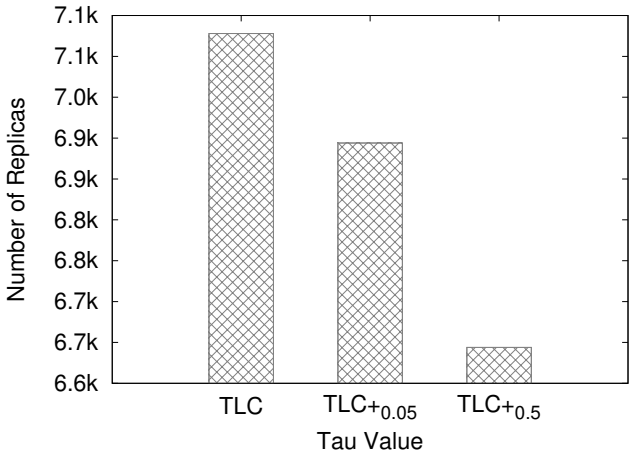


Figure 6: Number of replicas $R > 1$

recently requested the queries and have already received the results. The peaks observed in the number of replicas generated by the other approaches are a consequence of flash crowds queries and stale cached objects, which have to be replicated after the TTL assigned by the replicating peer.

The extension of the LCache structure of TLC to support more than one peer ID, impacts in the number of replicas. The simulations assume a transfer latency between peers of 0.05 time units. Figure 6 shows that the number of replicas decrease even more with the use of TLC+. With this solution more traffic is handled by the LCache structures deviating load to the objects located in the local caches.

Furthermore, solutions like Leafset and Bubble replication do not consider the load of other peers when they perform replication on them. However, if they do the same, that is, to perform replication by transferring responsibility and viceversa for the TLC/TLC+ they can decrease the number of replicas as shown in Figure 5. Our approach still produces less replication.

4.3. Traffic generated

We analyze the traffic generated within the P2P network to respond a query and the traffic generated towards the Web search engine by measuring the cache hits and misses respectively.

A cache hit is when an object is found in the P2P network and there is no need to fetch the object from the Web search engine. Figure 7 presents the evolution of cache hits through simulation progress. TLC/TLC+ improves cache hits, thus, reducing the amount of traffic generated towards the Web search engine. Compared to the best of the other approaches, TLC/TLC+ improves a 23% the number of cache hits when compared against the other approaches. Peaks in the curves are consequence of flash crowds. Improvements of TLC+ over TLC over traffic are not significant.

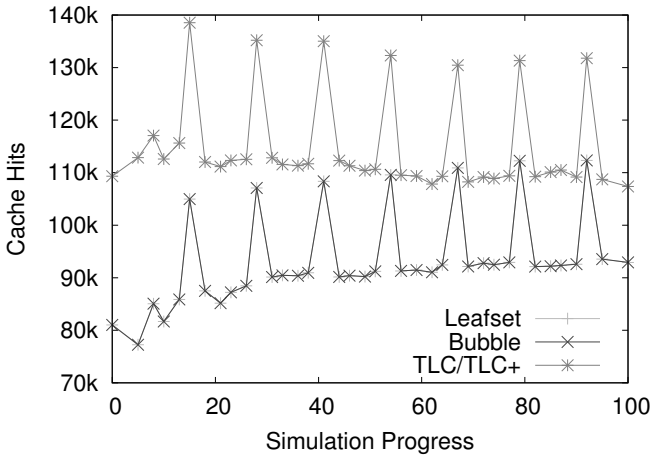


Figure 7: Number of hits

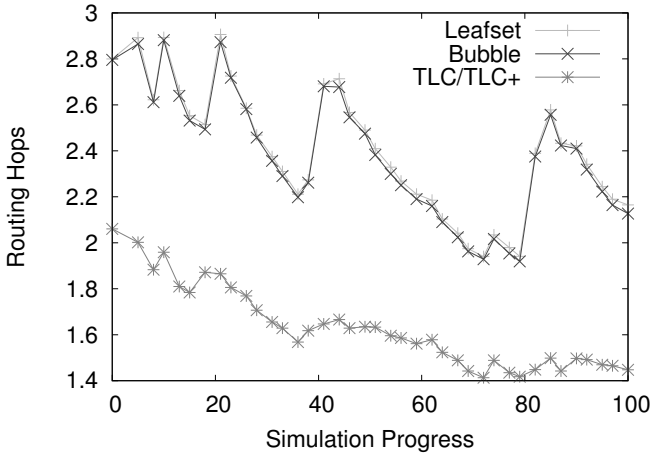


Figure 8: Average Number of hops

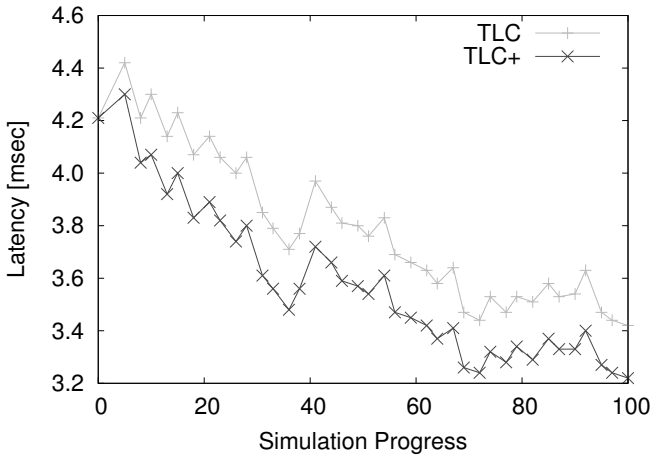


Figure 9: Latency TLC versus TLC+

4.4. Latency

We have measured latency in terms of the number of hops to find a cached object in the P2P network. Figure 8 presents the results. TLC/TLC+ reduces the number of hops due to the introduction of the short-term scheduling which quickly sends the query to a peer that is likely to have the object in its local cache.

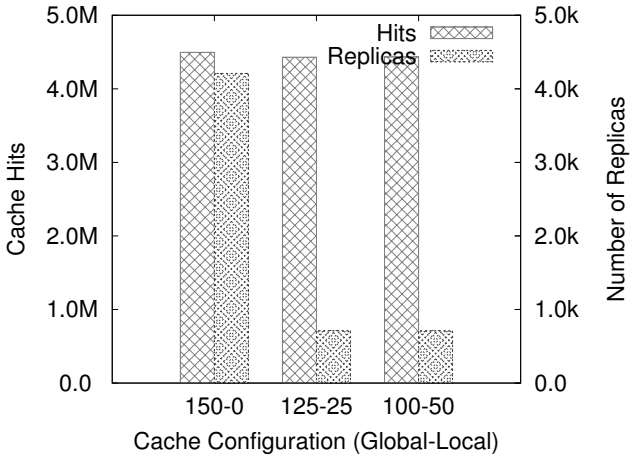
In the beginning of the simulation the number of hops start at a higher point since cache structures are empty. Along the simulation, the curves tend to decrease as the local caches start to contain popular queries and the routing process does not complete for all searches.

The Leafset and Bubble approaches are greatly affected by the TTL expiration since they cannot use copies stored in local caches and arrive to the responsible peer. Our approach, on the other hand, cope smoothly with the TTL expirations and maintain a stable number of hops during the whole process.

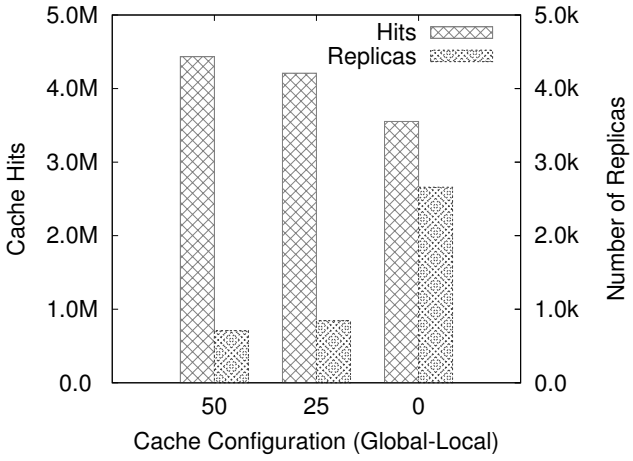
TLC and TLC+ have same results in terms of routing hops. However the TLC+ decreases response times. Figure 9 presents the latency improvement introduced by TLC+. In this case, the LCache enables faster searches since more peers can benefit from different copy objects stored in the local caches at the same time. This is an important result in the context of applications that require low latency for producing responses.

4.5. Cache Size

Figure 10(a) presents our results when simulating different cache size configurations for the global and local caches. The figure shows two measures: the number of cache hits in the left y axis and the number of replicas in the right y axis. The introduction of a minimal percentage of local cache greatly reduces the number of replicas in the system. Moreover, the number of hits also increases with the introduction of the local cache. We did not find much difference between introducing 15%, or 30% of local cache entries in the peers, and the bigger this number, the less cache hits are. Thus, if the local cache takes space to the global



(a) Cache Size



(b) Location Cache Size

Figure 10: Cache Structures Size

cache there are more entries that are not longer within the P2P community. Entries in the local cache, on the other hand, can be found in other peers in the system. We conclude that only a small percentage of local cache must be introduced in order to avoid increasing traffic towards the Web search engine.

We have analyzed different sizes of LCache in our results. Figure 10(b) presents two measures for comparison: the number of cache hits in the left y axis and the number of replicas in the right y axis. We can conclude that a small size for LCache can greatly decrease the number of replicas used in the system while increasing the number of hits.

Without the LCache structure the objects in the local caches are not found and most of the searches arrive to the responsible peer. More searches to the responsible peer produces more overloaded and consequently more replicas are generated.

4.6. Power consumption

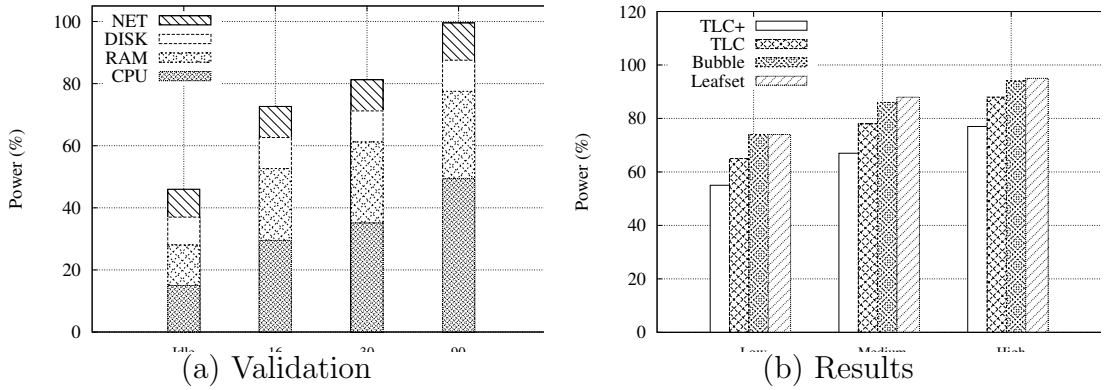
As stated in [5], it is necessary to provide incentive for home users since they will pay the price of the additional power consumption of the set-to-box. The incentives can be in the form of discounts or proportional service rewards. In [5] they estimate that the use of a Video-on-Demand application is about 2.5 kW/hour and the additional energy cost to a single home user is no significant. In our case, we are dealing with Web queries (part of the Web content of the Internet), which is about 5 times smaller than the video content generated on the Internet and consequently use less energy [5].

In order to properly tune the simulator to reflect power consumption values which are close to reality, we set the simulator power consumption curves in accordance with the curves presented in [43]. The results can be seen in Figure 11.a which show the average energy consumed by the different hardware devices. Values are identical to those presented in [43] typically at 0% utilization devices still demand about 50% of peak power.

Figure 11.b shows the impact in reduction of power consumption of the proposed TLC and TLC+ strategies under different workloads. The bars present the results obtained with the power curves from actual devices as in the context of Figure 11.a. Results show that the proposed strategies can lead to a significant reduction of total power consumption.

5. Conclusions

We have proposed a result cache strategy for structured P2P networks called Two-Level Result Caching (TLC+). TLC+ uses a short-term load balance strategy based on the use of a LCache which is a tiny LRU cache that stores information about the peers that recently requested queries in the first hop of the DHT routing path. The LCache increases parallelism in the network by making available to other peers the queries stored in the local cache of peers that recently requested queries. A mid-term load balancing strategy in TLC+ replicates the role of being a responsible peer for a subset of queries in its neighboring peers in terms of the DHT protocol. The replication strategy takes into account the peer capacity and current load. This strategy alleviates the load on the responsible peers caused by very popular objects in the P2P network.



(a) Validation (b) Results

Figure 11: Power consumption results.

The experimental results confirm that our solution improves load fairness among peers by 5% compared to the best performance strategy evaluated. Replication is reduced up to a 96% by our solution when compared to the alternative strategies. Replica generation is a communication bandwidth consuming operation which can saturate peers and increase the cache miss rate degrading performance. Moreover, TLC+ improves a 10% the number of cache hits compared to the best alternative strategy and a 18% compared to the worst performer. This implies that traffic towards the web search engine is reduced by the same factor. The experimentation contemplated flash crowd queries, namely situations in which a sudden peak of a few popular queries are introduced in the simulation. This allows the evaluation of real-life situations where users are reactive to big news events. This is a type of experiment that has not been considered by related work and we have observed that it can degrade performance significantly. The results show that TLC+ is more robust than alternative strategies as it does not get affected by this type of queries.

References

[1] V. G. Costa, J.Lobos, A. Inostrosa-Psijas, M. Marín, Capacity planning for vertical search engines: An approach based on coloured petri nets, in: PETRI NETS 2012, Vol. 7347 of Lecture Notes in Computer Science, Springer, 2012, pp. 288–307.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and zipf-like distributions: evidence and implications, in: IEEE INFOCOM '99, Vol. 1, 1999, pp. 126 –134 vol.1. doi:10.1109/INFCOM.1999.749260.

[3] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, J. Zahorjan, Measurement, modeling, and analysis of a peer-to-peer file-sharing workload, in: SOSP, 2003, pp. 314–329.

[4] N. R. D. Council, Improving the efficiency of television set-top boxes, `http://www.nrdc.org/energy/files/settopboxes.pdf`.

[5] V. Valancius, N. Laoutaris, L. Massoulié, C. Diot, P. Rodriguez, Greening the internet with nano data centers, in: CoNEXT '09, ACM, 2009, pp. 37–48.

[6] J. H. Ahnn, U. Lee, H. J. Moon, Geoserv: A distributed urban sensing platform, in: CCGRID, 2011, pp. 164–173.

[7] N. H. Erika Rosas, M. Marin, Two-level result caching for web search queries on structured p2p networks, in: Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, ICPADS '12, IEEE Computer Society, Washington, DC, USA, 2012.
[8] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: SIGCOMM, 2001, pp. 149–160.
[9] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: Middleware 2001, Vol. 2218 of LNCS, 2001, pp. 329–350.
[10] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, J. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment., IEEE Journal on Selected Areas in Communications 22 (1) (2004) 41–53.
[11] S. Iyer, A. I. T. Rowstron, P. Druschel, Squirrel: a decentralized peer-to-peer web cache, in: PODC, 2002, pp. 213–222.
[12] X. Wang, W. S. Ng, B. C. Ooi, K.-L. Tan, A. Zhou, Buddyweb: A p2p-based collaborative web caching system, in: NETWORKING 2002 Workshops, Vol. 2376 of LNCS, Springer, 2002, pp. 247–251.
[13] T. Stading, P. Maniatis, M. Baker, Peer-to-peer caching schemes to address flash crowds, in: IPTPS '01, Springer-Verlag, 2002, pp. 203–213.
[14] G. Silvestre, S. Monnet, R. krishnaswamy, P. Sens, AREN: a popularity aware replication scheme for cloud storage, in: Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, ICPADS '12, IEEE Computer Society, Washington, DC, USA, 2012.
[15] A. S. Tigelaar, D. Hiemstra, D. Trieschnigg, Search result caching in peer-to-peer information retrieval networks, in: A. Hanbury, A. Rauber, A. P. de Vries (Eds.), IRFC, Vol. 6653 of Lecture Notes in Computer Science, Springer, 2011, pp. 134–148.
[16] T. Fujimoto, R. Endo, K. Matsumoto, H. Shigeno, Video-popularity-based caching scheme for p2p video-on-demand streaming, in: AINA '11, IEEE Computer Society, 2011, pp. 748–755.
[17] J. Kangasharju, K. W. Ross, D. A. Turner, Adaptive content management in structured p2p communities, in: InfoScale '06, ACM, 2006.
[18] Z. Despotovic, Q. Hofstätter, M. Michel, W. Kellerer, An operator approach to popularity-based caching in dhts, in: ICC, IEEE, 2010, pp. 1–6.
[19] W. Rao, L. Chen, A.-C. Fu, G. Wang, Optimal resource placement in structured peer-to-peer networks, Parallel and Distributed Systems, IEEE Transactions on 21 (7) (2010) 1011–1026. doi:10.1109/TPDS.2009.136.
[20] F. Ferrarotti, M. Marín, M. Mendoza, A last-resort semantic cache for web queries, in: J. Karlgren, J. Tarhio, H. Hyyrö (Eds.), SPIRE, Vol. 5721 of Lecture Notes in Computer Science, Springer, 2009, pp. 310–321.
[21] M. Marín, F. Ferrarotti, M. Mendoza, C. Gómez-Pantoja, V. G. Costa, Location cache for web queries, in: D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, J. J. Lin (Eds.), CIKM, ACM, 2009, pp. 1995–1998.
[22] M. Marín, V. G. Costa, C. Gómez-Pantoja, New caching techniques for web search engines, in: S. Hariri, K. Keahey (Eds.), HPDC, ACM, 2010, pp. 215–226.
[23] T. Pitoura, N. Ntarmos, P. Triantafillou, Replication, load balancing and efficient range query processing in dhts, in: EDBT, 2006, pp. 131–148.
[24] G. Swart, Spreading the load using consistent hashing: A preliminary report, in: SPDC '04, IEEE Computer Society, 2004, pp. 169–176.
[25] D. Bauer, P. Hurley, M. Waldvogel, Replica placement and location using distributed hash tables, in: IEEE LCN 2007, IEEE Computer Society, 2007, pp. 315–324.
[26] S. Bianchi, S. Serbu, P. Felber, P. Kropf, Adaptive load balancing for dht lookups, in: ICCCN 2006, 2006, pp. 411 –418.
[27] H. Yamamoto, D. Maruta, Y. Oie, Replication methods for load balancing on distributed storages in p2p networks, in: SAINT '05, IEEE Computer Society, 2005, pp. 264–271.
[28] V. Ramasubramanian, E. G. Sirer, Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays, in: NSDI, 2004, pp. 99–112.

[29] X. Wang, Y. Zhang, X. Li, D. Loguinov, On zone-balancing of peer-to-peer networks: analysis of random node join, SIGMETRICS Perform. Eval. Rev. 32 (2004) 211–222.
[30] B. Godfrey, I. Stoica, Heterogeneity and load balance in distributed hash tables, in: INFOCOM, 2005, pp. 596–606.
[31] Y. Zhu, Y. Hu, Efficient, proximity-aware load balancing for dht-based p2p systems, IEEE Trans. Parallel Distrib. Syst. 16 (2005) 349–361. doi:http://dx.doi.org/10.1109/TPDS.2005.46.
URL http://dx.doi.org/10.1109/TPDS.2005.46
[32] H.-C. Hsiao, H. Liao, S.-T. Chen, K.-C. Huang, Load balance with imperfect information in structured peer-to-peer systems, Parallel and Distributed Systems, IEEE Transactions on 22 (4) (2011) 634–649. doi:10.1109/TPDS.2010.105.
[33] M. Bienkowski, M. Korzeniowski, Friedhelm, Dynamic Load Balancing in Distributed Hash Tables, in: IPTPS, 2005, pp. 217–225.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.1933
[34] G. Giakkoupis, V. Hadzilacos, A scheme for load balancing in heterogenous distributed hash tables, in: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, PODC '05, ACM, New York, NY, USA, 2005, pp. 302–311.
[35] C. Chen, K.-C. Tsai, The server reassignment problem for load balancing in structured p2p systems, Parallel and Distributed Systems, IEEE Transactions on 19 (2) (2008) 234–246. doi:10.1109/TPDS.2007.70735.
[36] H. Shen, C.-Z. Xu, Locality-aware and churn-resilient load-balancing algorithms in structured peer-to-peer networks, Parallel and Distributed Systems, IEEE Transactions on 18 (6) (2007) 849–862. doi:10.1109/TPDS.2007.1040.
[37] J. Ledlie, M. I. Seltzer, Distributed, secure load balancing with skew, heterogeneity and churn., in: INFOCOM, IEEE, 2005, pp. 1419–1430.
[38] D. R. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, in: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, SPAA '04, ACM, New York, NY, USA, 2004, pp. 36–43.
[39] N. Laoutaris, P. Rodriguez, L. Massoulie, Echos: edge capacity hosting overlays of nano data centers, SIGCOMM Comput. Commun. Rev. 38 (1) (2008) 51–54. doi:10.1145/1341431.1341442.
URL http://doi.acm.org/10.1145/1341431.1341442
[40] M. Marzolla, Libcppsim: A simula-like, portable process-oriented simulation library in c++, in: ESM, 2004.
[41] M. Jelasity, A. Montresor, G. P. Jesi, S. Voulgaris, The Peersim simulator, http://peersim.sf.net.
[42] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, O. Ulusoy, Adaptive time-to-live strategies for query result caching in web search engines, in: Proceedings of the 34th European conference on Advances in Information Retrieval, ECIR'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 401–412.
[43] L. A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37.