Cooperative Caching versus Proactive Replication for Location Dependent Request Patterns

Niels Sluijs, Frédéric Iterbeke, Tim Wauters, Filip De Turck, Bart Dhoedt and Piet Demeester
Department of Information Technology (INTEC)
Ghent University – IBBT
Gaston Crommenlaan 8, Bus 201, 9050 Ghent, Belgium
Niels.Sluijs@intec.ugent.be

Abstract

Today's trend to create and share personal content, such as music files, digital photos and digital movies, results in an explosive growth of a user's personal content archive. Managing such an often distributed collection becomes a complex and time consuming task, indicating the need for a personal content management system that provides storage space transparently, is quality-aware, and is available at any time and at any place to end-users. A key feature of such a Personal Content Storage Service (PCSS) is the ability to search worldwide through the dataset of personal files. Due to the extremely large size of the dataset of personal content, a centralized solution is no longer feasible and an interesting approach for an efficient distributed PCSS implementation is to use a structured peer-to-peer network, and more in particular a Distributed Hash Table (DHT), providing a logarithmic lookup performance (i.e. logarithmic in the number of network nodes). In order to further increase the lookup performance, a caching layer is typically used between the application layer and the DHT. These caching strategies are location neutral, and typically do not exploit location dependence of request patterns. In this contribution we present the caching layer and introduce the cooperative Request Times Distance (RTD) caching algorithm and protocol, which uses popularity and distance metrics to increase the lookup performance of requests that exhibit location dependent patterns. We present a systematic analysis of the caching framework and compare the cooperative caching algorithm to the state-of-the-art Beehive replication strategy. The cooperative RTD caching solution shows that when request patterns are more localized, the increase in lookup performance through cooperation is significantly better than Beehive.

1 Introduction

The interaction with digital information plays an important role in our daily life. Different websites, such as YouTube[1] and Flickr[2], offer platforms to store and share personal content (e.g. text documents, music files, digital photos and personal movies). Due to the explosive growth of the user's personal content collections, managing those archives becomes a complex and time consuming task. Nevertheless, end-users expect that they can access and share their personal content from any device, anywhere and at any time. Current systems that offer storage space for personal content fail to achieve this in a scalable and quality-aware way, constraints (e.g. on files sizes and formats) need to be set on the content in order to cope with the workload. To be able to deal with the future workload, a centralized approach is no longer feasible. A Personal Content Storage Service (PCSS) is a networked solution that offers storage space to end-users in a transparent manner, which can be accessed from different types of devices independent of place and time. Figure 1 presents an architectural view on such a distributed content management platform, where users (i.e. clients) are connected to super nodes in the PCSS overlay network.

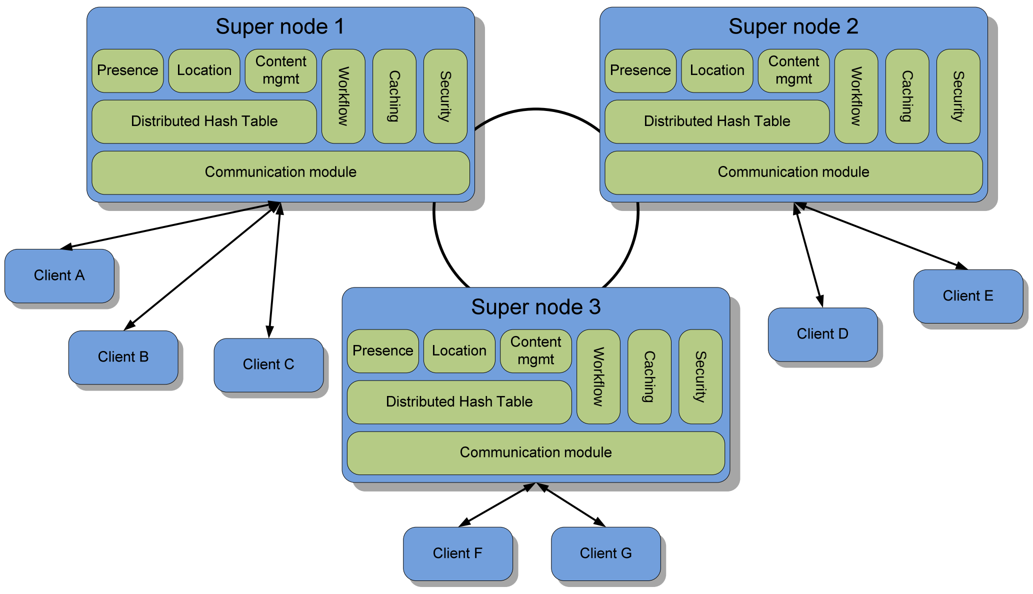

Figure 1: Overview of the Personal Content Storage Service architecture.

The PCSS uses a (hybrid) Peer-to-Peer (P2P) architecture to support all necessary operations and the architecture is split-up in two high-level components: super nodes and clients. The key functions of the super node component (as schematically shown in Figure 1) concern user and content management (including replica management and

[1] http://www.youtube.com/
[2] http://www.flickr.com/

indexing), query handling, presence management, security provisioning, monitoring of the underlying P2P network. The client component is responsible for advertising shared content as well as retrieving and uploading personal content items. For end-users the PCSS acts as a virtual hard disk, as if personal content were accessed using their local file system. Additionally, end-users are relieved from cumbersome back-up issues, since the PCSS provides data integrity through replication.

To efficiently lookup personal content references (i.e. through optimal content indexing), this paper presents a novel cooperative caching strategy that is able to react on location dependent request patterns and making use of an underlying Distributed Hash Table (DHT) infrastructure. A DHT is a (structured) P2P network offering scalable lookup with performance and functionally similar to a traditional hash table data structure. The caching strategy introduced in this paper is a more adequate and detailed explanation of the base algorithm presented in (Sluijs et al., 2009). We make a thorough analysis of all parameters that are of concern for the caching framework and compare the cooperative caching algorithm with a state-of-the-art replication strategy called Beehive (Ramasubramanian and Sirer, 2004). Beehive is based on a analytical model that finds the minimal replication level for each object such that the average lookup performance is a predefined constant number of (overlay) hops. Our solution clearly outperforms Beehive in case of (highly) localized request patterns due to the cooperation between caches.

The article continues in Section 2 with an overview of related work, while Section 3 introduces the caching architecture for the PCSS. Section 4 provides the replication and caching algorithms, and both frameworks are validated and evaluated by simulations in Section 5. Conclusions and future work are presented in Section 6.

2 Related work

Different solutions exist for providing distributed storage of files, ranging from client-server systems (e.g. NFS (Callaghan et al., 1995), AFS (Howard et al., 1988) and Coda (Braam, 1998)) over cluster file systems (e.g. Lustre (Philip Schwan, 2003), GPFS (Schmuck and Haskin, 2002) and the Google File System (Ghemawat et al., 2003)) to global scale Peer-to-Peer (P2P) file systems (e.g. OceanStore (Kubiatowicz et al., 2000), FARSITE (Bolosky et al., 2007) and Pangaea (Saito and Karamanolis, 2002)). However, none of the distributed file system are designed for efficiently handling scattered lookup of personal content items exhibiting locality in their request distributions, which is indeed a feature inherent to personal content.

Query search in *unstructured* P2P networks is done using a query flooding mode, using a TTL (Time-To-Live) mechanism to prevent overloading the network. In order to improve the efficiency of the query flooding model, Wang *et al* describe a distributed caching mechanism for search results in (Wang et al., 2006). However, using the TTL limit implies that personal content stored in such a network has no guarantees to be found, which makes this type of search mechanism less suitable for a PCSS. A data structure that guarantees that (even rare) objects that are stored in a network always can be located is called a *Distributed Hash Table* (DHT). A DHT is a *structured* P2P network that offers scalable lookup, similar to a hash table, where the average number of (overlay) per lookup request has a complexity of $O(\log N)$ with N the number of nodes in the DHT network. Different implementations of a DHT already exists, such as Chord (Stoica et al., 2003), Pastry (Rowstron and Druschel, 2001a), and Tapestry (Zhao et al., 2004). Various research studies have been performed to improve the lookup performance of DHTs. The Beehive (Ramasubramanian and Sirer, 2004) framework enables DHTs to provide an average (i.e. for all stored objects in the DHT) lookup performance of $O(1)$ through proactive replication. According to the evaluation made in (Ramasubramanian and Sirer, 2004), Beehive outperforms the passive caching technique used by Pastry (Rowstron and Druschel, 2001b) in terms of average latency, storage requirements, work load distribution and bandwidth consumption. In passive caching, objects are cached along all nodes on a query path (Ramasubramanian and Sirer, 2004), while Beehive’s replication strategy consists of finding the minimal replication level for each object such that the average lookup performance for the system is a constant C number of hops (Ramasubramanian and Sirer, 2004). Beehive assumes that there is a global power law (or Zipf-like) popularity distribution and requests are uniformly distributed over the network. However, in the scenario of the PCSS it is conceivable that locality exists in request patterns (Duarte et al., 2007), which has a major influence on the performance of a caching algorithm and requires a less expensive solution than Beehive.

In (Bhattacharjee et al., 2003) results of queries are cached and are re-used to answer more detailed queries. In this way unnecessary duplication of work and data movement is avoided. The results of (conjunctive attribute) queries are cached in a view tree and are used later on to resolve queries that contain (parts of) the cached query results. Although the view tree tries to avoid duplication of work and data movement, each search query is issued to the root (node) of the view tree. This aspect prevents successful deployment of a view tree in a PCSS system, since it introduces a single point of failure.

Previous studies on caching techniques (Liu and Xu, 2004) or distributed replica placement strategies for *Content Distribution Networks* (CDN) (Kangasharju et al., 2002; Wauters et al., 2005) show that by taking distance metrics and content popularity into account, a performance increase is obtained compared to more straightforward heuristics such as *Least Recently Used* (LRU) or *Least Frequently Used* (LFU). An even larger performance increase can be obtained by using *cooperative caching* (Chae et al., 2002), compared to independent caching. In cooperative caching it is important to keep track of (neighbor's) cache states and as a result of using neighbor caches the load is more evenly balanced among the nodes, leading to improved system scalability. The proposed caching strategy uses the distance metrics and content popularity, as well as cooperative caching to increase the PCSS lookup performance, where references of content are stored that exhibit locality in the distribution of requests over the network.

3 Caching architecture for DHT performance optimization

Since the dataset of personal content is extremely large and in order to deal with the future workload, a distributed approach to index the personal content collection is a prerequisite for the PCSS. As explained above, a *Distributed Hash Table* (DHT) allows for highly scalable lookup in extremely large distributed datasets. A <*key*, *value*>-pair is stored into the DHT and every node participating in the DHT is able to efficiently locate *values* that correspond to a certain *key*. For the PCSS, the *key* can be a file name, or could alternatively represent tags/keywords describing the personal content item. Often, the *value* represents a link to the location where the content is actually stored. To further optimize the content lookup process, typically a caching layer is introduced on top of the DHT (e.g. Pastry (Rowstron and Druschel, 2001b) and Beehive (Ramasubramanian and Sirer, 2004)). The caching layer is located between the application and DHT layer, and typically stores search results of important requests, as shown in Figure 2.

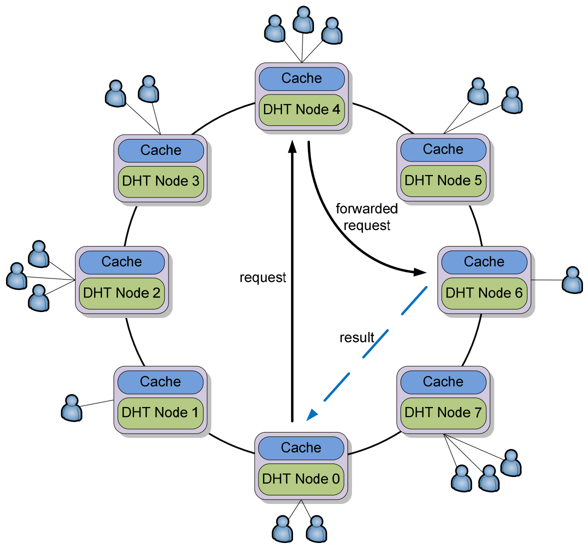

Figure 2: The Personal Content Storage Service enhances the distributed hash table with a caching architecture to increase the lookup performance.

In the example of Figure 2, eight nodes span the Chord-based DHT network for storing references to locations of personal content. In this paper, Chord is used as DHT implementation, in view of its wide spread use and its inclusion in multiple P2P network simulators. The approach taken, however, can be applied to any underlying DHT implementation. By using a *hash* function both content references and nodes can be mapped to a *numeric identifier space*. In Figure 2 we assume that nodes depicted with a higher number have a higher numeric identifier. Each node is responsible for storing *values* belonging to *keys*, which numeric identifier is between the numeric identifier of the preceding DHT-node (excluding) and the numeric identifier of the current node (including). In order to efficiently route messages in a DHT, every node keeps a *finger table*. This finger table maps numeric identifiers to nodes, where the distance between the numeric identifier of the current node and the numeric identifiers in the finger table increases exponentially. In this way, messages are sent to a node at least half the distance of the key space closer to the destination node. When using the same numeric identifier space as the node numbers in Figure 2, the finger table of, e.g., node 0 contains mappings to node 1, 2 and node 4. In this way the average (and worst case) number of hops for a lookup has a complexity of $O(\log N)$, where N is the number of nodes in the DHT network (Stoica et al., 2003).

When a user requests a personal content object in the PCSS, the DHT is used to lookup the link to the location the object is stored. Figure 2 also presents an example of a traditional lookup request, initiated by a user connected to node 0. Node 0 forwards the request to the node in its finger table with the numeric identifier closest to and smaller than the hash value (i.e. node 4), this process is repeated until the target node is reached (i.e. node 6). Finally, the target node replies directly to the requesting node (i.e. node 0). Storing references to object locations into a DHT is performed in a similar way, except no reply message is returned. Since the *value*-part of <*key*, *value*>-pairs are typically locations where (the latest version of) personal content items are stored, no synchronizations need to take place.

To improve the lookup performance, the PCSS provides each node with a relatively small amount of storage space (the cache) to temporarily duplicate <*key*, *value*>-pairs, obtained from lookup results on the DHT. The cache contains a *monitoring service* component for measuring object popularity and for keeping track of neighbor cache information. By storing <*key*, *value*>-pairs on average closer to end-users, the average time (measured in number of hops) needed for a lookup decreases. Another benefit of the caching architecture is that multiple nodes are able to handle lookup request of popular content, which alleviates the hotspot problem (i.e. sudden huge popularity of a limited set of content items) enormously.

4 Cooperative caching and proactive replication mechanisms

In order to utilize the available cache space on each node efficiently, a caching or replication algorithm is mandatory to decide which entry to remove for a more valuable lookup result. The popularity of personal content is typically described by a *power law* (*Zipf-like*) distribution. This distribution states that some personal content is highly popular and the remainder of the content is more or less equally popular. In (1) the Zipf-like probability mass function (Breslau et al., 1999) is provided, where M denotes the number of personal content items and α is the exponent characterizing the distribution.

$$P_{Zipf-like}(x) = \frac{x^{-\alpha}}{\sum_{a=1}^{M} a^{-\alpha}} \tag{1}$$

$P_{Zipf\text{-}like}(x)$ determines the probability that a personal content object having rank x is requested, where $x \in \{1, \ldots, M\}$. This implies that a personal content object having a lower rank (i.e. a larger value for x) is less popular, $\alpha > 0$. Typically for P2P file sharing applications the value of α is between 0.6 and 0.8 (Backx et al., 2002).

In Section 4.1 the analytical model of Beehive's proactive replication strategy is explained in detail and in Section 4.2 our cooperative caching strategy is introduced.

4.1 Beehive: proactive replication

The replication framework of Beehive (Ramasubramanian and Sirer, 2004) enables a DHT to achieve (on average) constant lookup performance for power law (Zipf-like) popularity of stored content. Through proactive replication Beehive reduces the average number of (overlay) hops, where copies of stored content are actively propagated among multiple nodes in the network. The goal of Beehive's replication strategy is to find the maximum replication level L for each object such that the average lookup performance for the system is a predefined constant number of hops (Ramasubramanian and Sirer, 2004). In order to reach this goal, popular items are replicated to more nodes than less popular objects, aiming to minimizing both storage and bandwidth overhead. According to (Ramasubramanian and Sirer, 2004), Beehive is a general replication framework that can operate on top of any DHT implementation using prefix-routing, such as Chord (Stoica et al., 2003). In Chord-based DHT implementations the search space halves in each step of the lookup process (i.e. Chord is a DHT with base 2) and therefore provides $O(\log N)$ lookup performance, where N is the number of nodes in the DHT network. The main idea of Beehive is that the *maximum* number of hops for a lookup is reduced by h hops if objects are proactively replicated to all nodes on all query paths that logically precede the home nodes for the last h hops. A home node is the responsible DHT node for storing an object, according to the numeric identifier produced by the hash function of the *key*. Beehive controls the number of replicas by assigning each stored object a replication level L. The *maximum* number of (overlay) hops, for every node in the DHT, to locate an object on level L equals L. When Chord (i.e. b = 2) is considered, each object replicated at level L is stored on $N/b^L = N/2^L$ nodes, where N is the number of nodes in the DHT. Figure 3 illustrates the replication level mechanism of Beehive.

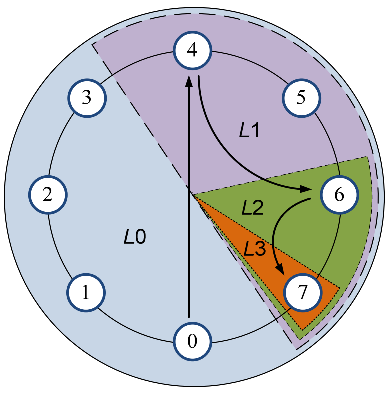

Figure 3: Beehive's level of replication mechanism, the maximum level L for this situation is three. The lower the level for a stored item in the DHT, the more it is replicated to other nodes. The goal is to find the minimal replication level for each item such that the average number of (overlay) hops per lookup is constant.

In Figure 3 a Chord-based DHT network is considered with 8 nodes, which means that the maximum replication level $k = log_2(8) = 3$. All personal content references on level 3 are only stored on the home nodes of the objects. On level 2 personal content item references are replicated to $8/2^2 = 2$ nodes, including the home node. The number of replicas made on level 1 is $8/2^1 = 4$ and level 0 lets all nodes store a replica of the personal content reference. The lookup query inserted at node 0 to lookup a personal content reference that is located on node 7, requires 3 (overlay) hops when the object is only stored on its home node (i.e. the replication level is 3). When the replication level for this object is set to 2, Figure 3 depicts that the number of (overlay) hops is reduced to 2 hops. The (maximum) number of hops can be reduced to one hop, when the object has a replication level of 1. When the number of hops for the objects has to be 0, all nodes store the object (i.e. replication level 0). In this way each stored item in the DHT receives a replication level, based on the popularity of the the item, so that the *weighted average of the maximum* number of hops for a lookup request for a stored item in the DHT matches a predetermined target number *C*.

Let f_i denote the fraction of items replicated at level i or lower (i.e. $f_k = 1$, where k is the maximum replication level). When *M* is the total number of items stored in the DHT, Mf_0 most popular objects are replicated at all nodes. The number of objects that have a *maximum* number of i (overlay) hops per lookup request is $Mf_i - Mf_{i-1}$. The average storage (i.e. average number of objects stored) on a node for a DHT implemenation with base *b* is expressed whith the following equation (Ramasubramanian and Sirer, 2004):

$$Mf_0 + \frac{M(f_1 - f_0)}{b} + \cdots + \frac{M(f_k - f_{k-1})}{b^k} \tag{2}$$

When $Q(m)$ represents the total number of lookup requests to the most popular *m* items, the number of queries that travel a *maximum* of i (overlay) hops is $Q(Mf_i) - Q(Mf_{i-1})$. The target number of hops *C* is reached when the folling expression is fullfilled on the *weighted average of the maximum number of (overlay) hops*:

$$\sum_{i=1}^{k} i \cdot \frac{Q(Mf_i) - Q(Mf_{i-1})}{Q(M)} \leq C \tag{3}$$

Note that the target number of hops *C* is the weighted average of the *maximum* number of (overlay) hops for a lookup request for a stored item in the DHT and not the average number of (overlay) hops as considered in (Ramasubramanian and Sirer, 2004). Finally, assume that in the optimal solution the problem $f_0 \leq f_1 \leq \cdots \leq$

$f_{k'-1} < 1$, for some $k' \leq k$. In (Ramasubramanian and Sirer, 2004) this leads, using equation (3), to the following closed-form solution that minimizes the (average) storage requirement but satisfying the target number of hops *C*:

$$f_i^* = \left[\frac{d^i \cdot (k' - C')}{1 + d + \cdots + d^{k'-1}}\right]^{\frac{1}{1-\alpha}}, \forall 0 \leq i < k' \tag{4}$$

Where $d = b^{\frac{1-\alpha}{\alpha}}$, $C' = C \cdot \left(1 - \frac{1}{M^{1-\alpha}}\right)$ and α is the parameter describing the (Zipf-like) popularity distribution. The value of *k'* can be derived by satisfying the condition that $f_{k'-1} < 1$, that is, $\frac{d^{k'-1} \cdot (k' - C')}{1+d+\cdots+d^{k'-1}} < 1$. All $f_i^* = 1, \forall k' \leq i \leq k$. With the closed-form solution of (4) the fraction f_i is approximated by f_i^*, to achieve the desired constant lookup performance and *k'* represents the upper bound for the worst case number of (overlay) hops for a lookup request.

Since the analytical model of Beehive provides the optimal solution to increase the lookup performance, we use the analytical model to compare it with our cooperative caching strategy. In the experiments we have assumed that the popularity of items is known, such that the replication level can be set for all items. This approach allows to investigate the performance after warm-up of the system.

4.2 RTDc: cooperative caching

An important concept for the caching algorithm is that locality exists in the request patterns of nodes inserting lookup requests. This idea is supported by the research performed by Duarte *et al* in (Duarte et al., 2007), where geographical characterizations of requests patterns are studied for YouTube content. However, until now no concrete and generalized probability mass function has been proposed (either based on theoretical or experimental grounds) that describes the locality based request distribution. Here, we model locality using a *Normal* function, where the mean is μ and σ the standard deviation.

Let $P_{Normal}(y)$ be the probability that a personal content item is requested from node *y*. Parameter μ represents the uploading (super) node, since it is conceivable that the (super) node that inserts the personal content object has the highest probability to request it. The value σ is used to model the locality of requests over the network. A higher value of σ makes the distribution more uniform, since more neighboring nodes will request the personal content item.

Basic DHTs use hash functions to map nodes onto the numeric identifier space, which means that nodes are more likely to have different neighbors in the DHT than in the actual network topology. Different research studies are already performed that address the issue of including physical neighboring nodes as logical neighbors in DHTs (Castro et al., 2003; Weiyu et al., 2008), in order to reduce latencies in overlay hops. In (Weiyu et al., 2008) the network topology is embedded into the DHT by assigning a locality-aware identifier to each node. In our use case, we assume that the DHT is locality-aware, neighbors in the PCSS overlay network map to neighboring nodes in the physical network.

Since we want to reduce the average number of hops needed for a lookup, the caching algorithm we propose reacts on both *popularity* and *distance* of lookups. The popularity $p_{n,a}$ represents the total number of requests to an object a, initiated by node n. The distance $d_{n,a}$ of a personal object a is measured by the number of (overlay) hops needed to obtain the lookup result from the requesting node n and the responsible node storing the object. The importance $I_{n,a}$ for node n to store object a, which is used as a metric in the *Request Times Distance* (RTD) caching algorithm, is calculated as:

$$I_{n,a} = p_{n,a} \times d_{n,a} \tag{5}$$

Consequently, the references to personal content objects with the highest importance values for $I_{n,a}$ in (5), will be stored in the local cache of node n. In (Sluijs et al., 2009) the RTD caching algorithm is extended with a sliding window in order to react on changes in content popularity. By adding a sliding window, only the last T requests that arrived in a node are used to determine the popularity of the requested content. However, to compare the caching algorithm with the analytical model of Beehive the sliding window size is set to infinite, since the popularity distribution of the stored content is constant during each simulation run.

Since in a Chord-based DHT each node knows its predecessor and successor node on the DHT ring (to be able to update finger tables when nodes suddenly join or leave the DHT network) the performance of the caching algorithm can be increased by keeping a local copy of both neighbors' cached *keys*. In order to keep the storage overhead to a minimum, only keys of both direct neighbors are kept locally. This *cooperative caching* strategy utilizes the neighbors' caches to virtually increase the size of the local cache. In this way, nodes can avoid storing the same copies of <*key*, *value*>-pairs that can be retrieved from their neighbor, in only one hop. Figure 4 visualizes the *update protocol* for the three possible scenarios of performing a lookup using cooperative caching. In all scenarios the destination node for the lookup is node 6 (i.e. the node responsible for storing the *values* belonging to the search *key*), the request is initiated from node 0 and the node numbers are used as the numeric identifier space. Figure 4a considers the case where the local copies of the cache entries of the neighbor nodes do not contain the search key of object *a*. The scenario in Figure 4b describes the case that the local copy of the cache entry of the neighbor node, in this case node 1, contains the search key. And Figure 4c represents the scenario where node 0 wrongly assumes that node 1 caches the search key of object *a* (i.e. in between cache update messages).

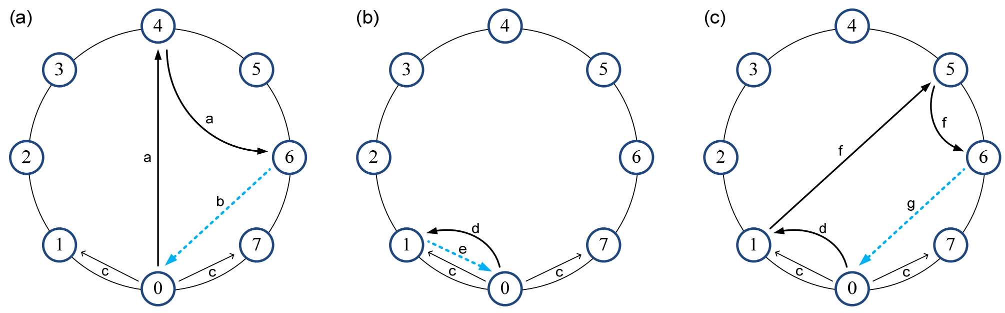

Figure 4: Three scenarios for a lookup using cooperative caching. Scenario (a) describes the case where local copies of neighbor cache entries do not contain the search key. In scenario (b), one of the local copies of the neighbor's cache contains the search key and in scenario (c) the situation that the requesting node wrongly assumes that its neighbor's caches the search key is depicted.

When the local copy of the neighbor's cache does not contain the search key (Figure 4a), the lookup is performed as usual. A request message REQ (a) is routed via node 4 to node 6. Node 6 responds by sending the requested value using a reply message REP (b). In the case that node 0 decides to cache the lookup value, it updates the local cache table of both its neighbor nodes with the cache update message CACHE (c). These nodes then re-compute their values of the importance $I_{1,a}$ and $I_{7,a}$ of object *a*, as the distances $d_{1,a}$ and $d_{7,a}$ are now equal to one hop. No extra lookup delay is introduced by this update mechanism.

In the case that one (or both) of the local copies of the neighbor's caches contain the search key, the lookup request is routed to that neighbor node. In Figure 4b the local copy in node 0 of the cache entries of node 1 has the search key, so the request message REQ (d) is forwarded to node 1. When node 1 still has the value of the search key in its cache, it updates the popularity $p_{1,a}$ and responds the value using the reply message REP (e). Node 0 again decides whether or not to cache locally the lookup value, in the case node 0 keeps the lookup results in its local cache it uses the cache update message CACHE (c) to inform the neighbors.

The situation that node 1 no longer caches the value of the search key and has not sent the corresponding cache update message CACHE to its neighbors yet (i.e. it very recently released the value), is illustrated in Figure 4c. The lookup message REQ (d) is forwarded by node 1 as usual using the request message REQ (f) via node 5 to node 6. Node 6 responds with the value of the search key, using the reply message REP (g). Similar to the other two scenarios, node 0 decides whether or not to store the result in its cache by computing the importance $I_{0,a}$ of object a (with distance $d_{0,a} = 1$ if the entry is stored in its neighbor's cache), and informs the neighbors in case of a cache change with the cache update message CACHE (c). Only in the case when a neighbor is contacted erroneously because it very recently released the requested value, one extra hop is added to the lookup delay. In all other cases, no extra delay is introduced.

To illustrate the inner working of the RTD caching algorithm, Figure 5 presents the pseudo-code for the methods that describe the initiation of a lookup request, receiving a lookup reply and receiving a cache update message. Other routines used by the DHT are not changed by the RTD algorithm.

```
A.01  initiateLookupRequest(key) {
A.02        if(storedOrCachedOnThisNod(key) {
A.03              return lookup_result;
A.04        } else if(neighborCachesKey(key)) {
A.05              sendLookupRequest(neighbor, key);
A.06        } else {
A.07              targetNode = hash(key);
A.08              sendLookupRequest(targetNode, key);
A.09        }
A.10  }

B.01  receiveLookupRequest(key) {
B.02        if(storedOrCachedOnThisNode(key)) {
B.03              sendLookupResult(key);
B.04        } else {
B.05              targetNode = hash(key);
B.06              sendLookupRequest(targetNode, key);
B.07        }
B.08  }

C.01  receiveLookupReply(key, value) {
C.02        updateCounters(key);
C.03        if(storedOrCachedOnThisNode(key)) {
C.04              return;
C.05        }
C.06        lowest_importance_value =
C.07                    getLowestImportanceValueofCachedKeys();
C.08        lookup_importance_value = calculateImportanceValue(key);
C.09        if(lookup_importance_value > lowest_importance_value) {
C.10              removed_key = removeLowestImportanceValueKeyFromCache();
C.11              insertNewKeyIntoTheCache(key, value);
C.12              updateCacheChangeToNeighbors(removed_key, key);
C.13        }
C.14  }

D.01  receiveCacheChangeUpdateOfNeighbor(removed_key, key) {
D.02        neighbor_cache.remove(removed_key);
D.03        neighbor_cache.add(key);
D.04  }
```

Figure 5: Pseudo-code of the method that handles the reply messages REP of lookup requests.

When a user initiates a lookup request, the method on line A.01 of Figure 5 is invoked. When the node is already storing (or caching) a local copy of the lookup result itself (A.02), the result is returned directly to the user (A.03). Otherwise, the node checks whether a neighbor caches the lookup result (A.04) and, if so, the request is then sent to that neighbor (A.05). In the case that result is not stored or cached locally, and not available through a neighbor's cache, the request is sent as a traditional DHT lookup (A.08) by using the hash function (A.07) to determine the target node.

Upon receiving a lookup request (B.01), the node replies (B.03) the result to the requesting node when the node is storing or caching the lookup result (B.02). In the case the node is not storing or caching the lookup result (in the situation of Figure 4c), the request is forwarded (B.06) as usual to the target node (determined by using the hash function (B.05)).

The node initiating the lookup request receives the lookup reply through method C.01. First, the counters that measure the popularity of objects and the distance (i.e. overlay hop count) needed to obtain the lookup request are updated (C.02), ensuring that the importance values are calculated correctly. In the case where the node already stores or caches the lookup result, no further actions need to take place (C.03). Otherwise, the entry in the local cache having the lowest importance value is retrieved (C.06) and the importance value of the lookup result is calculated (C.08) using equation (5). When the importance value of the lookup result is larger than the lowest importance value (C.09), the entry having the lowest importance value is evicted from the cache (C.10) and replaced by the lookup result (C.11). Finally, the neighbors are updated of the local cache change (C.12) using the cache update message CACHE.

When a node receives the cache update message CACHE (D.01), the node removes the old entry (D.02) from the local copy of the specific neighbor's cache entries and adds the new neighbor's cache entry into the local copy (D.03).

5 Evaluating cooperative caching with proactive replication

In order to compare the (cooperative) RTD caching algorithm to the analytical model of Beehive, the discrete-event simulator PlanetSim (Pujol Ahulló et al., 2009) is used. PlanetSim offers a framework to simulate large scale overlay services and networks, such as DHTs, and can be extended at the network, overlay or application layer. For the validation and evaluation of caching algorithms, we have extended PlanetSim at the application layer with a lookup service that can use the RTDc caching algorithm or Beehive's replication model. An advantage of PlanetSim is that it already has an implementation of the DHT lookup protocol Chord (Stoica et al., 2003). For each simulation the DHT network is created by PlanetSim and randomly selected <*key*, *value*>-pairs are inserted into the network, so that every personal content reference is initially stored on only one node. When the replication strategy of Beehive is used, all objects are replicated into the caches according to the analytical model of Beehive. When the RTDc (cooperative RTD) caching algorithm is used, the sizes of the caches are calculated according to the analytical model of Beehive and are left empty. To initialize (i.e. warm-up) the network properly for RTDc, a startup phase is used where search queries enter the network using the cooperative caching algorithm to decide which lookup result to cache. After the whole network is initialized properly, search queries are made by the peers according to the popularity and locality distribution. The simulation stops when the network reaches a non-equilibrium steady state, i.e. when the average number of hops and the cache hit ratio have stabilized. In order to cancel out noise due to random fluctuations, the average values over ten independent simulation runs are used.

In Section 5.1 both algorithms are compared using the traditional uniform distribution of requests over the DHT network. The distribution of lookup requests for personal content retrieval is expected to be more localized, therefore Beehive is compared with RTDc using the locality in lookup requests in Section 5.2. Finally, section 5.3 makes a more detailed study of the RTDc caching algorithm, addressing the message overhead of the update protocol and the temporal behavior of the RTDc caching framework.

5.1 Comparing Beehive with RTDc for traditional distribution of lookup requests

The analytical model of Beehive is used to calculate the replication factor for each personal content reference. The solution that Beehive proposes aims to minimize the storage space (i.e. number of personal content references stored), while offering a predetermined average number of (overlay) hops per lookup request, where the distribution of lookup requests over the network is expected to be uniform. As explained in Section 4.1, the target hops C of Beehive is the weighted average of the maximum number of (overlay) hops per item stored in the DHT. Figure 6 illustrates the relation between the average number of (overlay) hops in relation to the average storage space on a node. Three different curves are presented in Figure 6: Beehive calculated analytically (the dots representing the simulated results of the weighted average of the *maximum* number of hops), the simulated average number of hops for Beehive and simulated average number of hops for the cooperative RTD caching algorithm. In order to compare our caching framework with Beehive, all input parameters are set beforehand and used for each personal content item to determine in which replication level it belongs. In the simulation setup the network size N is set to 32 nodes, the number of personal content items M is $50 \times N$, the power law (Zipf-like) popularity distribution parameter α is 0.6. In order to get representative results for the weighted average of the maximum number of hops for each personal content object, a random personal content item is selected and is requested by all nodes in the DHT network.

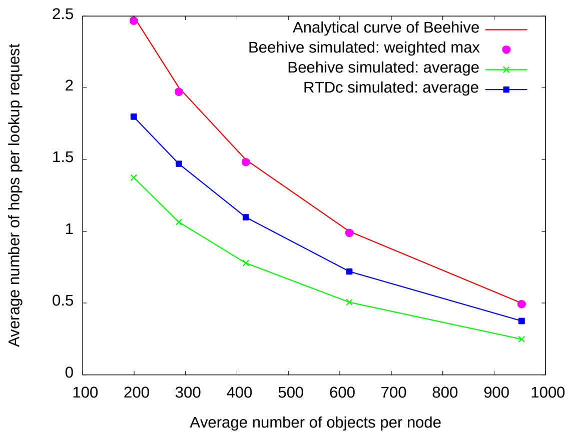

Figure 6: Number of hops per lookup request in relation to the average storage per node, for Beehive analytically calculated with dots representing simulated results of the weighted average of the maximum number of hops, simulated average number of hops for Beehive and simulated average number of hops for the cooperative RTD caching algorithm.

As shown in Figure 6, for relatively small caches, Beehive outperforms the RTDc considerably. This due to the fact that each node in the DHT makes independent estimations of the popularity distribution. When the cache space is relatively small, small mistakes in the estimations of the most important content have a high impact on the performance of the caching algorithm. As explained in Section 4.1, the output of Beehive's analytical model is the weighted average of the maximum number of overlay hops per stored item. Therefore, the simulated weighted average of the maximum number of hops is plotted onto the analytical curve, which shows the correct working of the simulation framework. In order to compare the average number of (overlay) hops for RTDc and Beehive, the simulation results of the average number of hops per lookup request for Beehive are used in the remainder of the article.

Figure 7 depicts the number of nodes storing a replica for each personal content item for both the Beehive and RTDc strategies, where the personal content items are sorted by their popularity rank (i.e. the smaller the rank, the more popular the personal content object). The same simulation results are used as for Figure 6 (i.e. $N = 32$, $M = 50 \times N$ and $\alpha = 0.6$), the target average number of hops C for Figure 7a is set to 1.0 and for Figure 7b to 2.0.

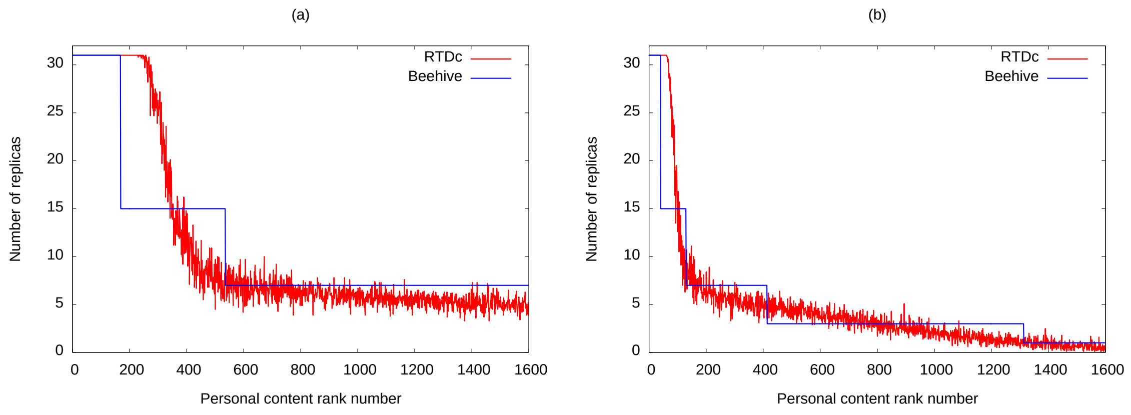

Figure 7: Relation between the number of caches that store a personal content reference and the popularity rank of the personal content reference for the both the model of Beehive as the cooperative RTD caching algorithm. A lower rank indicates a higher popularity and in (a) the target number of hops C is set to 1.0 and in (b) to 2.0.

Figure 7 shows that Beehive replicates a larger fraction of popular content items to more nodes, in order to decrease the average number of hops. When the same amount of storage space is provided to the RTDc caching framework, the RTDc caching algorithm also tries to cache popular content more often. However, the performance increase of RTDc compared to Beehive is lower (as depicted in Figure 6), since RTDc also caches a lot of relatively unpopular content. Note that the analytical model of Beehive has a perfect centralized view on content popularity beforehand.

5.2 Comparison between RTDc and Beehive for distributed lookup of personal content

In Figure 8 the influence on the performance when introducing locality of lookup requests on RTDc is shown, when the simulation has reached the non-equilibrium steady state situation. Since the performance of Beehive is not affected by the locality distribution of lookups, the average number of hops curve of Beehive (see Figure 6) is plotted as a reference. In order to get results for a larger P2P network the network size N is scaled to 256 nodes, the number of personal content items M is $50 \times N$, the power law (Zipf-like) popularity distribution α is 0.6 and the locality parameter σ ranges from 1.0 to 10.0.

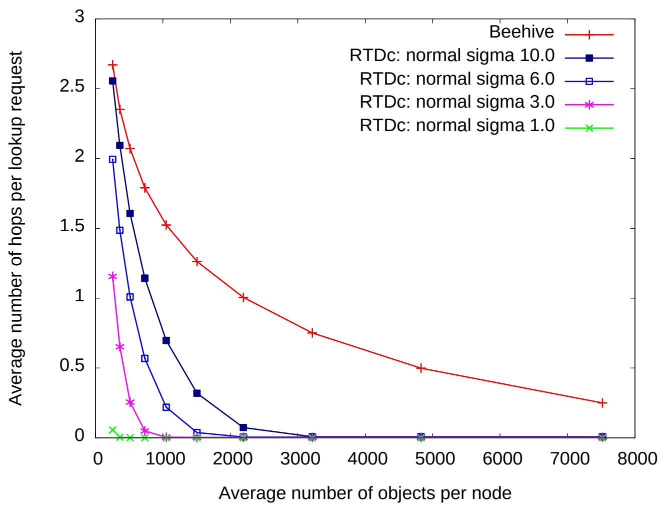

Figure 8: Influence on the performance of introducing locality in lookup request for RTDc with different values for the locality variance parameter σ. Since the performance of Beehive is not affected by the locality distribution of lookups, the curve representing the average number of hops for Beehive (of Figure 6) is plotted as a reference.

Figure 8 illustrates that a higher locality in the request pattern (i.e. a lower value of the locality variance parameter σ) increases the lookup performance when RTDc is used. When the cache space is relatively low, a more localized requests distribution induces a relatively higher performance gain.

5.3 Detailed evaluation of the RTDc caching algorithm

In this section the RTDc caching algorithm is evaluated in more detail in terms of message overhead, and more specifically overhead generated by the update protocol described in Section 5.3.1. This is done by inspecting the fraction of lookups that uses cooperative information versus standard lookup requests in Section 5.3.2. In addition, Section 5.3.3 examines the dynamic behavior of RTDc. Section 5.3.4 investigates the content of caches in terms of popularity.

5.3.1 Message overhead of the update protocol for cooperative caching

Although the main goal of the caching framework is to reduce the average number of hops required to obtain a lookup result, the message overhead created by keeping cache states of neighbors up-to-date should be as a low as possible. Therefore, the average number of messages sent (and forwarded) for a lookup request is shown in Figure 9, in relation to the network size (Figure 9a and Figure 9b) and the cache size (Figure 9c and Figure 9d). For these simulations the network size N is set to 256 nodes, the cache size is 10 items, the number of personal content items

M is 50 × *N*, the power law (Zipf-like) popularity distribution *α* is 0.6 and the locality parameter *σ* is set to 1.0 and 3.0.

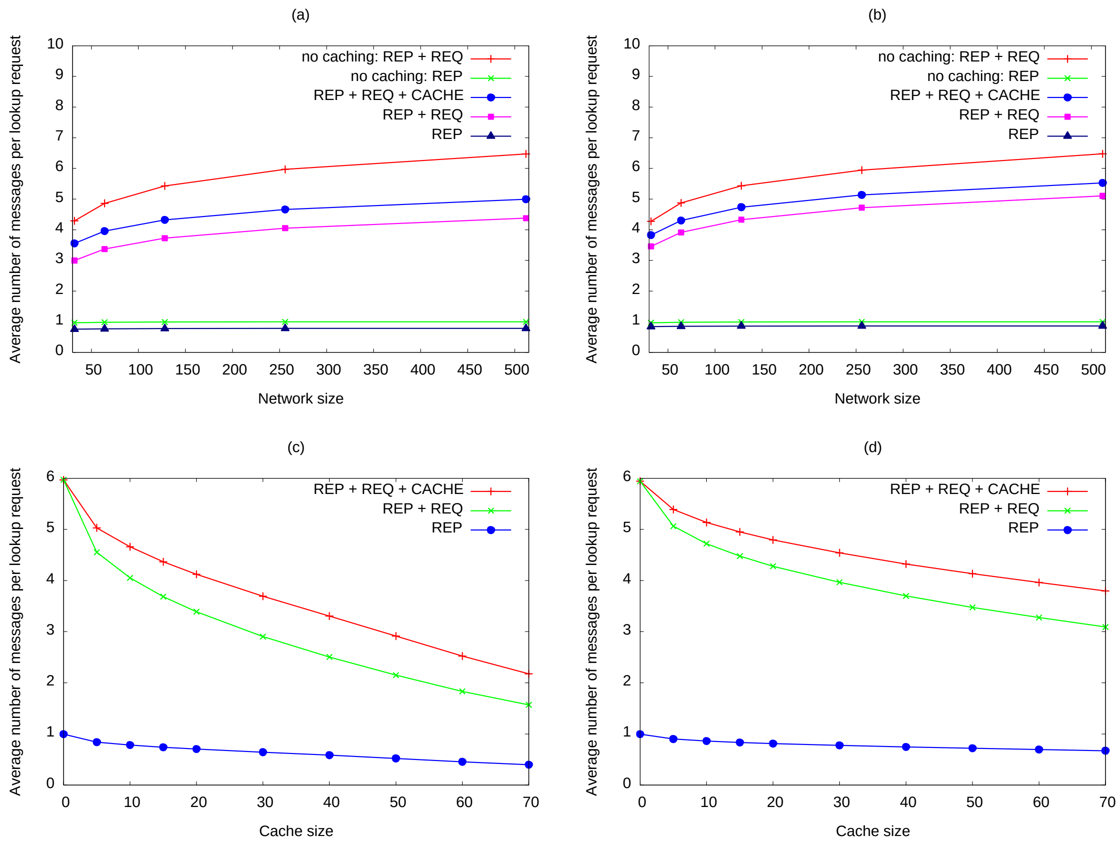

Figure 9: Average message overhead for a lookup request in relation to the network size (a: *σ* = 1.0 and b: *σ* = 3.0) and the cache size (c: *σ* = 1.0 and d: *σ* = 3.0). The message overhead is measured by the number of reply messages REP, request messages REQ en cache update messages CACHE. The cache size for (a and b) is set to 10 entries per node and is compared to the situation no caching is used. The network size of (c and d) is set to 256 peers.

In Figure 9a (*σ* = 1.0) and Figure 9b (*σ* = 3.0) the message overhead is depicted in relation to the network size, for the situation where no caching is used and the case that every node has a cache size of 10 items. The average amount of reply messages for a lookup request is (close to) one (i.e. only requests for items located on the requesting node need no lookup reply message REP) and independent of the network size, for the situation where no caching is used. When caching is enabled, the average number of reply messages REP per lookup request further reduces, since cache hits on the requesting node need no reply message as well and is still independent of the network size (i.e. the total number of items and cache size increases linearly with the network size). When the caching framework is used, both Figure 9a and Figure 9b show that the average number of sent and forwarded messages to obtain a lookup result (REP plus REQ messages) decreases significantly compared to the situation caching is disabled. The total cost (measured in terms of average number of messages for each lookup request, including cache update messages CACHE) of the cooperative RTD caching algorithm in Figure 9a and Figure 9b is considerably less than the total average cost for a lookup request when no caching is used, for all network sizes. This implies that the cooperative caching framework is able to efficiently update cache states to neighbors, without introducing extra network overhead.

When the cache size increases, Figure 9c ($\sigma = 1.0$) and Figure 9d ($\sigma = 3.0$) illustrate that the average number of reply and request messages decreases, since more objects are cached at (multiple) nodes. The message overhead created by the update protocol slightly increases when the cache size increases (i.e. the required number of cache update messages CACHE), because multiple items of similar popularity are stored in the same cache, which results in more cache changes taking place. However, the increase in the average number of cache update messages CACHE in Figure 9c and Figure 9d is much smaller than the decrease in average number of reply and request (REP plus REQ) messages and therefore has no negative impact on the performance of the caching algorithm. The benefits of using the cooperative caching via the cache update protocol are higher than the cost that is introduced to keep neighbor cache states up-to-date.

5.3.2 Fraction of lookup request using cooperative versus standard lookup

In (Sluijs et al., 2009) we show that using the update protocol to inform neighbors of cache state changes results in a performance surplus for the RTD algorithm, since the average number of cache duplicates between neighbors is reduced significantly. To understand this performance increase better, Figure 10 depicts the fraction of lookup requests that use cooperative information and the fraction performing standard DHT lookups. The same simulation setup is used as for Figure 9 (i.e. $N = 256$, $M = 50 \times N$, $\alpha = 0.6$, and σ is 1.0 and 3.0).

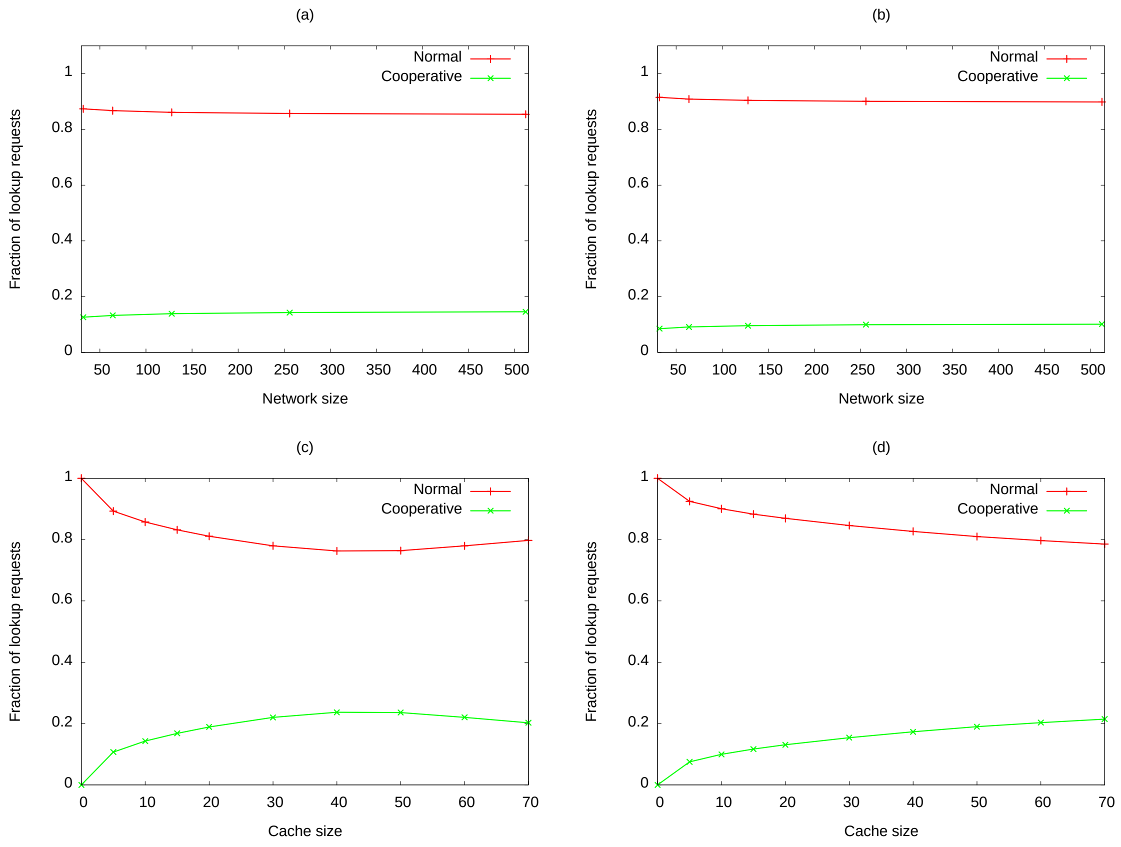

Figure 10: The fraction of lookup requests that is performed using standard lookup and the fraction using cooperative information is plotted as a function of the network size (a: $\sigma = 1.0$ and b: $\sigma = 3.0$) and the cache size (c: $\sigma = 1.0$ and d: $\sigma = 3.0$). The cache size for (a and b) is set to 10 items per node and the network size of (b and c) is set to 256 peers.

Figure 10a ($\sigma = 1.0$) and Figure 10b ($\sigma = 3.0$) indicate that when the network size increases, the fraction of lookup requests using cooperative information (i.e. a neighbor caches the result of the lookup request) is stable. However, when the cache size increases, Figure 10c ($\sigma = 1.0$) and Figure 10d ($\sigma = 3.0$) illustrate that the fraction of lookup requests using cooperative information initially increases and then slightly decreases. The increase can be explained by the fact that nodes get more space available to cache lookup results that are also requested often by their neighbors. When the cache space increases even further, all nodes can store those lookup results themselves and therefore the fraction of lookup requests using cooperative information decreases. In all simulations, the lookup requests that use cooperative information successfully find the result at their neighbor (i.e. scenario (c) of Figure 4 did not occur during the simulation experiments).

5.3.3 Temporal behavior of the RTDc caching framework

Another important aspect of the caching framework is the behavior of cache changes over time. In Figure 11 the number of local cache changes over the last 10 lookup requests are shown as a function of the number of lookup requests initiated. For each simulation run 10 nodes are randomly chosen that record the moments their cache changes, finally, the averages are taken over all nodes together (over ten independent simulation runs). The same simulation setup is used as for Figure 9 (i.e. $N = 256$, $M = 50 \times N$, $\alpha = 0.6$), with locality variance σ 1.0 and 3.0, and a cache size of 10 items for each node.

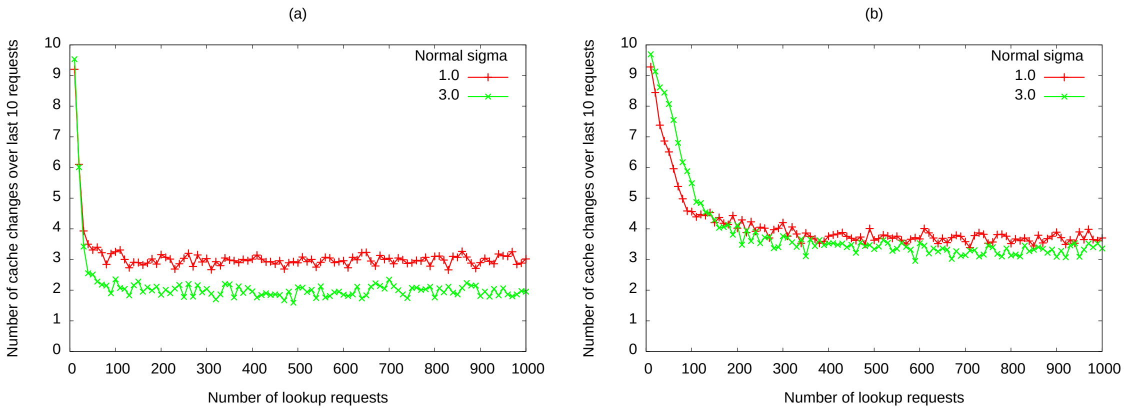

Figure 11: Average number of cache changes (over the last 10 requests) in relation to the number of lookup requests initiated locally, the cache size is set in (a) at 10 entries and in (b) at 50 entries.

Figure 11 shows that the average number of cache changes for the first 10 lookup requests is higher when σ is lower. During initialization, when σ is larger the chance that a neighboring node initiates a lookup request for a local item is larger, therefore the chances are lower that the same item is requested from the local node. Due to this lower probability, the average number of cache changes in the first 10 lookup requests are larger when the value σ is higher (i.e. the caches start empty, which means that the first 10 distinct requests result in a cache change) and thus also requires more lookup request to get a (more or less) stabilized cache change rate.

When reaching steady state, cache changes regularly take place, with a higher average number of cache changes for a smaller value of σ and a larger cache size. When σ is smaller the probability that neighbors request items that are popular for a node is smaller (i.e. the distribution of requests gets more localized), and therefore neighbor's caches can be used less often which requires nodes themselves to decide whether to cache an item. A larger cache size (Figure 11b) indicates that more lookup results are stored locally that have similar importance values. Therefore, the average number of cache changes over the last 10 lookup requests is higher for a larger cache size.

To examine the relative importance of replaced cache entries, Figure 12 depicts the normalized fraction of cache removals for each personal content item stored in the PCSS. On the x-axis, the personal content items are ranked according to their afterwards calculated rank number (i.e. rank 0 is the locally most important object). The same simulation results are used as for Figure 11 (i.e. $N = 256$, $M = 50 \times N$ and $\alpha = 0.6$).

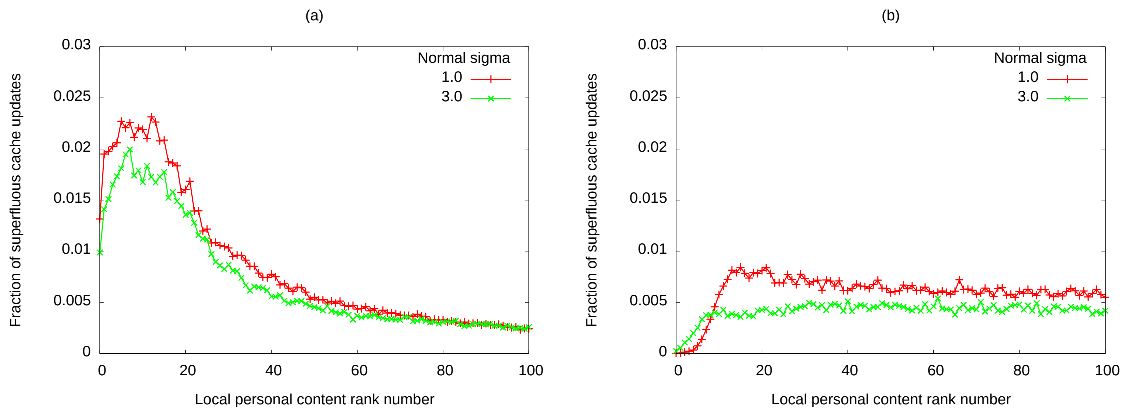

Figure 12: Normalized fraction of cache removals in relation to the local personal content rank number. In (a) the fraction of cache removals is shown for a cache size of 10 items and in (b) the cache size is set to 50 entries.

The optimal solution is to cache the most locally important items at all times (i.e. with highest values for $I_{n,a}$). By measuring the frequency that a specific item is located in the cache, the importance of the item for that node can be established (automatically taking cached items of neighbors into account). Figure 12 depicts, as expected, that the locally most important personal content items are removed less often from the cache than the objects that are just slightly less popular. The reason that the fraction of cache removals decreases when the local popularity rank decreases further, is that those less important items are sometimes accidentally cached and are removed almost instantly. The chance that a less important object is requested decreases according to the personal content rank number and therefore the number of cache removals decreases. When the lookup pattern is more location dependent, the fraction of cache removals is more concentrated on the locally important personal content items, since the probability that a less important item is requested is lower (i.e. a lower value of locality variance σ indicates a higher fraction of lookup requests of locally more important items). Figure 12b shows that when the cache size increases, the most important items are not removed at all once they are stored locally in a cache. Additionally, cache removals are also more evenly distributed over all items, since there is more cache space to store (more or less) equally important lookup results. Most of the cache changes involve two lookup results having a roughly equal importance value.

5.3.4 Fraction of nodes caching locally popular personal content items

The fraction of nodes caching a specific item in its final cache state is depicted in Figure 13, the location variance σ is set to 1.0 (a) and σ is set to 3.0 (b). For each node the local theoretical importance rank is calculated by multiplying $P_{Zipf\text{-}like}(x)$ with $P_{Normal}(x)$, where x is an item stored in the DHT (i.e. the popularity rank does not take neighbor values into account). For each item the availability of that item at one (or both) of the neighbors is also measured, when the specific item is not stored in the local cache. The same simulation parameters are used as for Figure 11 (i.e. $N = 256$, $M = 50 \times N$ and $\alpha = 0.6$) with a cache size of 10 entries at each node.

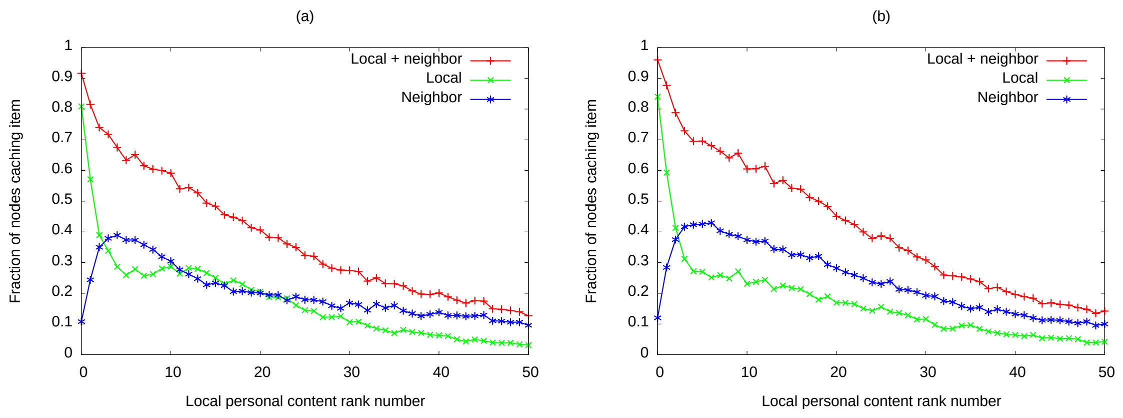

Figure 13: Relation between the average fraction of nodes (and their neighbors) caching an object and the objects, ordered by their local rank (i.e. the smaller the rank number, the more popular the object is on a node locally). Part (a) of the figure shows the simulation results for locality variance σ set to 1.0 and (b) the results for locality variance σ set to 3.0.

Figure 13 shows that on average more than 80% of the nodes are storing the locally most popular object into their cache. More than 20% of the nodes store on average their top 10 locally most popular items. In the case that a node does not store a top 10 item in its cache, chances are relatively high that one (or both) of the neighbors is caching that particular object. When the locality of lookup requests decreases (i.e. the value of locality variance σ increases), the chance increases that a neighbor caches a specific item that is not available from the local cache. For the situations depicted in Figure 13, more than 60% of all nodes is able to obtain their top 10 local popular items within one (overlay) hop.

6 Conclusion and future work

In order to successfully deploy a *Personal Content Storage Service* (PCSS), it has to provide storage space to end-users *transparently*, with *small access times*, and available at *any place* and at *any time*. One of the main features of a PCSS is the ability to search through the dataset of personal files. To optimize searching times in a PCSS, we introduced a caching solution on a *Distributed Hash Table* (DHT). The scalability of a DHT is increased by using the *cooperative Requests Times Distance* (RTDc) caching algorithm.

The RTDc caching framework is compared to the state-of-the-art proactive replication framework Beehive. When the lookup request distribution over the nodes (participating in the DHT) is uniform, the analytical model of Beehive provides a better performance increase compared to our RTDc caching solution. However, the analytical model of Beehive has a perfect centralized view on the content popularity beforehand and therefore no performance is lost by making small mistakes when estimating the popularity parameters. Furthermore, it is highly conceivable that lookup requests are localized (i.e. popularity of objects is different for each node). Unlike the RTDc caching framework, Beehive has no mechanism to take advantage of the locality pattern. When locality exists in the request distribution of lookup request, the RTDc caching algorithm outperforms Beehive quickly. Besides the comparison between the RTDc caching algorithm and Beehive, this paper also presents a more detailed evaluation of RTDc's inner working. We show that the message overhead caused by the update protocol to enable cooperative caching is acceptable, since the performance increase (i.e. reduction of the average number of hops needed for a lookup request) is higher than the cost the update protocol introduces. Besides that, the simulation results show that more than 20% of the nodes store on average their top 10 locally most popular items. When a node does not store a top 10 item in its cache, the chances are relatively high that one (or both) of the neighbors is caching that particular item.

Although the proposed solution optimizes the scalable lookup in a DHT, it can only be used for lookup when the exact name of the *key* is known. This deterministic search property introduces limitations on the suitability of using a DHT for a PCSS. However, the performance of any existing DHT-based framework offering multiple keyword and range queries can already be increased by the proposed framework. Nevertheless, we plan for further research to focus on optimizing DHTs by enabling multiple keywords and range query searches, since currently no solution exists that fulfils all needs for a PCSS. An issue not addressed in this article is that by reducing the time it takes to obtain content locations does not imply that the actual content itself can be accessed quickly. Therefore, we plan to investigate caching/replication algorithms for personal content itself, in order to allow fast access of personal content by using a PCSS.

7 Acknowledgments

The research of Niels Sluijs is funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Tim Wauters acknowledges the F.W.O.-V. (Fund for Scientific Research – Flanders) for its support through a postdoctoral fellowship. Part of this work has been funded by the IBBT PeCMan project.

8 References

Backx P, Wauters T, Dhoedt B and Demeester P, "A comparison of peer-to-peer architectures", in *Conference proceedings of Eurescom 2002 Powerful Networks for Profitable Services,* Heidelberg, Germany: 2002, 1-8.

Bhattacharjee B, Chawathe S, Gopalakrishnan V, Keleher P and Silaghi B, "Efficient Peer-To-Peer Searches Using Result-Caching", in *2nd International Workshop on Peer-to-Peer Systems,* Berkeley, CA, USA.: 2003, 1-6.

Bolosky WJ, Douceur JR and Howell J, "The Farsite project: a retrospective", in *SIGOPS Operating Systems Review*, Volume: 41, no. 2, 2007, 17-26.

Braam PJ, "The Coda Distributed File System", in *Linux Journal*, Volume: 1998, no. 50, 1998, 46-51.

Breslau L, Cao P, Fan L, Phillips G and Shenker S, "Web Caching and Zipf-like Distributions: Evidence and Implications", in *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99),* Volume 1, New York, NY, USA: 21-3-1999, 126-134.

Callaghan B, Pawlowski B and Staubach P, "NFS Version 3 Protocol Specification", 1995.

Castro M, Druschel P, Hu Y and Rowstron A, "Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks", in *Future Directions in Distributed Computing - Lecture Notes in Computer Science,* 2003, 103-107.

Chae Y, Guo K, Buddhikot MM, Suri S and Zegura EW, "Silo, rainbow, and caching token: schemes for scalable, fault tolerant stream caching", in *Ieee Journal on Selected Areas in Communications*, Volume: 20, no. 7, 2002, 1328-1344.

Duarte F, Benevenuto F, Almeida V and Almeida J, "Geographical Characterization of YouTube: a Latin American View", in *Latin American Web Conference (LA-WEB 2007),* 2007, 13-21.

Ghemawat S, Gobioff H and Leung S-T, "The Google File System", in *19th ACM Symposium on Operating Systems Principles,* 2003.

Howard JH, Kazar ML, Menees SG, Nichols DA, Satyanarayanan M, Sidebotham RN et al, "Scale and performance in a distributed file system", in *ACM Transactions on Computer Systems*, Volume: 6, no. 1, 1988, 51-81.

Kangasharju J, Roberts J and Ross KW, "Object replication strategies in content distribution networks", in *Computer Communications*, Volume: 25, no. 4, 1-3-2002, 376-383.

Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Geels D, Gummadi R et al, "OceanStore: An Architecture for Global-Scale Persistent Storage", in *International Conference on Architectural Support for Programming Languages and Operating Systems,* 12-11-2000, 190-201.

Liu J and Xu J, "Proxy caching for media streaming over the Internet", in *IEEE Communications Magazine*, Volume: 42, no. 8, 2004, 88-94.

Philip Schwan, "Lustre: Building a File System for 1,000-node Clusters", in *Proceedings of the Linux Symposium,* 23-7-2003, 380-386.

Pujol Ahulló J, García López P, Sànchez Artigas M and Arrufat-Arias M, "An extensible simulation tool for overlay networks and services", in *Proceedings of the 2009 ACM symposium on Applied Computing,* 2009, 2072-2076.

Ramasubramanian V and Sirer EG, "Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays", in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation,* Volume 1, San Francisco, California, USA: 2004, 1-14.

Rowstron AIT and Druschel P, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems", in *Lecture Notes in Computer Science - Middleware 2001,* Volume: 2218/2001*,* 2001a, 329-350.

Rowstron AIT and Druschel P, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", in *Proceedings of the eighteenth ACM symposium on Operating systems principles,* Banff, Canada: 2001b, 188-201.

Saito Y and Karamanolis C, "Pangaea: a symbiotic wide-area file system", in *Proceedings of the 10th workshop on ACM SIGOPS European workshop,* 2002, 231-234.

Schmuck F and Haskin R, "GPFS: A Shared-Disk File System for Large Computing Clusters", in *Proceedings of the FAST 2002 Conference on File and Storage Technologies,* 28-1-2002, 231-244.

Sluijs N, Wauters T, De Turck F, Dhoedt B and Demeester P, "Caching Strategy for Scalable Lookup of Personal Content", in *First International Conference on Advances in P2P Systems 2009,* 2009, 19-26.

Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F et al, "Chord: A scalable peer-to-peer lookup protocol for Internet applications", in *Ieee-Acm Transactions on Networking*, Volume: 11, no. 1, 2003, 17-32.

Wang C, Xiao L, Liu YH and Zheng P, "DiCAS: An efficient distributed caching mechanism for P2P systems", in *Ieee Transactions on Parallel and Distributed Systems*, Volume: 17, no. 10, 2006, 1097-1109.

Wauters T, Coppens J, Dhoedt B and Demeester P, "Load balancing through efficient distributed content placement", in *Next Generation Internet Networks,* 18-4-2005, 99-105.

Weiyu W, Yang C, Xinyi Z, Xiaohui S, LinCong, Beixing D et al, "LDHT: Locality-aware Distributed Hash Tables", in *International Conference on Information Networking,* 2008, 1-5.

Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD and Kubiatowicz JD, "Tapestry: A resilient global-scale overlay for service deployment", in *Ieee Journal on Selected Areas in Communications*, Volume: 22, no. 1, 2004, 41-53.