RelaxDHT: a churn-resilient replication strategy for peer-to-peer distributed hash-tables[1]

Sergey Legtchenko, Sébastien Monnet, Pierre Sens and Gilles Muller
LIP6/UPMC/CNRS/INRIA

DHT-based P2P systems provide a fault-tolerant and scalable means to store data blocks in a fully distributed way. Unfortunately, recent studies have shown that if connection/disconnection frequency is too high, data blocks may be lost. This is true for most of the current DHT-based system's implementations. To deal with this problem, it is necessary to build more efficient replication and maintenance mechanisms. In this paper, we study the effect of churn on PAST, an existing DHT-based P2P system. We then propose solutions to enhance churn tolerance and evaluate them through discrete event simulation.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; *Maintenance*; D.4.5 [**Operating Systems**]: Reliability—*Backup procedures*; *Fault-tolerance*; E.5 [**Files**]: —*Backup/recovery*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Churn, distributed hash tables (DHT), fault tolerance, peer-to-peer (P2P), replication

1. INTRODUCTION

Distributed Hash Tables (DHTs), are distributed storage services that use a structured overlay relying on key-based routing (KBR) protocols [Rowstron and Druschel 2001b; Stoica et al. 2003]. DHTs provide the system designer with a powerful abstraction for wide-area persistent storage, hiding the complexity of network routing, replication, and fault-tolerance. Therefore, DHTs are increasingly used for dependable and secure applications like backup systems [Landers et al. 2004], distributed file systems [Dabek et al. 2001; Busca et al. 2005], multi-range query systems [Schmidt and Parashar 2004; Gupta et al. 2003; Chawathe et al. 2005], and content distribution systems [Jernberg et al. 2006].

A practical limit in the performance and the availability of a DHT relies in the variations of the network structure due to the unanticipated arrival and departure of peers. Such variations, called *churn*, induce at least performance degradation, due

[1]This work is an extended version of [Legtchenko et al. 2009]

Primary author contact information:
Sergey Legtchenko - LIP 6 - 4 place Jussieu - 75005 Paris - France.
Sergey.Legtchenko@lip6.fr.
+33 1 44 27 88 17.

to the reorganization in the set of the replicas of the affected data that consumes bandwidth and CPU cycles, and at worst the loss of some data. In fact, Rodrigues and Blake have shown that using classical DHTs to store large amounts of data is only viable if the peer lifetime is in the order of several days [Rodrigues and Blake 2004]. Until now, the problem of churn resilience has been mostly addressed at the peer-to-peer (P2P) routing level to ensure the reachability of peers by maintaining the consistency of the logical neighborhood, *e.g.*, the leafset, of a peer [Rhea et al. 2004; Castro et al. 2004]. At the storage level, avoiding data migration is still an issue when a reconfiguration of the peers has to be done.

In a DHT, each peer and each data block is assigned a key (*i.e.*, an identifier). A data block's key is usually the result of a hash function performed on the block. Each data block is associated a *root* peer whose identifier is numerically the closest to its key. The traditional replication scheme uses the subset of the root's leafset containing the closest logical peers to store the copies of a data block [Rowstron and Druschel 2001b]. Therefore, if a peer joins or leaves the leafset, the DHT enforces the placement constraint on the closest peers and may migrate many data blocks. In fact, it has been shown that the cost of these migrations can be high in terms of bandwidth consumption [Landers et al. 2004].

This paper proposes RelaxDHT, a variant of the leafset replication strategy designed to tolerate higher churn rates than traditional DHT protocols. Our goal is to avoid data block migrations when the desired number of replicas is still available in the DHT. We relax the "logically closest" placement constraint on block copies and allow a peer to be inserted in the leafset without forcing migration. Then, to reliably locate the block copies, the root peer of a block maintains replicated localization metadata. Metadata management is integrated to the existing leafset management protocol and does not incur substantial overhead in practice.

We have implemented both PAST and RelaxDHT, on top of PeerSim [Jelasity et al. 2008]. The main results of our evaluations are:

—We show that RelaxDHT achieves higher data availability in presence of churn, than the original PAST replication strategy. For a connection or disconnection occurring every six seconds (for 1000 peers) our strategy looses three times less blocks than the PAST's one.

—We show that our replication strategy induces less unnecessary block transfers than the PAST's one.

—If message compression is used, our maintenance protocol is more lightweight than the maintenance protocol of PAST.

The rest of this paper is organized as follows. Section 2 first presents an overview of the basic replication schemes and maintenance algorithms commonly used in DHT-based P2P systems, then their limitations are highlighted. Section 3 introduces RelaxDHT, an enhanced replication scheme for which the DHT's placement constraints are relaxed so as to obtain a better churn resilience. Evaluation of this algorithm is presented in Section 4. Section 5 presents an analysis of RelaxDHT maintenance cost before Section 6 concludes with an overview of our results.

2. BACKGROUND AND MOTIVATION

DHT based P2P systems are usually structured in three layers: 1) a routing layer, 2) the DHT itself, 3) the application that uses the DHT. The routing layer is based on keys for peer identification and is therefore commonly qualified as *Key-Based Routing* (KBR). Such KBR layers hide the complexity of scalable routing, fault tolerance, and self-organizing overlays to the upper layers. In recent years, many research efforts have been made to improve the resilience of the KBR layer to a high churn rate [Rhea et al. 2004]. The main examples of KBR layers are Pastry [Rowstron and Druschel 2001a], Chord [Stoica et al. 2003], Tapestry [Zhao et al. 2004] and Kademlia [Maymounkov and Mazieres 2002]. The DHT layer is responsible for storing data blocks. It implements a distributed storage service that provides persistence, fault tolerance and can scale up to a large number of peers. DHTs provide simple `get` and `put` abstractions that greatly simplify the task of building large-scale distributed applications. PAST [Rowstron and Druschel 2001b] and DHash [Dabek et al. 2004] are DHTs respectively built on top of Pastry [Rowstron and Druschel 2001a] and Chord [Stoica et al. 2003]. Thanks to their simplicity and efficiency, the DHTs became standard components of modern distributed applications. They are used in storage systems supporting multi-range queries [Schmidt and Parashar 2004; Gupta et al. 2003; Chawathe et al. 2005], mobile *ad hoc* networks [Zahn and Schiller 2006], aswell as massively multiplayer online gaming [Varvello et al. 2009], or robust backup systems [Landers et al. 2004].

In the rest of this section we present replication techniques that are used to implement the DHT layer. Then, we describe the related work that considers the impact of churn on the replicated data stored in the DHT.

2.1 Replication in DHTs

In a DHT, all the peers are arranged in a logical structure according to their identifiers, commonly a ring as used in Chord [Stoica et al. 2003] and Pastry [Rowstron and Druschel 2001a] or a d-dimensional torus as implemented in CAN [Ratnasamy et al. 2001] and Tapestry [Zhao et al. 2003].

A peer possesses a restricted local knowledge of the P2P network, *i.e.*, the leafset, which amounts to a list of `L` close neighbors in the ring. For instance, in Pastry the leafset contains the addresses of the `L/2` closest neighbors in the clockwise direction of the ring, and the `L/2` closest neighbors counter-clockwise. Periodically, each peer monitors its leafset, removing peers that have disconnected from the overlay and adding new neighbor peers as they join the ring.

In order to tolerate failures, each data block is replicated on `k` peers which compose the *replica set* of a data block. Two protocols are in charge of the replica management, the initial placement protocol and the maintenance protocol. We now describe existing solutions to implement these two protocols.

2.1.1 *Replica placement protocols.* There are two main basic replica placement strategies, leafset-based and multiple-key-based:

Leafset-based replication. The data block's root is responsible for storing one copy of the block. The block is also replicated on the root's closest neighbors in a subset of the leafset. The neighbors storing a copy of the data block may be either immediate

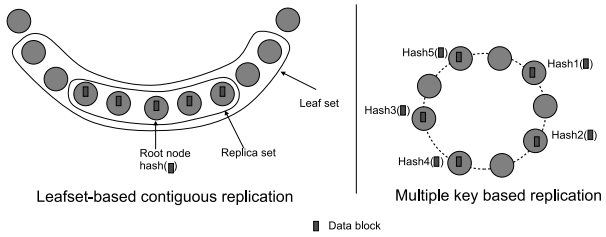


Fig. 1. Leafset-based and multiple-key-based replication ($k = 5$).

successors of the root in the ring as in DHash [Dabek et al. 2004], immediate predecessors or both as in PAST [Rowstron and Druschel 2001b]. Therefore, the different copies of a block are stored *contiguously* in the ring as shown by Figure 1.

Multiple key replication. This approach relies on computing `k` different storage keys corresponding to different root peers for each data block. Data blocks are then replicated on the `k` root peers. This solution has been implemented by CAN [Ratnasamy et al. 2001] and Tapestry [Zhao et al. 2003]. GFS [Ghemawat et al. 2003] uses a variant based on random placement to improve data repair performance. *Path* and *symmetric replication* are variants of multiple-key-based replication [Ghodsi et al. 2005; Ktari et al. 2007].

Lian *et al.* propose a hybrid stripe replication scheme where small objects are grouped in blocks and then randomly placed [Lian et al. 2005]. Using an analytical framework, they show that their scheme achieves on near-optimal reliability. Finally, several works have focused on the placement strategies based on availability of nodes. Van Renesse [van Renesse 2004] proposes a replica placement algorithm on DHT by considering the reliability of nodes and placing copies on nodes until the desired availability is achieved. To this end, he proposes to track the reliability of each node such that each node knows the reliability information about each peer. In FARSITE [Adya et al. 2002], dynamic placement strategies improve the availability of files. Files are swapped between servers according to the current availability of these latter. With theses approaches, the number of copies can be reduced. However, such approaches may lead to a high unbalanced distribution whereby highly available nodes contain most of the replicas and can become overloaded. Furthermore, such solutions are complementary, and taking nodes-availability into account could be done on top of RelaxDHT.

2.1.2 *Maintenance protocols.* The maintenance protocols have to maintain `k` copies of each data block without violating the initial placement strategy. It means that the `k` copies of each data block have to be stored on the root-peer-contiguous neighbors in the case of the leafset-based replication scheme and on the root peers in the multiple-key-based replication scheme.

The leafset-based maintenance mechanism is based on periodic information exchanges within the leafsets. Leafset-based maintenance mechanisms have good scalable properties: the number of messages of the protocol does not depend on the number of data blocks managed by a peer, but only on the leafset size. Yet, even if the overall quantity of maintenance data linearly grows with the number of

blocks, it is possible to efficiently aggregate and compress the data. For instance, in the fully decentralized PAST maintenance protocol [Rowstron and Druschel 2001b], each peer sends a bloom filter to all the peers in its leafset. The bloom filter is a compact probabilistic data structure containing the set of block identifiers stored by the peer [Broder et al. 2002]. When a leafset peer receives such a request, it uses the bloom filter to determine whether it stores one or more blocks that the requester should also store. It then answers with the list of the keys of these blocks. The requesting peer can then fetch the missing blocks listed in all the answers it receives. Notice that the maintenance interval at the KBR layer is much smaller than the maintenance interval of blocks (for instance in PAST the default value of leafset maintenance interval is 1 minute whereas the data block interval is set to 10 minutes).

In the case of the multiple-key-based replication strategies, the maintenance has to be done on a "per data block" basis. For each data block stored in the system, it is necessary to periodically check if the different root peers are still alive and are still storing a copy of the data block. This replication method has the drawback that a maintenance message has to be sent to each root of each data block, which means that the number of messages linearly grows with the number of blocks. Therefore, it seems impossible to aggregate maintenance information in order to optimize its propagation ,*e.g.*, by compressing the messages like in leafset-based replication. For backup and file systems that may store up to thousands of data blocks per peer, this is a severe limitation.

2.2 Impact of the churn on the DHT performance

A high churn rate induces consequent changes in the P2P network, and the maintenance protocol must frequently adapt to the new structure by migrating data blocks. While some migrations are mandatory to restore `k` copies, some others are only necessary to enforce placement invariants.

For example, as a new peer joins the system, if its identifier is closer to a block's key than the identifier of the current block's root, the data block needs to be migrated on the new peer. A second example occurs in leafset-based replication if a peer joins the DHT between two nodes storing replicas of the same block. One of the replicas then needs to be migrated on the new peer in order to maintain the contiguity between replicas. It should be noticed that larger the replica set is, higher is the probability for a new peer to induce migrations. Kim and Park try to limit this problem by allowing data blocks to interleave in leafsets [Kim and Park 2006]. However, they have to maintain a global knowledge of the complete leafset: each peer has to know the content of all the peers in its leafset. Unfortunately, the maintenance algorithm is not described in detail and its real cost is unknown.

In the case of the multiple-key-based replication strategy, a new peer may be inserted between two replicas without requiring the migration of data blocks, as long as the new peer identifier does not make it one of the data block roots.

3. RELAXING THE DHT'S PLACEMENT CONSTRAINTS TO TOLERATE CHURN

The goal of this work to is to design a DHT that tolerates a high rate of churn without degradation of performance. In order to achieve this, we avoid to copy data blocks when this is not mandatory to restore a missing replica. We introduce

a leafset based replication that relaxes the placement constraints in the leafset. Our solution, named RelaxDHT, is presented thereafter.

3.1 Overview of RelaxDHT

RelaxDHT is built on top of a KBR layer such as Pastry or Chord. Our design decisions are to use replica localization metadata and separate them from data block storage. We keep the notion of a root peer for each data block. However, the root peer does not necessarily store a copy of the blocks for which it is the root. It only maintains metadata describing the replica set and periodically sends messages to the replica set peers to ensure that they keep storing their copy. It is possible, but not mandatory, for a peer to be both root and part of the replica set of the same data block. Using localization metadata allows a data block replica to be anywhere in the leafset; a new peer may join a leafset without necessarily inducing data block migrations.

We choose to restrain the localization of replicas within the root's leafset for two reasons. First, we believe that it is more scalable, because the *number* of messages of our protocol does not depend on the number of data blocks managed by a peer, but only on the leafset size [2]. Second, since the routing layer already induces many exchanges within leafsets, the local view of the leafset at the DHT-layer can be used as a failure detector. We now detail the salient aspects of the RelaxDHT algorithm.

3.1.1 *Insertion of a new data block.* To be stored in the system, a data block `b` with key `key` is inserted using the `put(k,b)` operation. This operation produces an "insert" message which is sent to the root peer through the KBR layer. Then, the root randomly chooses a replica set of `k` peers around the center of the leafset. The peer belongs to the center of a root's leafset if its hop-distance to the root is less than a fixed value ϵ_1. The algorithm chooses the replica-peers inside the center to reduce the probability that a chosen peer quickly quits the leafset due to the arrival of new peers. Finally, the root sends to the replica set peers a "store message" containing:

(1) the data block itself,
(2) the identity of the peers in the replica set,
(3) the identity of the root.

As a peer may be root for several data blocks and part of the replica set of other data blocks, it stores:

(1) a list `rootOfList` of data block identifiers with their associated replica-set-peer list for blocks for which it is the root;
(2) a list `replicaOfList` of data blocks for which it is part of the replica set. Along with data blocks, this list also contains: the identifier of the data block, the associated replica set peer-list and the identity of the root peer.

A *lease counter* is associated to each stored data block. This counter is set to the value "`Lease`" which is a constant. It is then decremented at each leafset

[2] see section 2.1.2.

maintenance time. The maintenance protocol described below is responsible for periodically resetting this counter to "`Lease`".

3.1.2 *Maintenance protocol.* The goal of this periodic protocol is to ensure that: 1) a root peer exists for each data block. The root is the peer that has the closest identifier to the data block's one; 2) each data block is replicated on `k` peers located in the data block root's leafset.

At each period T, a peer `p` executes Algorithm 1, so as to send maintenance messages to the other peers of the leafset. It is important to notice that Algorithm 1 uses the leafset knowledge maintained by the peer routing layer which is relatively accurate because the inter-maintenance time of the peers is much smaller than the DHT-layer's one.

Algorithm 1: RelaxDHT maintenance message construction.

```
   Result: msgs, the built messages.
1  for data ∈ rootOfList do
2      for replica ∈ data.replicaSet do
3          if NOT isInCenter (replica,leafset) then
4              newPeer =choosePeer (replica,leafset);
5              replace (data.replicaSet, replica,newPeer);
6      for replica ∈ data.replicaSet do
7          add(msgs [replica ],<STORE, data.blockID, data.replicaSet >);
8  for data ∈ replicaOfList do
9      if NOT checkRoot (data.rootPeer,leafset) then
10         newRoot =getRoot (data.blockID,leafset);
11         add(msgs [newRoot ],<NEW ROOT, data.blockID, data.replicaSet >);
12 for p ∈ leafset do
13     if NOT empty (msgs [p ]) then
14         send(msgs [p ],p);
```

The messages constructed by Algorithm 1 contain a set composed of the following elements:

STORE. elements for asking a replica node to keep storing a specific data block (*i.e.*, resetting the lease counter).

NEW ROOT. elements for notifying a node that it has become the new root of a data block.

Each element contains both a data block identifier and the associated replica-set-peer list. Algorithm 1 sends at most one single message to each leafset member.

Algorithm 1 is composed of three phases: the first one computes STORE elements using the `rootOfList` structure (lines 1 to 7), the second one computes NEW ROOT elements using the `replicaOfList` structure (from line 8 to 11), the last one sends messages to the destination peers in the leafset (line 12 to the end). Message elements computed in the two first phases are added to *msgs[]*. *msgs[q]* is a message containing all the elements to send to node q at the last phase.

Therefore, each peer periodically sends a maximum of `leafset-size` maintenance messages to its neighbors. The size of these messages depends on the number of blocks. However, we will show in the performance section that our approach remains scalable even if the number of blocks per node is huge.

In the first phase, for each block for which the peer is the root, it checks if all the replicas are placed sufficiently far from the extremity of the leafset (line 3) using its local view provided by the KBR layer. We define the *extended center* of a root's leafset as the set of the peers which distance to the root is less than a system-defined value ϵ_2 (with $\epsilon_2 > \epsilon_1$, *i.e.*, the center of the leafset is a subset of the extended center of the leafset[3]). If a peer is out of the extended center, its placement is considered to be too close to the leafset extremity. In that case, the algorithm replaces the peer by randomly choosing a new peer in the center of the leafset. It then updates the replica set of the block (lines 4 and 5). Finally, the peer adds a STORE element in each replica-set-peer message (lines 6 and 7). In the second phase, for each block stored by the peer (*i.e.*, the peer is part of the block's replica set), it checks if the root node did not change. This is done by checking that the identifier of the current root is still the closest to the block's key (line 9). If the root has changed, the peer adds a NEW ROOT message element to announce to the future root peer that it is the new root of the data block. Finally, from line 12 to line 14, a loop sends the computed messages to each leafset member.

Note that it is possible to temporarily have two different peers acting as a root peer for the same data block. However, this phenomenon is rare. This may happen when a peer joins, becomes root of a data block and then receive a NEW ROOT message element from a replica node for this data block *before* the old ROOT (its direct neighbor in the leafset) detects its insertion. Moreover, even if this happens, it does not lead to data loss. The replica set of the root will simply receive more STORE messages, and the anomaly will be resolved with the next maintenance of the wrong root (*i.e.* at most 10 minutes later in our system).

Algorithm 2: RelaxDHT maintenance message reception.

```
   Data: message, the received message.
1  for elt ∈ message do
2      switch elt.type do
3          case STORE
4              if elt.id ∈ replicaOfList then
5                  newLease(replicaOfList,elt.id);
6                  updateRepSet(replicaOfList,<elt.id,elt.replicaSet >);
7              else
8                  requestBlock(elt.id,elt.replicaSet);
9          case NEW ROOT
10             addRootElt(rootOfList,<elt.id,elt.replicaSet >);
```

3.1.3 *Maintenance message treatment*

For a STORE element. (line 3), if the peer already stores a copy of the corresponding data block, it resets the associated lease counter and updates the corresponding replica set if necessary (lines 4, 5 and 6). If the peer does not store the associated data block (*i.e.*, it is the first STORE message element for this data

[3]see section 3.1.1 for the definition of the leafset center

block received by this peer), it fetches the latter from one of the peers mentioned in the received replica set (line 8).

For a NEW ROOT element. , a peer adds the data block id and the corresponding replica set in the `rootOfList` structure (line 10).

3.1.4 *End of a lease treatment.* If a data block lease counter reaches 0, it means that no STORE element has been received for a long time. This can be the result of numerous insertions that have pushed the peer outside the leafset center of the data block's root. The peer sends a message to the root peer of the data block to ask for the authorization to delete the block. Then, the peer receives an answer from the root peer, either allowing to remove the data block or asking to *put* the data block again in the DHT (if the data block has been lost).

3.1.5 *Impact of the ϵ_1 and ϵ_2 values on the protocol performance.* The placement constraints of RelaxDHT are defined by the parameters ϵ_1 and ϵ_2. The first parameter defines a set of nodes around a block's root (called the center of the leafset) on which the replicas of the block are initially placed. The second one defines the tolerance threshold of the protocol: the replacement of the replica inside the center is performed only its hop distance to the root exceeds ϵ_2 due to node arrival.

It is thus important to set optimal values for ϵ_1 and ϵ_2. We varied these parameters to study how they affect the churn resilience of the protocol. As it can be seen on the figure 2, ϵ_1 has the most important impact on the churn resilience. Low values of ϵ_1 reduce the churn tolerance of the DHT, because they strengthen the initial placement constraints. Strong placement constraints increase the correlation between the blocks stored by a node, thus reducing the number of sources that can be used to restore replicas when a node fails. For example, if $\epsilon_1 = k$, the replicas of a block are necessarily placed on the root and its $k-1$ contiguious neighbors. It means that in case of the failure of a node n, the replicas of all the blocks stored on n are located on $k+1$ nodes, which is less than for larger values of ϵ_1.

The ϵ_2 parameter should not be too low because it would reduce the placement tolerance of the protocol, and should not be to high, because if a replica shifts out the extended center *and* out of the leafset, its lease is no more renewed, which may lead to the loss of the replica. However, the figure 2 shows that the parameter has less influence on data loss than ϵ_1, because its value is important only when the leafset faces an important node arrival. For our evaluation, we choose $\epsilon_1 = 4$ and $\epsilon_2 = 8$.

3.2 Side effects and limitations

By relaxing placement constraints of data block copies in leafsets, our replication strategy for DHTs significantly reduces the number of data blocks to be transferred when peers join or leave the system. Thanks to this, we show in the next section that our maintenance mechanism allows us to better tolerate churn. However, this enhancement has several possibly inconvenient effects. The two main ones concern the data block distribution on the peers and the lookup performance. While the changes in data block distribution can provide positive effects, the lookup performance can be slightly reduced.

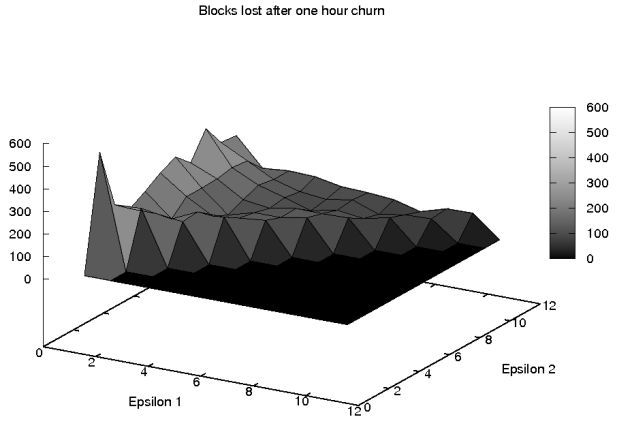


Fig. 2. Number of blocks lost on a 100 peer-DHT after one hour churn in function of the ϵ_1 ans ϵ_2 parameters.

3.2.1 *Data blocks distribution.* With usual replication strategies in DHT's, the data blocks are distributed among peers according to some hash function. Therefore, if the number of data blocks is big enough, data blocks should be uniformly distributed among all the peers. When using RelaxDHT, this remains true if there are no peer connections/disconnections. However, in presence of churn, as our maintenance mechanism does not transfer data blocks if it is not necessary, new peers will store much less data blocks than peers involved for a longer time in the DHT. It is important to notice that this side effect is rather positive: the more stable a peer is, the more data blocks it will store. Furthermore, it is possible to easily counter this effect by taking into account the quantity of stored data blocks while randomly choosing peers to add in replica sets.

3.2.2 *Lookup performance.* Our strategy also induces additionnal delay while retrieving blocks from the DHT because the placement of the data block on its root is no more mandatory. During a lookup, if no replica is stored on the root, the root has to forward the request to one of the k replica nodes. This adds one hop to the request, and the latency of the last link is added to the overall lookup latency. This never happens in PAST, because the root of a block necessarily stores it.

We compared the lookup performance of both strategies by simulating a DHT with 1000 nodes in presence of churn. As we can see in figure 3.a, the lookup latency is in average about 13% lower with PAST than with RelaxDHT. The figure 3.b shows that the percentage of failed lookups increases the same way for the two strategies as the churn intensifies. RelaxDHT has a slightly better failure percentage, because all the nodes of the replica set (*i.e.*, the root *and* the k replica nodes) have the complete list of the replicas, thus allowing to re-route the request if the root does not have the block. In PAST, if the block on the root is missing, the lookup fails.

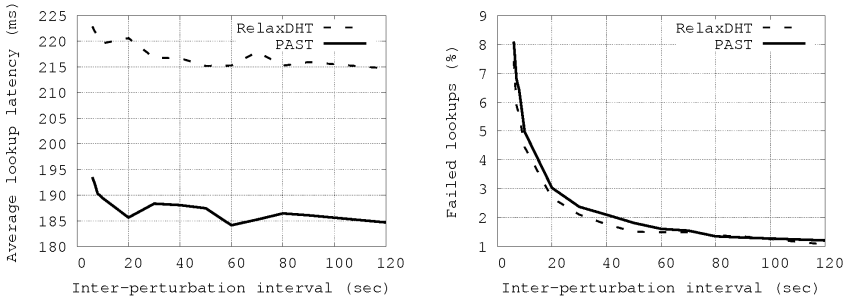


Fig. 3. For 1000 nodes, varying the churn rate: (a) Average lookup latency, (b) Percentage of failed lookups.

The observed latency overhead is constant and relatively low, because the root has the list of all the replica-nodes, and is able to choose the least expensive link for the last hop. As the main concern of DHTs is to provide large scale persistent data storage, we consider that the slight latency overhead in data retrieval is affordable in most of the cases. Moreover, if the lookup performance is essential to the application, it is possible to lower the probability that a root does not to store a replica by choosing small values for ϵ_1[4]. Small values of ϵ_1 increase the placement constraints, reducing the churn resilience, but a tradeoff between the lookup efficiency and the churn resilience can be found.

4. EVALUATION

This section provides a comparative evaluation of RelaxDHT and PAST [Rowstron and Druschel 2001b]. This evaluation, based on discrete event simulations, shows that RelaxDHT provides a considerably better tolerance to churn: for the same churn levels, the number of data losses is divided by more than two when comparing both systems.

4.1 Experimental setup

To evaluate RelaxDHT, we have built a discrete event simulator using the PeerSim [Jelasity et al. 2008] simulation kernel. We have implemented both PAST and RelaxDHT strategies. It is important to note that all the different layers and all message exchanges are simulated. Our simulator also takes into account network congestion because DHT maintenance during churn incurs a lot of simultaneous downloads that are likely to congest the links. Moreover, we used real-life latency traces. Measurements performed by Madhyastha et al. [Madhyastha et al. 2006] between DNS servers in 2004 were injected in the simulation to simulate a realistic latency distribution.

For all the simulation results presented in the section, we used a 1000-peer network with the following parameters (for both PAST and RelaxDHT):

—a leafset size of 24, which is the Pastry default value;

[4] *e.g.,* if $\epsilon_1 = k$, k being the replica rate, all the replicas are contiguously stored around the root, forcing the root to store a replica.

—an inter-maintenance duration of 10 minutes at the DHT level;

—an inter-maintenance duration of 1 minute at the KBR level;

—100,000 data blocks of 10,000 KB replicated 3 times;

—network links of 1 Mbits/s for upload and 10 Mbits/s for download.

—for RelaxDHT maintenance protocol, the leafset center is set to the 8 central nodes, while the extended center is set to the 16 central nodes of the leafset[5].

—the replica set lease is set to 5 DHT maintenance periods, *i.e.*, 50 minutes.

We have injected churn following two different scenarii:

One hour churn. This scenario allows us to study 1) how many data blocks are lost after a perturbation period and 2) how long it takes to the system to return to a state where all remaining/non-lost data blocks are replicated `k` times. It consists of one perturbation phase with churn during one hour followed by another phase without connections/disconnections. In real-life, there are some period without churn within a leafset, and the system has to take advantage of them to converge to a safer state.

Continuous churn. This scenario focuses on the perturbation period: it provides the ability to study how the system resists when it has to repair lost copies in presence of churn. For this set of simulations, we focus on phase one of the previous case observing a snapshot of the DHT during churn.

During the churn phase at each *perturbation period* we randomly choose either a new peer connection or a peer disconnection. This perturbation can occur anywhere in the ring (uniformly chosen). We have run numerous simulations varying the inter-perturbation delay.

4.2 Single failure

In order to better understand the simulation results using the two scenarii, we start by measuring the impact of a single peer failure/disconnection. When a single peer fails, data blocks it stored have to be replicated on a new one. Those blocks are transferred to the new peer in order to rebuild the initial replication degree k. In our simulations, with the parameters given above, it takes 4609 seconds to PAST to recover the failure: *i.e.*, to create a new replica for each block stored on the faulty peer, while with RelaxDHT, it takes only 1889 seconds. The number of peers involved in the recovery is indeed much more important. This gain is due to the parallelization of the data block transfers:

—in PAST, the content of contiguous peers is really correlated. With a replication degree of 3, only peers located at one or two hops of the faulty peer in the ring may be used as sources or destinations for data transfers. In fact, only `k+1` peers are involved in the recovery of one faulty peer, where `k` is the replication factor.

—in RelaxDHT, all of the peers in the extended center of the leafset (the extended center contains 16 peers in our simulations) may be involved in the transfers.

[5]i.e., $\epsilon_1 = 4$ and $\epsilon_2 = 8$, see section 3 for the description of these sets.

4.3 One hour churn

We first study the number of lost data blocks (data block for which the 3 copies are lost) in PAST and in RelaxDHT under the same churn conditions. Figure 4.a shows the number of lost data blocks after a one-hour churn period. The inter-perturbation delay is increasing along the X axis. With RelaxDHT and our maintenance protocol, the number of lost data blocks is 2 to 3 times lower than with the PAST's one.

The main reason of the result presented above is that, using the PAST replication strategy, the peers have more data blocks to download. It implies that the mean download time of one data block is longer using PAST replication strategy. Indeed, the maintenance of the replication scheme location constraints generate a continuous network traffic that slows down critical transfers, preventing efficient data block copy restoration.

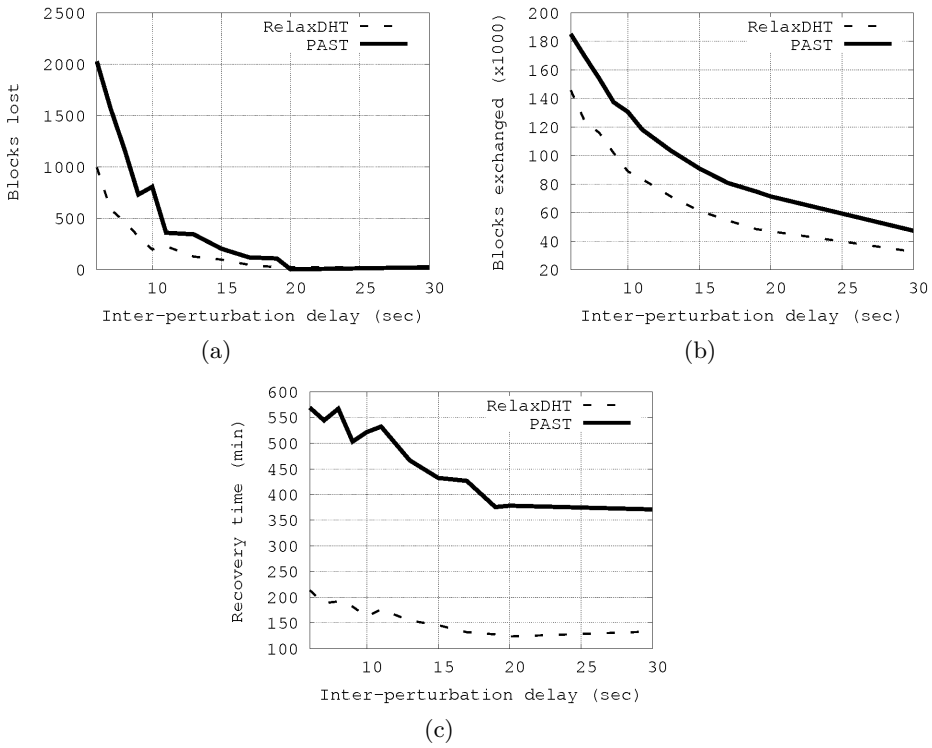


Fig. 4. (a) Number of data block lost (i.e., all copies are lost) after one hour of churn, (b) Number of exchanged data blocks to restore a stable state after one hour of churn, (c) Recovery time: time for retrieving all the copies of every remaining data block.

Figure 4.b shows the total number of blocks exchanged in both cases. There again, the X axis represents the inter-perturbation delay. The figure shows that with RelaxDHT the number of exchanged blocks is always smaller than with PAST. This is mainly due to the fact that in PAST's case some transfers are not critical

and are only done to preserve the replication scheme constraints. For instance, each time a new peer joins, it becomes root of some data blocks [6] or is inserted within replica sets that should remain contiguous. As a consequence, a reorganization of the storage has to be performed. In RelaxDHT, most of the non-critical bandwidth consumption is replaced by critical data transfers: the maintenance traffic is more efficient.

Using PAST replication strategy, a newly inserted peer may need to download data blocks during many hours, even if no failure/disconnection occurs. During all this time, its neighbors need to send it the required data blocks, using a large part of their upload bandwidth.

In the case of RelaxDHT, *no* or very *few* data block transfers are required when new peers join the system. It becomes mandatory only if some copies become too far from their root-peer in the logical ring (*i.e.*, they leave the leafset center, which is, in our simulation, formed of the 16 peers that are the closest to the root peer). In this case, they have to be transferred closer to the root before their hosting peer leaves the root-peer's leafset. With a replication degree of 3 and a leafset size of 24, many peers can join a leafset before any data block transfer is required.

Finally, we have measured the time the system takes to return to a normal state in which every remaining data block is replicated `k` times. Blocks for which all copies are lost can not be recovered and are thus not taken into account. Figure 4.c shows the results obtained while varying the delay between perturbations. We can observe that the recovery time is four to five times longer in the case where PAST is used compared to RelaxDHT. This result is mainly explained by the efficiency of the maintenance protocol: RelaxDHT transfers only very few blocks for placement constraints compared to PAST's one.

This last result shows that the DHT using RelaxDHT repairs damaged data blocks (data blocks for which some copies are lost) faster than PAST. It means that it will be promptly able to cope with a new churn phase. The next section describes our simulations with continuous churn.

4.4 Continuous churn

The above simulation results show that: 1) RelaxDHT induces less data transfers, and 2) remaining data transfers are more parallelized. Thanks to this two points, even if the system remains under continuous churn, RelaxDHT provides a better churn tolerance.

Figure 5.a shows the number of data block losses under continuous churn using the parameters described at the beginning of this section. Here again, we can see that PAST starts to loose data blocks with lower churn rate than RelaxDHT. This delay needs to be of less than 20 seconds [7] for RelaxDHT to loose a significative amount of blocks, whereas PAST continues to loose blocks even for inter-perturbation intervals greater than 40 seconds. If the inter-perturbation delay continues to decrease, the number of lost data blocks using RelaxDHT strategy remains less than the third of the number of data blocks lost using PAST strategy.

[6] because its identifier is closer than the current root-peer's one
[7] for 1000 nodes

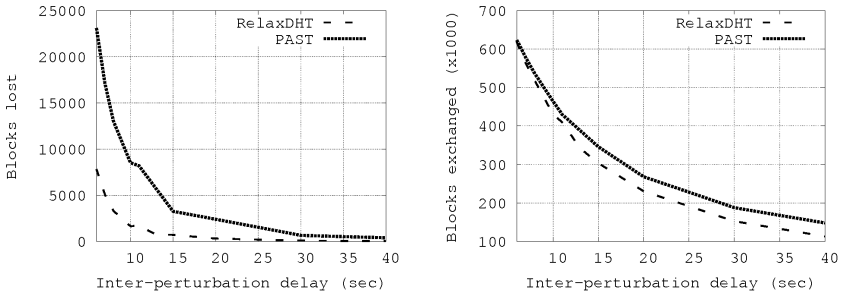


Fig. 5. While the system is under continuous churn: (a) Number of data block transfers, (b) Number of data block losses (all k copies lost)

Finally, Figure 5.b confirms that even with a continuous churn pattern, during a 5 hour run, the proposed solution behaves better than PAST. The number of data transfers required by RelaxDHT is still always smaller than the number of data transfers induced by the PAST's replication strategy (about 10% smaller for a 10 second inter-perturbation delay and 50% smaller for a 35 second delay). In that case, the continuous churn rate induces more critical data exchange than in the first scenario. For RelaxDHT, the bandwidth gained by relaxing the placement constraints is mostly re-used to exchange critical data. Therefore, the bandwidth consumption of our protocol is closer to the PAST's one than in the first scenario, but the bandwidth management is more efficient.

5. MAINTENANCE PROTOCOL COST

In the simulation results presented above, we have considered that the maintenance protocol cost was negligible. This section evaluates the network cost of PAST and RelaxDHT maintenance protocols. Both RelaxDHT and PAST peers send at most one maintenance message to each leafset member, that is why it is appropriate to compare the *global* amount of data to be sent by a node in order to perform a maintenance. Then, we evaluate the optimizations that can be made to reduce network cost of the protocols.

5.1 Amount of exchanged data

Let: 1) M be the overall number of blocks in the DHT, 2) N be the number of DHT nodes, 3) m be the number of peers in the center of the leafset RelaxDHT uses to replicate a block, 4) k be the mean replication factor and 5) $|leafset|$ the size of the leafset of the DHT (PAST and RelaxDHT have the same $|leafset|$).

Let S be the set of blocks a node has to maintain. As the blocks are uniformly distributed by the hash function, we have in average $|S| = \frac{M \times k}{N}$.

PAST maintenance cost. While performing a maintenance, a PAST peer sends all the identifiers of blocks it stores to every member of its leafset. Therefore, the average cost of the maintenance is $Maintenance_{PAST} = |S| \times |leafset| = \frac{M \times k}{N} \times |leafset|$ identifiers to send to the leafset neighbors.

RelaxDHT maintenance cost. In RelaxDHT, on each node, S can be partitioned in 3 subsets:

(1) **Subset R.** Data blocks for which the node is the root and is *not* part of the replica set. Since the DHT hash-function is uniform, and each block has a root, $|R| = \frac{M}{N}$ blocks.

(2) **Subset T.** Data blocks for which the node is not the root of the block, but is part of its replica set. Let T be the subset of S formed of such blocks. Since $S = R \cup T \cup (T \cap R)$, $|T| = \frac{M \times k}{N} - \frac{M}{N} - |T \cap R|$.

(3) **Subset $T \cap R$.** Data blocks for which the node is both the root of the block and part of the replica set. As a block is inserted, the root chooses k replica-peers among m central leafset-members. Let p be the probability for the root to choose itself as a replica[8]. Thus, $|T \cap R| = \frac{M}{N} \times p$ blocks.

Each maintenance time, Algorithm 1 computes STORE and NEW ROOT elements: 1) For each block of the set R, k STORE elements are created. 2) For each block of the set $T \cap R$, only $k-1$ STORE elements are created. 3) There are no STORE elements for blocks that belong to the set T.

Therefore, to perform a maintenance, a node has to send $\#ST = |R| \times k + |T \cap R| \times (k-1)$ STORE elements.

Moreover, depending on the churn rate, some NEW ROOT elements are sent for the members of the set T. If there is no churn at all, no NEW ROOT elements are sent. On the other hand, in the worst case, it could be mandatory to send a NEW ROOT element per member of T. In the worst case, the number of NEW ROOT elements is therefore: $\#NR = |T| \approx \frac{M}{N} \times (k-1)$. It is important to notice that this occurs only in case of a massive node failure. In practice, the amount of NEW ROOT messages induced by churn is considerably smaller.

Each element contains $k+1$ identifiers: the identifier of the block and the k identifiers of the replica set members. Thus, in average, a RelaxDHT node has to send $Maintenance_{RelaxDHT} = (\#NR + \#ST) \times (k+1) \approx \frac{M \times k}{N} \times p \times (k+1)$ identifiers per maintenance.

Comparison. Putting aside all optimisations that are made for both RelaxDHT and PAST, the cost of both protocols can now be compared. As we usually have $k \ll |leafset|$ (for example, in our simulations, $k = 3$ and $|leafset| = 24$) , $Maintenance_{RelaxDHT} < \frac{M \times k}{N} \times p \times |leafset|$

Therefore, since $p < 1$, $Maintenance_{RelaxDHT} < Maintenance_{PAST}$.

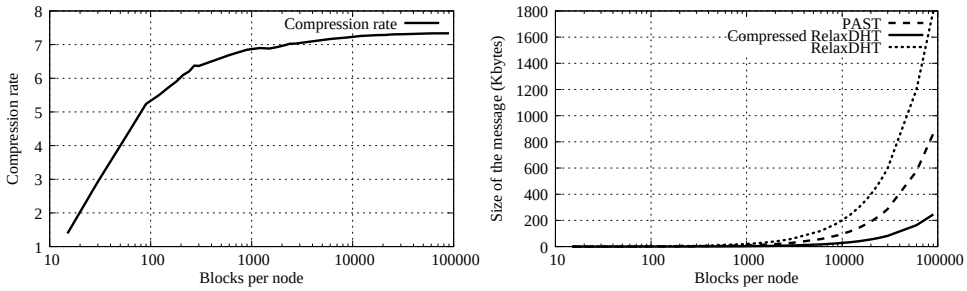
This result is mainly due to the fact that PAST peers send their content to all the members of their leafset while RelaxDHT peers use extra metadata to locally compute the information that needs to be transferred from one peer to another. Moreover, as there are less NEW ROOT messages when there is less churn means that the RelaxDHT protocol is able to adapt itself to be more lightweight as churn drops, whereas PAST protocol cost is constant, even if there is no churn.

We now discuss the impact of the optimizations that can be made to both protocols on their network load.

[8]in our case, the choice is made at random with a uniform distribution, therefore $p = \frac{k}{m}$.

5.2 Optimization of maintenance message size

In PAST, the optimization of maintenance traffic relies on the usage of bloom filters. This space-efficient probabilistic data structure helps each peers to propagate the information about the data blocks it stores. Given a data block identifier, the bloom filter is used to determine whether or not this identifier belongs to the set of identifiers from which the bloom filter has been formed. With a certain probability, depending on its size and on the size of the set, the bloom filter allows false positives [Broder et al. 2002]. It means that a peer examining a neighbor's bloom filter searching for missing data blocks could decide that this neighbor stores a data block, while it is actually missing. In order to minimize the probability of false positives, the size of the bloom filter needs to be increased. For example, allocating an average size of 10 bits per element in the bloom filter provides approximatively 1% of false positives [Broder et al. 2002].



(a) (b)

Fig. 6. (a) Compression rate of RelaxDHT maintenance messages in function of the number of blocks per node, (b) Compressed RelaxDHT maintenance message size compared to PAST maintenance message size using bloom filters with 1% false positive rate (9.6 bits per element).

RelaxDHT is unable to use bloom filters for traffic optimization, because its maintenance messages have a well defined structure. Each data block key is associated a set of its replica-peers, whereas bloom filters are only capable of providing information about the belonging of a key to a key-set. Therefore, RelaxDHT uses non-probabilistic lossless compression, such as dictionary coders [Ziv and Lempel 1977]. This kind of compression is efficient for maintenance message compression, because they contain many redundant digit sequences. This is due to two main factors: 1) All peer identifiers and data block keys are close in the DHT ring because they are located in the same leafset. It means that digital distance between two identifiers is low and decreases as the number of blocks/peers increases. 2) All the replica-peers are located inside the middle of the block's root leafset. Therefore, the sets of peer identifiers associated with different STORE and NEW ROOT elements are likely to be close to each other. This phenomenon also increases with the number of blocs.

Thanks to the two factors, the compression rate of messages increases with the load of the DHT. Figure 6.a recapitulates the results obtained while compressing maintenance messages generated by the simulation. Assuming that a standard DHT node is able to allocate at least 10Gbytes of local storage space, it is able to store more than 1000 blocks of 10Mbytes each. Therefore, it is reasonable to suppose that a compressed message is commonly 6 to 8 times smaller than the original one[9].

We compared the cost of both maintenance protocols, using 1% acceptable false-positive bloom filter rate for PAST. Figure 6.b shows that uncompressed RelaxDHT messages are more voluminous than PAST's ones. However, if compression is activated, RelaxDHT messages are 3 to 4 times smaller than PAST's ones. Furthermore, it is important to notice that in absence of churn (*i.e.*, when the leafset does not change between two maintenance times), a RelaxDHT root node may replace a regular maintenance message by a simple ping to perform maintenance. Therefore, taking this optimization into account, the average maintenance message size should be very limited. In other words, RelaxDHT maintenance protocol is more efficient than the PAST's one because: 1) in absence of churn, the maintenance is almost negligible: the protocol is able to adapt itself to the churn rate; 2) in presence of churn, its messages are less voluminous than PAST ones; 3) there is no information loss (no false positives as in PAST).

Regardless of the data compression efficiency, the size of the RelaxDHT maintenance messages linearly grows with the overall number of data blocks stored in the DHT (cf. logarithmic-scaled figure 6.b). However, in order to provide an acceptable rate of false positives induced by the bloom filters, PAST maintenance algorithm also should gradually increase the size of the its bloom filters as the number of stored blocks grows. It means that PAST maintenance message size also linearly grows with the number of stored blocks.

Finally, the linear growth of maintenance message size is not a problem for RelaxDHT. Indeed, consider that 1,000,000 blocks stored on a 100-peer RelaxDHT with replica rate of 3 induce 70Kbytes of data per maintenance message (see Figure 6.b). Knowing that these messages are sent by a peer with a sparse interval (the default PAST value is 10 minutes) and only to peers located "in the middle" of its leafset (*i.e.*, its 16 closest neighbors in our simulations), this size may be considered as negligible compared to data block transfer.

6. CONCLUSION

Distributed Hash Tables provide an efficient, scalable and easy-to-use storage system. However, existing solutions loose many data in presence of a high churn rate or are not really scalable in terms of stored data block number. We have identified one of the reasons why they do not tolerate high churn rate: they impose strict placement constraints that induce unnecessary data transfers.

In this paper, we propose a new replication strategy, RelaxDHT that relaxes the placement constraints: it relies on metadata (replica-peers/data identifiers) to allow a more flexible location of data block copies within leafsets. Thanks to this

[9]We used the gzip software to compress messages [ZZZ-gzip].

design, RelaxDHT entails fewer data transfers than classical leafset-based replication mechanisms. Furthermore, as data block copies are shuffled among a larger peer set, peer contents are less correlated. This benefits to RelaxDHT because in case of failure, more data sources are available for the download of a missing block, which makes the recovery more efficient and thus the system more churn-resilient. Our simulations, comparing the PAST system to ours, confirm that RelaxDHT 1) induces less data block transfers, 2) faster recovers lost data block copies and 3) looses less data blocks. Furthermore, we show that the churn-resilience does not involve a prohibitive maintenance overhead.

REFERENCES

Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer, R. 2002. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. Boston, MA, USA.

Broder, A., Mitzenmacher, M., and Mitzenmacher, A. B. I. M. 2002. Network applications of bloom filters: A survey. In *Internet Mathematics*. 636–646.

Busca, J.-M., Picconi, F., and Sens, P. 2005. Pastis: A highly-scalable multi-user peer-to-peer file system. In *Euro-Par '05: Proceedings of European Conference on Parallel Computing*. 1173–1182.

Castro, M., Costa, M., and Rowstron, A. 2004. Performance and dependability of structured peer-to-peer overlays. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 9.

Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., and Hellerstein, J. M. 2005. A case study in building layered dht applications. In *SIGCOMM*, R. Guérin, R. Govindan, and G. Minshall, Eds. ACM, 97–108.

Dabek, F., Kaashoek, F. M., Karger, D., Morris, R., and Stoica, I. 2001. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the 8th ACM symposium on Operating Systems Principles*. Vol. 35. ACM Press, New York, NY, USA, 202–215.

Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, F. F., and Morris, R. o. 2004. Designing a DHT for low latency and high throughput. In *NSDI '04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*. San Francisco, CA, USA.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The google file system. In *SOSP '03: Proceedings of the 9th ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 29–43.

Ghodsi, A., Alima, L. O., and Haridi, S. 2005. Symmetric replication for structured peer-to-peer systems. In *DBISP2P '05: Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*. Trondheim, Norway, 12.

Gupta, A., Agrawal, D., and Abbadi, A. E. 2003. Approximate range selection queries in peer-to-peer systems. In *CIDR*.

Jelasity, M., Montresor, A., Jesi, G. P., and Voulgaris, S. 2008. The Peersim simulator. `http://peersim.sf.net`.

Jernberg, J., Vlassov, V., Ghodsi, A., and Haridi, S. 2006. Doh: A content delivery peer-to-peer network. In *Euro-Par '06: Proceedings of European Conference on Parallel Computing*. Dresden, Germany, 13.

Kim, K. and Park, D. 2006. Reducing data replication overhead in DHT based peer-to-peer system. In *HPCC '06: Proceedings of the 2nd International Conference on High Performance Computing and Communications*. Munich, Germany, 915–924.

Ktari, S., Zoubert, M., Hecker, A., and Labiod, H. 2007. Performance evaluation of replication strategies in DHTs under churn. In *MUM '07: Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*. ACM Press, New York, NY, USA, 90–97.

Landers, M., Zhang, H., and Tan, K.-L. 2004. Peerstore: Better performance by relaxing in peer-to-peer backup. In *P2P '04: Proceedings of the 4th International Conference on Peer-to-Peer Computing*. IEEE Computer Society, Washington, DC, USA, 72–79.

Legtchenko, S., Monnet, S., Sens, P., and Muller, G. 2009. Churn-resilient replication strategy for peer-to-peer distributed hash-tables. In *The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*. Lecture Notes in Computer Science, vol. 5873. Springer Verlag, Lyon, Fr, 485–499.

Lian, Q., Chen, W., and Zhang, Z. 2005. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 187–196.

Madhyastha, H. V., Anderson, T. E., Krishnamurthy, A., Spring, N., and Venkataramani, A. 2006. A structural approach to latency prediction. In *Internet Measurement Conference*, J. M. Almeida, V. A. F. Almeida, and P. Barford, Eds. ACM, 99–104.

Maymounkov, P. and Mazieres, D. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '02: Proceedings of the 1st International Workshop on Peer-to-Peer Systems*. Cambridge, MA, USA, 53–65.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. 2001. A scalable content-addressable network. In *SIGCOMM*. Vol. 31. ACM Press, 161–172.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. 2004. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*.

Rodrigues, R. and Blake, C. 2004. When multi-hop peer-to-peer lookup matters. In *IPTPS '04: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*. San Diego, CA, USA, 112–122.

Rowstron, A. and Druschel, P. 2001a. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science 2218*, 329–350.

Rowstron, A. I. T. and Druschel, P. 2001b. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the 8th ACM symposium on Operating Systems Principles*. 188–201.

Schmidt, C. and Parashar, M. 2004. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing 8,* 3, 19–26.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, F. F., Dabek, F., and Balakrishnan, H. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11,* 1 (February), 17–32.

van Renesse, R. 2004. Efficient reliable internet storage. In *WDDDM '04: Proceedings of the 2nd Workshop on Dependable Distributed Data Management*. Glasgow, Scotland.

Varvello, M., Diout, C., and Biersack, E. W. 2009. P2p second life: Experimental validation using kad. In *INFOCOM*. IEEE, 1161–1169.

Zahn, T. and Schiller, J. H. 2006. Dht-based unicast for mobile ad hoc networks. In *PerCom Workshops*. IEEE Computer Society, 179–183.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. 2003. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. 2004. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 41–53.

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 337–343.

ZZZ-gzip. Gzip. `http://www.gzip.org/`.