INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Churn-resilient replication strategy for peer-to-peer distributed hash-tables

Sergey Legtchenko — Sébastien Monnet — Pierre Sens — Gilles Muller

N° 6897

April 2009

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6897--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche PARIS - ROCQUENCOURT

Churn-resilient replication strategy for peer-to-peer distributed hash-tables

Sergey Legtchenko*, Sébastien Monnet* , Pierre Sens* , Gilles Muller†

Thème COM — Systèmes communicants
Équipe-Projet REGAL

Rapport de recherche n° 6897 — April 2009 — 21 pages

Abstract: DHT-based P2P systems provide a fault-tolerant and scalable mean 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 current DHT-based system's implementations. To avoid this problem, it is necessary to build really efficient replication and maintenance mechanisms. In this paper, we study the effect of churn on an existing DHT-based P2P system such as Chord or Pastry. We then propose solutions to enhance churn tolerance and evaluate them through discrete event simulations.

Key-words: Churn, Replication, Distributed hash tables (DHT), peer-to-peer (P2P), Fault tolerance

* LIP6/University of Paris VI/CNRS/INRIA
† EMN/INRIA

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Stratégie de réplication résistante au churn pour les tables de hachage distribuées pair-à-pair

Résumé : Les systèmes pair-à-pair basés sur des tables de hachage distribuées (DHT pour *Distrubuted Hash Tables*) offrent un moyen de stockage à large échelle tolérant aux fautes. Malheureusement, des études récentes ont montré que si les connexions/déconnexions étaient trop fréquentes, des données pouvaient être perdues. Ceci est vrai pour la plupart des implémentations existantes. Il est nécessaire de concevoir des stratégies de réplication et des mécanismes de maintenance très efficaces afin de faire face à ce problème. Dans ce papier, nous étudions l'effet des connexions/déconnexions fréquentes (*churn*) sur des implémentations existantes et proposons des solutions améliorant la résistance au churn que nous évaluons à travers un simulateur à événements discrets.

Mots-clés : Churn, Réplication, Tables de hachage distribuées (DHT), pair-à-pair (P2P), Tolérance aux fautes

1 Introduction

Distributed Hash Tables (DHTs), are distributed storage services that use a structured overlay relying on key-based routing (KBR) protocols [1, 2]. 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 [3], distributed file systems [4, 5] and content distribution systems [6].

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 worse the loss of some data and at least performance degradation, due to the reorganization of the set of replicas of the affected data, that consumes bandwith and CPU cycles. In fact, Rodrigues and Blake have shown that using classical DHTs to store large amounts of data is only viable if the peer life-time is in the order of several days [7]. Until now, the problem of churn resilience has been mostly addressed at the P2P routing level to ensure the reachability of peers by maintaining the consistency of the logical neighborhood, i.e., the leafset, of a peer [8, 9]. 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 data block is associated a *root* peer whose identifier is the (numerically) closest to its key. The traditional replication scheme relies on using the subset of the root leafset containing the closest logical peers to store the copies of a data block [1]. 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 term of bandwidth consumption [3]. A solution to this problem, relies on creating multiple keys for a single data block [10, 11]; therefore, only a peer maintaining a key can be affected by a reconfiguration. However, each peer maintaining a data block has to periodically check the state of all the peers possessing a replica. Since copies are randomly spread on the overlay the number of peers to check can be huge.

This paper proposes a variant of the leafset replication strategy that tolerates a high churn rate. 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 additional overhead in practice.

We have implemented both PAST and our replication strategy on top of PeerSim [12]. The main results of our evaluations are:

- We show that our approach achieves higher data availability in presence of churn, than the original PAST replication strategy. For a connection/disconnection occuring every minute our strategy loses two times less blocks than PAST's one.
- We show that our replication strategy induces an average of twice less block transfers than PAST's one.

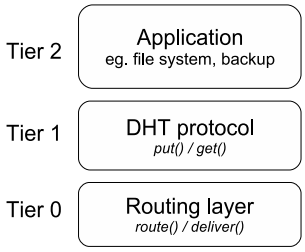

Figure 1: Structure of a DHT-based system

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 an enhanced replication scheme for which the DHT's placement constraints are relaxed so as to obtain a better churn resilience. Simulations of this algorithm are presented in Section 4. Section 5 concludes with an overwiew of our results.

2 Background and motivation

DHT based P2P systems are usually structured in three layers as illustrated in Figure 1: 1) a routing layer, 2) the DHT itself, 3) the application that uses the DHT. The routing layer is based on keys for identifying peers and is therefore commonly qualified as *Key-Based Routing* (KBR). Such KBR layer hides 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 [8]. The main examples of KBR layers are Pastry [13], Chord [2], Tapestry [14] and Kademlia [15].

The DHT layer is responsible for storing data blocks. It implements a distributed storage service that provides persistence and fault tolerance, and can scale up to a large number of peers. DHTs provide simple get and put abstractions that greatly simplifies the task of building large-scale distributed applications. PAST [1] and DHash [16] are DHTs respectively built on top of Pastry [13] and Chord [2]. Finally, the application layer is a composition of any distributed application that may take advantage of a DHT. Representative examples are the CFS distributed file system [5] and the PeerStore backup system [3].

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

2.1 Replication in DHTs

In a DHT, each peer and each data block is assigned an identifier (i.e., a key). A data block's key is usually the result of a hash function performed on the block. The peer which identifier is the closest to the block's key is called the block's *root*. All the identifiers are arranged in a logical structure, such as a ring

as used in Chord [2] and Pastry [13] or a d-dimensional torus as implemented in CAN [10] and Tapestry [11].

A peer possesses a restricted local knowledge of the P2P network, i.e., the leafset, which amounts to a list of its 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. Each peer monitors its leafset, removing peers which 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 for implementing these two protocols.

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 successors of the root in the ring, predecessors or both. Therefore, the different copies of a block are stored *contiguously* in the ring as shown by Figure 2. This strategy has been implemented in PAST [1] and DHash [16]. *Successor replication* is a variant of leafset-based replication where replica peers are only the immediate successors of the root peer instead of being the closest peers [17].

- **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 [10] and Tapestry [11]. GFS [18] uses a variant based on random placement to improve data repair performance. *Path* and *symmetric replication* are variants of multiple key based replication [19, 17]. Path replication stores data blocks along a routing path, using the path to attribute the keys, then each peer on the path is responsible for monitoring its successor [17]. Symmetric replication is a particular kind of multiple key based replication [19] where an identifier of a block is statically associated with $f - 1$ other identifiers. Harvesf and Blough propose a random placement scheme focusing on producing disjoint routes for each replica set [20].

Lian *et al.* propose an hybrid stripe replication scheme where small objects are grouped in blocks and then randomly placed [21]. They show using an analytical framework that their scheme achieves on near-optimal reliability.

Finally, several works have focused on the placement strategies based on availability of nodes. Van Renesse [22] proposes a replica placement algorithm on DHT by considering the reliability of nodes and placing copies on nodes until the desired availability was 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 [23], dynamic placement strategies improve the

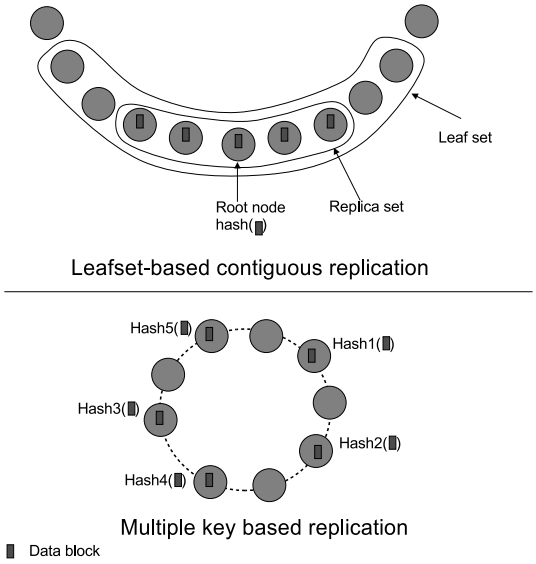

Figure 2: Leafset-based and multiple key based replication ($k = 5$).

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, the cost to track reliability of nodes can be high. Furthermore, such approaches may lead to an high unbalanced distribution whereby highly available nodes contain most of the replicas and can become overloaded.

Maintenance protocols

The maintenance protocols have to maintain `k` copies of each data block without violating the initial placement strategy. This 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. For instance, in the fully decentralized PAST maintenance protocol [1], each peer sends a bloom filter[1] of the blocks it stores to its leafset. 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 such blocks. The requesting peer can then fetch the missing blocks listed in all the answers it receives.

In the case of the multiple key 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.

[1]For short, the sent bloom filter is a compact and approximative view of the list of blocks stored by a peer.

2.2 Impact of the churn on the DHT performance

A high churn rate induces a lot 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 necessary only for enforcing placement invariants.

A first example arises at the root peer level which may change if a new peer with a closer identifier joins the system. In this situation, the data block will be migrated on the new peer. A second example occurs in leafset-based replication, if a peer possesses an identifier that places it within a replica-set. Therefore, data blocks have to be migrated by the DHT to enforce replicas to maintain the "closest peers from the root" property. It should be noticed that larger the replica-set, higher 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 [24]. 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 replication strategy, a new peer may be inserted between two replicas without requiring migrating data blocks, as long as the new peer identifier does not make it one of the data block roots. However, this replication method has the drawback that maintenance has to be done on a per-data block basis; therefore it does not scale up with the number of blocks managed by a peer. For backup and file systems that may store up to thousands of data blocks per peer, this is a severe limitation.

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 degradating of performance. For this, we avoid to copy data blocks when this is not mandatory for restoring 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 meta-data and separate them from data block storage. We keep the notion of a root peer for each data block. However, the root peer does no longer 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. Using localization medata allows a data block replica to be anywhere in the leafset; a new peer may join a leafset without necessarily inducing data blocks migrations.

We choose to restrain the localization of replicas within the root's leafset for two reasons. First, to remain scalable, 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. Second, because 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.

Insertion of a new data block

To be stored in the system, a data block is inserted using the `put(k,b)` operation. This operation produces an "insert message" which is sent to the root peer. Then, the root randomly chooses a replica-set of `k` peers around the center of the leafset. This reduces the probability that a chosen peer quickly becomes out of 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 (i.e., the metadata),
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[2], 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 (see Figure 3). This counter is set to a value `L`, and is then decremented at each KBR-layer maintenance. The maintenance protocol described below is responsible to periodically reset this counter to `L`.

Maintenance protocol

The goal of this periodic protocol is to ensure that:

- A root peer exists for each data block. The root is the peer that the closest identifier from the data block's one.
- 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 KBR layer which is relatively accurate because the inter-maintenance time of the KBR layer is much smaller than the DHT-layer's one.

The messages constructed by Algorithm 1 contain a set of following two elements (see Figure 4):

[2]It is naturally possible, but not mandatory at all, for a peer the be both root and part of the replica-set of a same data block.

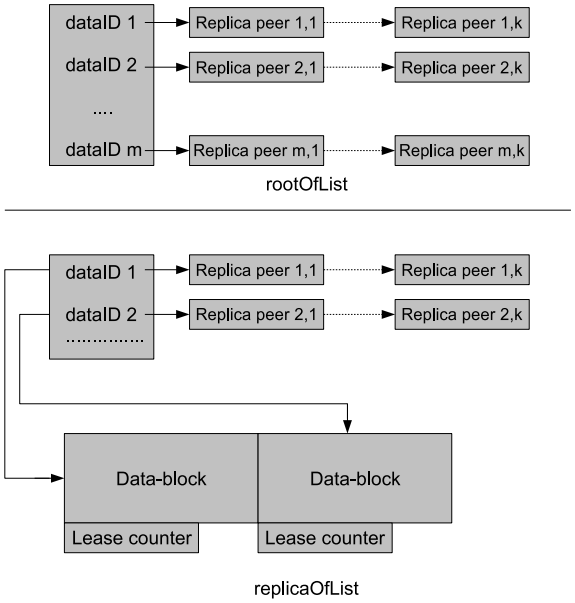

Figure 3: Data structures managed on each peer.

Algorithm 1: RelaxDHT maintenance message construction.

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

- **STORE** element for asking a replica node to keep storing a specific data block.
- **NEW ROOT** element for notifying a node that it has become the new root of a data block.

These message elements contain both a data block identifier and the associated replica-set peer-list. In order to remain scalable in term of the number

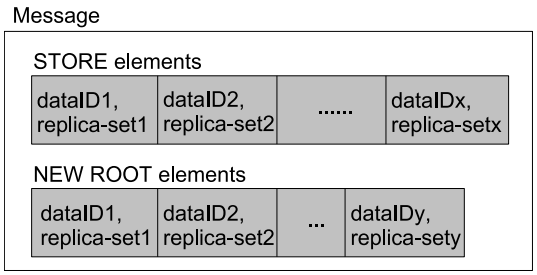

Figure 4: Message composed of x STORE elements and y NEW ROOT elements.

of data blocks 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 in *msgs[]*. *msgs[q]* is a message like the one presented by Figure 4 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.

In the first phase, for each block for which the peer is the root, it checks if every replica is still in the center of its leafset (line 3) using its local view provided by the KBR layer. If a replica node is outside, the peer replaces it by randomly choosing a new peer in the center of the leafset and it then updates the replica-set of the block (lines 4 and 5). Finally, the peer adds a STORE element in each replica set peers messages (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 verification is done by comparing the `replicaOfList` metadata and the current leafset state (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 root of the data block.

Finally, from line 12 to line 14, a loop sends the computed messages to each leafset member.

Maintenance message treatment

Upon reception of a maintenance message, a peer executes Algorithm 2.

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

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.data ∈ replicaOfList then
5                  newLease(replicaOfList,elt.data);
6                  updateRepSet(replicaOfList,elt.data);
7              else
8                  requestBlock(elt.data);
9          case NEW ROOT
10             rootOfList = rootOfList ∪ elt.data ;
```

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

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 center of the leafset 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. Later, the peer will receive an answer from the root peer. This answer either allows it to remove the data block or asks it to *put* the data block again in the DHT (in the case the data block has been lost).

3.2 Side effects and limitations

Our replication strategy for peer-to-peer DHTs, by relaxing placement constraints of data block copies in leafsets, 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, but it implies other effects. The two main ones concern the data block distribution on the peers and the lookup performance. While the changes in data blocks distribution can provide positive effects, the lookup performance can be damaged.

Data blocks distribution

While with usual replication strategies in peer-to-peer 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 of the system. With both leafset-based replication and multiple key based replication, this remains true even when peers leave or join the system, due to the maintenance algorithms. 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: more stable a peer is, more data blocks it will store. Furthermore, it is possible to counter this effect easily by taking into account the quantity of stored data blocks while randomly choosing peers to add in replica-sets.

Lookup performance

We have focused our research efforts on data loss. We show in the next section that for equivalent churn patterns, the quantity of data lost using RelaxDHT is considerably lower than the quantity of data lost using a standard strategy like PAST's one. However, with RelaxDHT, it is possible that temporarily some data block roots are not consistent, inducing a network overhead to find the data. For example, when a peer which is root for at least one data block fails, the data block copies are still in the system but the standard lookup mechanism may not find them: the new peer whose identifier is the closest may not know the data block. This remains true until the failure is detected by one of the peer in the replica-set and repaired using a "new root" message (see algorithms above).

It would be possible to flood the leafset or to perform a "limited range walk" when a lookup fails, allowing lookups to find data blocks even in the absence of root, but this solution may slow down lookups and induce network overhead. However, notice that 1) lookups that occur between a root peer failure and its reparation are rare, 2) this could be done in conjunction with the leafset maintenance protocol (which already use flooding to maintain leafset).

Using the standard leafset replication scheme, this problem does not exist: if the root peer fails, one of its direct neighbors instantaneously becomes the new root (direct neighbors already store a copy of the data block). At last, with the multiple key replication, lookups are generally launched in parallel on multiple roots. As soon as one of the root answers, the lookup is considered as successful.

4 Evaluation

This section provides a comparative evaluation of RelaxDHT and PAST [1]. 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 up to two when comparing both systems.

4.1 Experimental setup

To evaluate RelaxDHT, we have build a discrete event simulator using the PeerSim [12] simulation kernel. We have based our simulator on an already existing PeerSim module simulating the Pastry KBR layer. We have implemented both the PAST strategy and the RelaxDHT strategy on top of this module. It is important to notice that all the different layers and all message exchanges are simulated. Our simulator also takes into account network congestion: in our case, network links may often be congested.

Simulation parameters

For all the simulation results presented in the section, we used a 100-peer network with the following parameters (for both PAST and RelaxDHT):

- a leafset size of 24;
- an inter-maintenance duration of 10 minutes at the DHT level;
- an inter-maintenance duration of 1 minute at the KBR level;
- 10 000 data blocks of 10 000 KB replicated 3 times;
- network links of 1 Mbits/s for upload and 10 mbits/s for download with a delay uniformly chosen between 80 and 120 ms.

A 100-peer network may seem a relatively small scale. However, for both replication strategies, PAST and RelaxDHT, the studied behavior is local, contained within a leafset (which size is bounded). It is however necessary to simulate a whole ring in order to take into account side effects induced by the neighbor leafsets. Furthermore, a tradeoff has to be made between system accuracy and system size. In our case, it is important to simulate very precisely all peer communications. We have run several simulations with a larger scale (1000 peers and 100,000 data blocks) and have observed similar phenomenons.

We have injected churn following three different scenarii:

- **One hour churn**. One perturbation phase with churn during one hour. This phase is followed by another phase without connections/disconnections. In this case study, during the churn phase each *perturbation period* we chose randomly 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.
- **Continuous churn**. For this set of simulations, we focus on phase one of the previous case. We study the system while varying the inter-perturbation delay. In this case, "perturbation" can be either a new peer connection or a disconnection.

We also experiment a scenario for which only one peer gets disconnected. We then study the reaction of the system.

The first set of experiments allows us to study 1) how many data blocks are lost after a period of perturbation 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. In real-life systems there will be some period without churn, the system has to take advantage of them to converge to a safer state.

The second set of experiments zooms on the perturbation period. It provides the ability to study how the system can resist when it has to repair lost copies in presence of churn.

Finally, the last set of simulations is done to measure the reparation of one single failure.

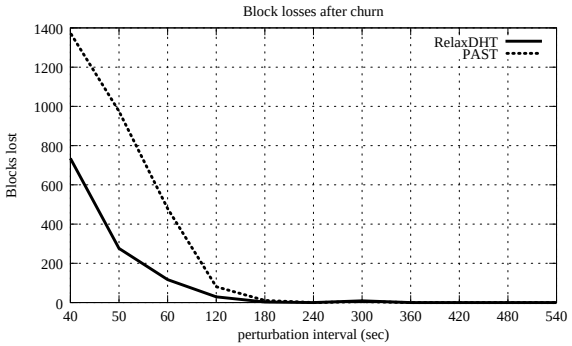

Figure 5: Number of data block lost (ie. all copies are lost).

4.2 Losses and stabilization time after 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 5 shows the number of lost data blocks after a period of one hour of churn. The inter-perturbation delay is increasing along the X axis. With RelaxDHT and our maintenance protocol, the number of lost data blocks is much lower than with the PAST's one: it reaches 50% for perturbations interval from lower than 50 *seconds*.

The main reason of the result presented above is that, using PAST replication strategy, the peers have more data blocks to download. This 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 traffic whose goal is to restore lost data block copies.

Figure 6 shows the total number of blocks exchanged for 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 near 2 times smaller than in PAST. This is mainly due to the fact that in PAST case, many transfers (near half of them) are only done to preserve the replication scheme constraints. For instance, each time a new peer joins the DHT, it becomes root of some data blocks (because its identifier is closer than the current root-peer's one), or if it is inserted within replica-sets that should remain contiguous.

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 our case, *no* or very *few* data blocks transfers are required when new peers join the system. It becomes mandatory, only if some copies becomes too far from their root-peer in the logical ring. 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.

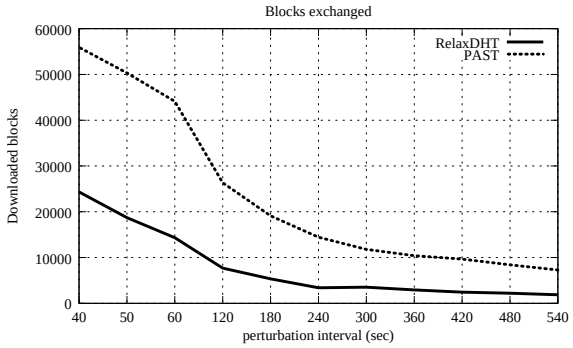

Figure 6: Number of exchanged data blocks to restore a stable state.

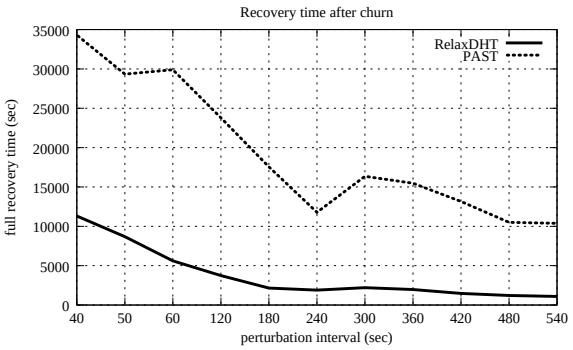

Figure 7: Recovery time: time for retreiving all the copies of every remaining data block.

Finally, we have measured the time the system takes to return in a normal state in which every remaining[3] data block is replicated `k` times. Figure 7 shows the results obtained while varying the delay between perturbations. We can observe that the recovery time is twice longer in the case where PAST is used compared to RelaxDHT. This result is mainly explained by the number of blocks to transfer which is much more lower in our case: our maintenance protocol transfers only very few blocks for location 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 implies that it will recover very fast, which means it will be able to cope with a new churn phase. The next section describes our simulations with continuous churn.

4.3 Continuous churn

Before presenting simulation results under continuous churn, it is important to measure 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 such a new peer in order to rebuild

[3]Blocks for which all copies are lost will never retreive a normal state and thus are not taken into account.

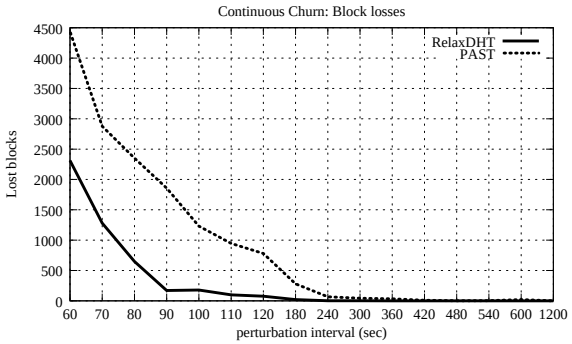

Figure 8: Number of data blocks losses (all k copies lost) while the system is under continuous churn, varying inter-perturbation delay.

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 much more important indeed. This gain is due to the parallelization of the data blocks-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, most of the peers contained in the faulty peer leafset (the leafset contains 24 peers in our simulations) may be involved in the transfers.

The above simulation results show that RelaxDHT: 1) induce 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 will provide a better churn tolerance.

Such results are illustrated in Figure 8. We can observe that, using the parameters described at the beginning of this section, PAST starts to lose data blocks when the inter-perturbation delay is around 7 minutes. This delay has to reach less than 4 minutes for data blocks to be lost using RelaxDHT. If the inter-perturbation delay continues to decrease, the number of lost data blocks using RelaxDHT strategy remains near half the number of data blocks lost using PAST strategy.

Finally, Figure 9 confirms that even with a continuous churn pattern, during a 5 hour run, the number of data transfers required by the proposed solution is much smaller (around half) than the number of data transfers induced by PAST's replication strategy.

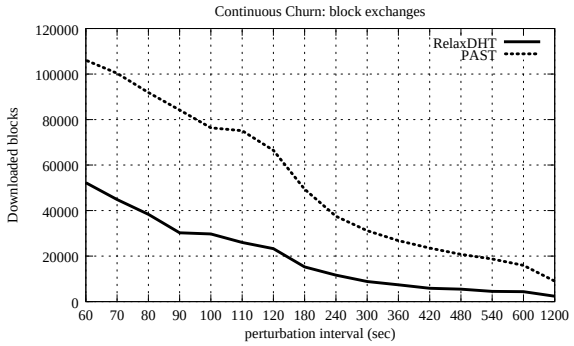

Figure 9: Number of data blocks transfers required while the system is under continuous churn, varying inter-perturbation delay.

4.4 Maintenance protocol cost

In the simulation results presented above, we have considered that maintenance protocol message size was negligible.

Both PAST and RelaxDHT maintenance protocols require $N * m$ messages, where N is the number of peers in the whole system and m is the $leafset\_size$. We explain below that in absence of churn, while using RelaxDHT, m can be reduced to the nodes in the center of the leafset (smaller than $leafset\_size$).

For PAST, each peer periodically sends a maintenance message to each node of its leafset. This message has to contain the identifier of each stored block: an average of $\frac{M*k}{N} identifiers$, where M is the total number of blocks in the system and k the mean replication factor. A peer stores data blocks for which it is the root, but also copies of data blocks for which its immediate $k-1$ logical neighbors are root. Therefore each peer sends and receives $(\frac{M*k}{N}) * leafset\_size * Id\_size$ at each period (this can be lowered through the use of bloom filters).

For RelaxDHT, in absence of churn the messages contain only STORE message elements. A peer is root of an average of $\frac{M}{N}$ data blocks which are replicated in average on k peers distributed in the center of the leafset: the m inner peers. This implies that the *average* number of STORE elements per message is: $\frac{M}{N} * \frac{k}{m}$ blocks for each of the peers in its leafset. Furthermore, if a replica-set has not changed since last maintenance, it is not necessary to send the replica-set again to all of its members[4]. Therefore, each maintenance message in absence of churn has to contain identifiers of each block for which the source is the root and the destination is part of the replica-set: an average of $\frac{M}{N} * \frac{k}{m} * (m) = \frac{M*k}{N}$ identifiers, which is $leafset\_size$ times lower than in the PAST case.

PAST uses bloom filters to convey identifier lists. In absence of churn, i.e., when the leafset is equal to the one at the previous period, it is also possible to use bloom filters in RelaxDHT.

In presence of churn, however, it becomes difficult to use bloom filters with RelaxDHT as message elements have a structure (data block identifiers associated to peer identifiers). For each block identifier, it may be necessary to send the block identifier and the peer identifiers of the members of the block's replica-set (k peers in average). Thus, if we put aside the bloom-filter opti-

[4]This optimization is very easy to implement.

mization, in our case each peer sends/receives $\frac{M*k^2}{N}$ identifiers at each period while peers using PAST send/receive $\frac{M*k*leafset\_size}{N}$ identifiers at each period; k being usually an order of magnitude lower than $leafset\_size$.

This 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 compute locally the information that needs to be transferred from one peer to another.

A smart implementation of RelaxDHT should try to use bloom filters whenever it is possible. To put it in a nutshell, the cost of our maintenance protocol is close to the cost of PAST maintenance protocol.

5 Conclusion

Peer to peer distributed hash tables provide an efficient, scalable and easy-to-use storage system. However, existing solutions do not tolerate a high churn rate or are not really scalable in terms of number of stored data blocks. We have identified the reasons why they do not tolerate high churn rate: they impose strict placement constraints that induces 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 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. It results that in case of failure more data sources are available for the recovery, 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) recovers lost data block copies faster and 3) loses less data blocks. Furthermore, we have shown that the churn-resilience is obtained without adding a great maintenance overhead.

References

[1] A. I. T. Rowstron and P. Druschel, "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*, December 2001, pp. 188–201.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, February 2003.

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

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

[5] F. Dabek, F. M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP '01: Proceedings of the 8th ACM symposium on Operating Systems Principles*, vol. 35, no. 5. New York, NY, USA: ACM Press, December 2001, pp. 202–215.

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

[7] R. Rodrigues and C. Blake, "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, February 2004, pp. 112–122.

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

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

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172.

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

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

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

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

[15] P. Maymounkov and D. Mazieres, "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, March 2002, pp. 53–65.

[16] F. Dabek, J. Li, E. Sit, J. Robertson, F. F. Kaashoek, and R. Morris, "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, March 2004.

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

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

[19] A. Ghodsi, L. O. Alima, and S. Haridi, "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, August 2005, p. 12.

[20] C. Harvesf and D. M. Blough, "The effect of replica placement on routing robustness in distributed hash tables," in *P2P '06: Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing*. Washington, DC, USA: IEEE Computer Society, September 2006, pp. 57–6.

[21] Q. Lian, W. Chen, and Z. Zhang, "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*. Washington, DC, USA: IEEE Computer Society, June 2005, pp. 187–196.

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

[23] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "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, December 2002.

[24] K. Kim and D. Park, "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, September 2006, pp. 915–924.

Contents

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399