# Data Currency in Replicated DHTs[1]

Reza Akbarinia Esther Pacitti Patrick Valduriez

Atlas team, INRIA and LINA

University of Nantes, France

{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}

## ABSTRACT

Distributed Hash Tables (DHTs) provide a scalable solution for data sharing in P2P systems. To ensure high data availability, DHTs typically rely on data replication, yet without data currency guarantees. Supporting data currency in replicated DHTs is difficult as it requires the ability to return a current replica despite peers leaving the network or concurrent updates. In this paper, we give a complete solution to this problem. We propose an Update Management Service (UMS) to deal with data availability and efficient retrieval of current replicas based on timestamping. For generating timestamps, we propose a Key-based Timestamping Service (KTS) which performs distributed timestamp generation using local counters. Through probabilistic analysis, we compute the expected number of replicas which UMS must retrieve for finding a current replica. Except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of available replicas are current then the expected number of retrieved replicas is less than 3. We validated our solution through implementation and experimentation over a 64-node cluster and evaluated its scalability through simulation up to 10,000 peers using SimJava. The results show the effectiveness of our solution. They also show that our algorithm used in UMS achieves major performance gains, in terms of response time and communication cost, compared with a baseline algorithm.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *distributed databases, concurrency, query processing.*

## General Terms

Algorithms, performance, reliability.

## Keywords

Peer-to-Peer, distributed hash table (DHT), data availability, data currency, data replication

[1] Work partially funded by ARA "Massive Data" of the French ministry of research and the European Strep Grid4All project.

## 1. INTRODUCTION

Peer-to-peer (P2P) systems adopt a completely decentralized approach to data sharing and thus can scale to very large amounts of data and users. Popular examples of P2P systems such as Gnutella [9] and KaaZa [12] have millions of users sharing petabytes of data over the Internet. Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems, such as Gnutella and KaaZa, which rely on flooding. This work led to structured solutions based on distributed hash tables (DHT), *e.g.* CAN [19], Chord [29], and Pastry [23]. While there are significant implementation differences between DHTs, they all map a given key *k* onto a peer *p* using a hash function and can lookup *p* efficiently, usually in O(*log n*) routing hops where *n* is the number of peers [5]. DHTs typically provide two basic operations [5]: *put(k, data)* stores a key *k* and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with *k* in the DHT.

One of the main characteristics of P2P systems is the dynamic behavior of peers which can join and leave the system frequently, at anytime. When a peer gets offline, its data becomes unavailable. To improve data availability, most DHTs rely on data replication by storing (*k, data*) pairs at several peers, *e.g.* using several hash functions [19]. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. However, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Let us illustrate the problem with a simple update scenario in a typical DHT. Let us assume that the operation *put*($k$, $d_0$) (issued by some peer) maps onto peers $p_1$ and $p_2$ which both get to store the data $d_0$. Now consider an update (from the same or another peer) with the operation *put*($k$, $d_1$) which also maps onto peers $p_1$ and $p_2$. Assuming that $p_2$ cannot be reached, *e.g.* because it has left the network, then only $p_1$ gets updated to store $d_1$. When $p_2$ rejoins the network later on, the replicas are not consistent: $p_1$ holds the *current* state of the data associated with *k* while $p_2$ holds a *stale* state. Concurrent updates also cause inconsistency. Consider now two updates *put*($k$, $d_2$) and *put*($k$, $d_3$) (issued by two different peers) which are sent to $p_1$ and $p_2$ in reverse order, so that $p_1$'s last state is $d_2$ while $p_2$'s last state is $d_3$. Thus, a subsequent *get(k)* operation will return either stale or current data depending on which peer is looked up, and there is no way to tell whether it is current or not. For some applications (*e.g.* agenda management, bulletin boards, cooperative auction management, reservation management, etc.) which could take advantage of a DHT, the ability to get the current data is very important.

Many solutions have been proposed in the context of distributed database systems for managing replica consistency [17] but the high numbers and dynamic behavior of peers make them no longer applicable to P2P [6]. Supporting data currency in replicated DHTs requires the ability to return a current replica despite peers leaving the network or concurrent updates. The problem is partially addressed in [13] using data versioning. Each replica has a version number which is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica.

In this paper, we give a complete solution to data availability and data currency in replicated DHTs. Our main contributions are the following:

- We propose a service called Update Management Service (UMS) which deals with improving data availability and efficient retrieval of current replicas based on timestamping. After retrieving a replica, UMS detects whether it is current or not, *i.e.* without having to compare with the other replicas, and returns it as output. Thus, in contrast to the solution in [13], UMS does not need to retrieve all replicas to find a current one. In addition, concurrent updates raise no problem for UMS.
- We give a probabilistic analysis of UMS's communication cost. We compute the expected number of replicas which UMS must retrieve for finding a current replica. We prove that it is less than the inverse of the probability of currency and availability, *i.e.* the probability that a replica is current and available. Thus, except for the cases where the availability of current replicas is very low, the expected number of replicas which UMS must retrieve is typically small.
- We propose a new Key-based Timestamping Service (KTS) which generates monotonically increasing timestamps, in a distributed fashion using local counters. KTS does distributed timestamp generation in a way that is similar to data storage in the DHT, *i.e.* using peers dynamically chosen by hash functions. To maintain timestamp monotonicity, we propose algorithms which take into account the cases where peers leave the system either normally or not (*e.g.* because they fail). To the best of our knowledge, this is the first paper that introduces the concept of key-based timestamping, and proposes efficient techniques for realizing this concept in DHTs. Furthermore, KTS is useful to solve other DHT problems which need a total order on operations performed on each data, *e.g.* read and write operations which are performed by concurrent transactions.
- We provide a comprehensive performance evaluation based on the implementation of UMS and KTS over a 64-node cluster. We also evaluated the scalability of our solution through simulation up to 10,000 peers using SimJava. The experimental and simulation results show the effectiveness of our solution.

The rest of this paper is organized as follows. In Section 2, we first propose a model for DHTs which will be useful to present our solution, and then we state the problem. Section 3 presents our update management service for DHTs. In Section 4, we propose a distributed timestamping service to support updates. Section 5 describes a performance evaluation of our solution through implementation and simulation. In Section 6, we discuss related work. Section 7 concludes.

## 2. DHT MODEL AND PROBLEM STATEMENT

In this section, we first present a model of DHTs which is needed for describing our solution and proving its properties. Then, we precisely state the problem.

### 2.1 DHT Model

A DHT maps a key $k$ to a peer $p$ using a hash function $h$. We call $p$ the *responsible for k wrt h*. A peer may be responsible for $k$ wrt a hash function $h_1$ but not responsible for $k$ wrt another hash function $h_2$. The responsible for $k$ wrt $h$ may be different at different times, *i.e.* because of peers' joins and leaves. We can model the mapping mechanism of DHT as a function that determines at anytime the peer that is responsible for $k$ wrt $h$; we call this function *DHT's mapping function*.

**Definition 1: DHT's mapping function.** *Let K be the set of all keys accepted by the DHT, P the set of peers, H the set of all pairwise independent hash functions which can be used by the DHT for mapping, and T the set of all numbers accepted as time. We define the DHT's mapping function as* $m: K \times H \times T \rightarrow P$ *such that* $m(k,h,t)$ *determines the peer* $p \in P$ *which is responsible for* $k \in K$ *wrt* $h \in H$ *at time* $t \in T$.

Let us make precise the terminology involving peers' responsibility for a key. Let $k \in K$, $h \in H$ and $p \in P$, and let $[t_0..t_1)$ be a time interval such that $t_1 > t_0$. We say that $p$ is *continuously responsible* for $k$ wrt $h$ in $[t_0..t_1)$ if it is responsible for $k$ wrt $h$ at anytime in $[t_0..t_1)$. In other words, $(\forall t \in T,\ t_0 \leq t < t_1\ ) \Rightarrow (\ p = m(k,h,t))$. If $p$ obtains and loses the responsibility for $k$ wrt $h$ respectively at $t_0$ and $t_1$, and is continuously responsible for $k$ wrt $h$ in $[t_0..t_1)$, then we say that $[t_0..t_1)$ is a *p's period of responsibility* for $k$ wrt $h$. The peer that is responsible for $k$ wrt $h$ at current time is denoted by *rsp(k,h)*. We also denote by *prsp(k,h)* the peer that was responsible for $k$ wrt $h$ just before *rsp(k,h)*. The peer that will become responsible for $k$ wrt $h$ just after *rsp(k,h)* is denoted by *nrsp(k,h)*.

**Example 1**. Figure 1 shows the peers responsible for $k \in K$ wrt $h \in H$ since $t_0$. The peer that is currently responsible for $k$ wrt $h$ is $p_1$, thus $p_1 = rsp(k,h)$ and $p_3 = prsp(k,h)$. In the time interval $[t_1..t_2)$, $p_2$ is continuously responsible for $k$ wrt $h$. It has obtained and lost its responsibility respectively at $t_1$ and $t_2$, thus $[t_1..t_2)$ is $p_2$'s period of responsibility for $k$ wrt $h$. Also $[t_0..t_1)$ and $[t_2..t_3)$ are respectively $p_4$'s and $p_3$'s periods of responsibility for $k$ wrt $h$.

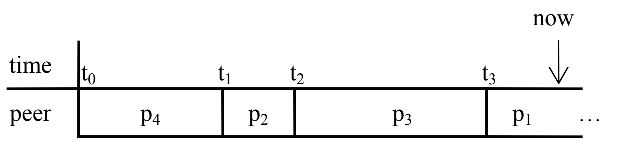

**Figure 1. Example of peers' responsibilities**

In the DHT, there is a *lookup service* that can locate *rsp(k,h)* efficiently. The lookup service can return the address of *rsp(k,h)* usually in $O(Log\ |P|)$ routing hops, where $|P|$ is the number of peers in the system.

## 2.2 Problem Statement

To improve data availability we replicate the pairs *(k, data)* at several peers using several hash functions. We assume that there is an operation that stores a pair *(k, data)* at *rsp(k,h)* which we denote by $put_h(k, data)$. This operation can be issued concurrently by several peers. There is another operation, denoted by $get_h(k)$, that retrieves the data associated with *k* which is stored at *rsp(k,h)*.

Over time, some of the replicas stored with *k* at some peers may get stale. Our objective is to provide a mechanism which returns efficiently a current replica in response to a query requesting the data associated with a key.

Formally, the problem can be defined as follows. Given a key $k \in K$, let $R_k$ be the set of replicas such that for each $r \in R_k$, the pair *(k, r)* is stored at one of the peers of the DHT. Our goal is to return efficiently an $r \in R_k$ which is current, *i.e.* reflects the latest update.

## 3. UPDATE MANAGEMENT SERVICE

To deal with data currency in DHTs, we propose an *Update Management Service (UMS)* which provides high data availability through replication and efficient retrieval of current replicas. UMS only requires the DHT's lookup service with $put_h$ and $get_h$ operations. To return current replicas, it uses timestamps attached to the pairs *(k, data)*. In this section, we give an overview of our timestamping solution and present in more details UMS' update operations. We also analyze UMS's communication cost.

## 3.1 Timestamping

To provide high data availability, we replicate the data in the DHT using a set of pairwise independent hash functions $H_r \subset H$ which we call *replication hash functions*. To be able to retrieve a current replica we "stamp" each pair *(k, data)* with a logical timestamp, and for each $h \in H_r$ we replicate the pair *(k, newData)* at *rsp(k,h)* where *newData={data, timestamp}*, i.e. newData is a data composed of the initial data and the timestamp. Upon a request for the data associated with a key, we can thus return one of the replicas which are stamped with the latest timestamp. The number of replication hash functions, *i.e.* $|H_r|$, can be different for different DHTs. For instance, if in a DHT the availability of peers is low, for increasing data availability a high value of $|H_r|$ (*e.g.* 30) is used. Constructing $H_r$, which is a set of pairwise independent hash functions, can be done easily, *e.g.* by using the methods presented in [14].

To generate timestamps, we propose a distributed service called *Key-based Timestamping Service (KTS)*. The main operation of KTS is *gen_ts(k)* which given a key *k* generates a real number as a *timestamp for k*. The timestamps generated by KTS have the *monotonicity* property, *i.e.* two timestamps generated for the same key are monotonically increasing. This property permits us to order the timestamps generated for the same key according to the time at which they have been generated.

**Definition 2: Timestamp monotonicity**. *For any two timestamps* $ts_1$ *and* $ts_2$ *generated for a key k respectively at times* $t_1$ *and* $t_2$, *if* $t_1 < t_2$ *then we have* $ts_1 < ts_2$.

At anytime, KTS generates at most one timestamp for a key (see Section 4 for the details). Thus, regarding to the monotonicity property, there is a total order on the set of timestamps generated

```
insert(k, data)
begin
   ts := KTS.gen_ts (k);
   for each $h \in H_r$ do
      newData := {data, ts};
      DHT.$put_h$(k, newData);
end;

retrieve(k)
begin
   $ts_1$ := KTS.last_ts(k);
   $data_{mr}$ := null;
   $ts_{mr}$ := - ∞;
   for each $h \in H_r$ do begin
       newData := DHT.$get_h$(k);
       data := newData.data;
       ts := newData.ts;
       if ($ts_1$ = ts) then begin
          return data; // one current
                       // replica is found
          exit;
       end
       else if (ts > $ts_{mr}$) then begin
          $data_{mr}$ := data;//keep the most
          $ts_{mr}$ := ts;//recent replica and
                      //its timestamp
       end;
   end;
   return $data_{mr}$
end;
```

**Figure 2. UMS update operations**

for the same key. However, there is no total order on the timestamps generated for different keys.

KTS has another operation denoted by *last_ts(k)* which given a key *k* returns the last timestamp generated for *k* by KTS.

## 3.2 Update Operations

To describe UMS, we use the *KTS.gen_ts* and *KTS.last_ts* operations discussed above. The implementation of these operations is detailed in Section 4. UMS provides *insert* and *retrieve* operations (see Figure 2).

**Insert(k, data):** inserts a pair *(k, data)* in the DHT as follows. First, it uses KTS to generate a timestamp for *k*, *e.g. ts*. Then, for each $h \in H_r$ it sends the pair *(k, {data, ts})* to the peer that is *rsp(k,h)*. When a peer *p*, which is responsible for *k* wrt one of the hash functions involved in $H_r$, receives the pair *(k, {data, ts})*, it compares *ts* with the timestamp, say $ts_0$, of its data (if any) associated with *k*. If $ts > ts_0$, *p* overwrites its data and timestamp with the new ones. Recall that, at anytime, *KTS.gen_ ts (k)* generates at most one timestamp for *k*, and different timestamps for *k* have the monotonicity property. Thus, in the case of concurrent calls to *insert(k, data)*, *i.e.* from different peers, only the one that obtains the latest timestamp will succeed to store its data in the DHT.

**Retrieve(k)**: retrieves the most recent replica associated with *k* in the DHT as follows. First, it uses KTS to determine the latest timestamp generated for *k*, *e.g.* $ts_1$. Then, for each hash function $h \in H_r$, it uses the DHT operation $get_h(k)$ to retrieve the pair *{data, timestamp}* stored along with *k* at *rsp(k,h)*. If *timestamp* is equal to $ts_1$, then the data is a current replica which is returned as output and the operation ends. Otherwise the retrieval process continues while saving in $data_{mr}$ the most recent replica. If no replica with a timestamp equal to $ts_1$ is found (*i.e.* no current replica is found) then the operation returns the most recent replica which is available, *i.e.* $data_{mr}$.

## 3.3 Cost Analysis

In this section, we give a probabilistic analysis of the communication cost of UMS in terms of number of messages to retrieve a data item. For a non replicated DHT, this cost, which we denote by $c_{ret}$, is *O(log n)* messages where *n* is the number of peers. The communication cost of retrieving a current replica by UMS is $c_{ums} = c_{kts} + n_{ums} * c_{ret}$, where $c_{kts}$ is the cost of returning the last generated timestamp by KTS and $n_{ums}$ is the number of replicas that UMS retrieves, *i.e.* the number of times that the operation $get_h(k)$ is called. As we will see in the next section, $c_{kts}$ is usually equal to $c_{ret}$, *i.e.* the cost of contacting the responsible of a key and getting the last timestamp from it. Thus, we have $c_{ums} = (1 + n_{ums}) * c_{ret}$.

The The number of replicas which UMS retrieves, *i.e.* $n_{ums}$, depends on the probability of currency and availability of replicas. The higher this probability, the lower $n_{ums}$ is. Let $H_r$ be the set of replication hash functions, *t* be the retrieval time, and $p_t$ be the probability that, at time *t,* a current replica is available at a peer that is responsible for *k* wrt some $h \in H_r$. In other words, $p_t$ is the ratio of current replicas, which are available at *t* over the peers responsible for *k* wrt replication hash functions, to the total number of replicas, *i.e.* $|H_r|$. We call $p_t$ the *probability of currency and availability* at retrieval time. We give a formula for computing the expected value of the number of replicas, which UMS retrieves, in terms of $p_t$ and $|H_r|$. Let *X* be a random variable which represents the number of replicas that UMS retrieves. We have $Prob(X=i) = p_t * (1- p_t)^{i-1}$, *i.e.* the probability of having *X=i* is equal to the probability that *i-1* first retrieved replicas are not current and the *i*th replica is current. The expected value of *X* is computed as follows:

$$E(X) = \sum_{i=0}^{|H_r|} i * \Pr ob(X = i)$$

$$E(X) = p_t * (\sum_{i=0}^{|H_r|} i * (1 - p_t)^{i-1}) \qquad (1)$$

Equation 1 expresses the expected value of the number of retrieved replicas in terms of $p_t$ and $|H_r|$. Thus, we have the following upper bound for *E(X)* which is solely in terms of $p_t$:

$$E(X) < p_t * (\sum_{i=0}^{\infty} i * (1 - p_t)^{i-1}) \qquad (2)$$

From the theory of series [2], we use the following equation for $0 \leq z < 1$:

$$\sum_{i=0}^{\infty} i * z^{i-1} = \frac{1}{(1-z)^2}$$

Since $0 \leq (1- p_t) < 1$, we have:

$$\sum_{i=0}^{\infty} i * (1 - p_t)^{i-1} = (\frac{1}{(1-(1-p_t))^2} \qquad (3)$$

Using Equations 3 and 2, we obtain:

$$E(X) < \frac{1}{p_t} \qquad (4)$$

**Theorem 1:** *The expected value of the number of replicas which UMS retrieves is less than the inverse of the probability of currency and availability at retrieval time.*

**Proof:** Implied by the above discussion.□

**Example.** Assume that at retrieval time *35%* of replicas are current and available, *i.e.* $p_t=0.35$. Then the expected value of the number of replicas which UMS retrieves is less than 3.

Intuitively, the number of retrieved replicas cannot be more than $|H_r|$. Thus, for *E(X)* we have:

$$E(X) \leq \min(\frac{1}{p_t}, |H_r|) \qquad (5)$$

## 4. KEY-BASED TIMESTAMP SERVICE

The main operation of KTS is *gen_ts* which generates monotonically increasing timestamps for keys. A centralized solution for generating timestamps is obviously not possible in a P2P system since the central peer would be a bottleneck and single point of failure. Distributed solutions using synchronized clocks no longer apply in a P2P system. One popular method for distributed clock synchronization is Network Time Protocol (NTP) which was originally intended to synchronize computers linked via Internet networks [16]. NTP and its extensions (*e.g.* [8] and [18]) guarantee good synchronization precision only if computers have been linked together long enough and communicate frequently [18]. However, in a P2P system in which peers can leave the system at any time, these solutions cannot provide good synchronization precision.

In this section, we propose a distributed technique for generating timestamps in DHTs. First, we present a technique based on local counters for generating the timestamps. Then we present a direct algorithm and an indirect algorithm for initializing the counters, which is very important for guaranteeing the monotonicity of timestamps. We also apply the direct algorithm to CAN and Chord. Finally, we discuss a method for maintaining the validity of counters.

## 4.1 Timestamp Generation

Our idea for timestamping in DHTs is like the idea of data storage in these networks which is based on having a peer responsible for storing each data and determining the peer dynamically using a hash function. In KTS, for each key we have a peer responsible for timestamping which is chosen dynamically using a hash function. Below, we discuss the details of timestamp responsibility and timestamp generation.

### *4.1.1 Timestamping Responsibility*

Timestamp generation is performed by KTS as follows. Let $k \in K$ be a key, the *responsible of timestamping for k* is the peer that is responsible for *k* wrt $h_{ts}$, *i.e.* $rsp(k, h_{ts})$, where $h_{ts}$ is a hash function accepted by the DHT, i.e. $h_{ts} \in H$. Each peer *q* that needs a timestamp for *k,* called *timestamp requester*, uses the DHT's lookup service to obtain the address of $rsp(k, h_{ts})$ to which it sends a *timestamp request (TSR)*. When $rsp(k, h_{ts})$ receives the request of *q*, generates a timestamp for *k* and returns it to *q*. Figure 3 illustrates the generation of a timestamp for *k* initiated by peer *q*.

If the peer that is *rsp(k, $h_{ts}$)* leaves the system or fails, the DHT detects the absence of that peer, *e.g.* by frequently sending "ping" messages from each peer to its neighbors [19], and another peer becomes responsible for *k* wrt $h_{ts}$. Therefore, if the responsible of timestamping for *k* leaves the system or fails, another peer automatically becomes responsible of timestamping for *k*, *i.e.* the peer that becomes responsible for *k* wrt $h_{ts}$. Thus, the dynamic behavior of peers causes no problem for timestamping responsibility.

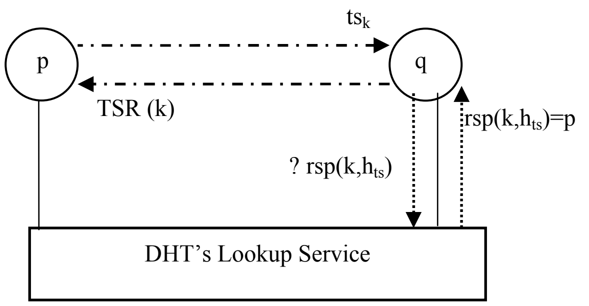

**Figure 3. Example of timestamp generation**

### *4.1.2 Guaranteeing Monotonicity*

Let us now discuss what a responsible of timestamping should do to maintain the monotonicity property. Let *k* be a key, *p* the peer that is responsible of timestamping for *k*, and $ts_k$ a timestamp for *k* which is generated by *p*. To provide the monotonicity property, we must guarantee two constraints: (1) $ts_k$ is greater than all timestamps for *k* which have been previously generated by *p* itself; (2) $ts_k$ is greater than any timestamp for *k* generated by any other peer that was responsible of timestamping for *k* in the past.

To enforce the first constraint, for generating timestamps for each key *k*, we use a local *counter of k at p* which we denote as $c_{p,k}$. When *p* receives a timestamp request for *k*, it increments the value of $c_{p,k}$ by one and returns it as the timestamp for *k* to the timestamp requester.

To enforce the second constraint, *p* should *initialize* $c_{p,k}$ so that it is greater than or equal to any timestamp for *k* previously generated by other peers that were responsible of timestamping for *k* in the past. For this, *p* initializes $c_{p,k}$ to the last value of $c_{q,k}$ where *q* is the last peer that has generated a timestamp for *k*. In Section 4.2, we discuss how *p* can acquire $c_{q,k}$. The following lemma shows that the initialization of $c_{p,k}$ as above enforces the second constraint.

**Lemma 1**: *If each peer p, during each of its periods of responsibility for k wrt $h_{ts}$, initializes $c_{p,k}$ before generating the first timestamp for k, then each generated timestamp for k is greater than any previously generated one.*

**Proof:** Follows from the fact that initializing $c_{p,k}$ makes it equal to the last timestamp generated for *k*, and the fact that timestamp generation is done by increasing the value of $c_{p,k}$ by one and returning its value as output. □

After $c_{p,k}$ has been initialized, it is a *valid counter*, *i.e. p* can use it for generating timestamps for *k*. If *p* loses the responsibility for *k* wrt $h_{ts}$, *e.g.* because of leaving the system, then $c_{p,k}$ becomes *invalid*. The peer *p* keeps its valid counters in a *Valid Counters Set* which we denote by $VCS_p$. In other words, for each $k \in K$, if

```
gen-ts(k) // timestamp generation by KTS
begin
   p := DHT.lookup(k, $h_{ts}$);
   return gen-ts(p, k);
end;

gen-ts(p, k) //generating a timestamp
            // for a key k by peer p
            // that is rsp(k, $h_{ts}$)
begin
   $c_{p,k}$ := search_counter($VCS_p$, k);
   if ($c_{p,k}$ is not in $VCS_p$) then
   begin
      new($c_{p,k}$);//allocate memory for $c_{p,k}$
      KTS.CounterInitialize(k, $c_{p,k}$);
      $VCS_p$ := $VCS_p$ + {$c_{p,k}$};
   end;
   $c_{p,k}$.value := $c_{p,k}$.value + 1;
   return $c_{p,k}$.value;
end;
```

**Figure 4. Timestamp generation**

$c_{p,k}$ is valid then $c_{p,k}$ is in $VCS_p$. Each peer $p \in P$ has its own $VCS_p$ and respects the following rules for it:

1. When *p* joins the P2P system, it sets $VCS_p = \varnothing$.
2. $\forall k \in K$, when *p* initializes $c_{p,k}$, it adds $c_{p,k}$ to $VCS_p$.
3. $\forall k \in K$, when *p* loses the responsibility for *k* wrt $h_{ts}$, if $c_{p,k}$ is in $VCS_p$ then *p* removes it from $VCS_p$.

When *p* receives a timestamp request for a key *k*, it checks for the existence of $c_{p,k}$ in $VCS_p$. If $c_{p,k}$ is in $VCS_p$ then *p* generates the timestamp for *k* using $c_{p,k}$. Otherwise *p* initializes $c_{p,k}$, appends it to $VCS_p$ and then generates the timestamp using $c_{p,k}$ (see Figure 4).

The data structure used for $VCS_p$ is such that given a key *k* seeking $c_{p,k}$ in $VCS_p$ can be done rapidly, *e.g.* a binary search tree. Also, for minimizing the memory cost, when a counter gets out of $VCS_p$, *p* releases the memory occupied by the counter, *i.e.* only the counters involved in $VCS_p$ occupy a memory location. To prevent the problem of overflow, we use a large integer, *e.g.* 128 bits, for the value of $c_{p,k}$.

The following theorem shows that using $VCS_p$ and respecting its rules guarantees the monotonicity property.

**Theorem 2**: *If the peer p, which is responsible for k wrt $h_{ts}$, for generating timestamps for k uses $c_{p,k}$ that is in $VCS_p$, then each generated timestamp for k is greater than any previously generated one.*

**Proof:** Let $[t_0, t_1)$ be a *p*'s period of responsibility for *k* wrt $h_{ts}$ and let us assume that *p* generates a timestamp for *k* in $[t_0, t_1)$. Rules 1 and 3 assure that at $t_0$, $c_{p,k}$ is not in $VCS_p$. Thus, for generating the first timestamp for *k* in $[t_0, t_1)$, *p* should initialize $c_{p,k}$ and insert it into $VCS_p$ (Rule 2). Therefore, in each of its periods of responsibility for *k* wrt $h_{ts}$, *p* initializes $c_{p,k}$ before generating the first timestamp for *k*. Thus, each peer *p*, during each of its periods of responsibility for *k* wrt $h_{ts}$, initializes $c_{p,k}$ before generating the first timestamp for *k*, so by Lemma 1 the proof is complete. □

The other KTS operation *last_ts(k)*, which we used in Section 3, can be implemented like *gen_ts* except that *last_ts* is simpler: it only returns the value of $c_{p,k}$ and does not need to increase its value.

## 4.2 Counter Initialization

Initializing the counters is very important for maintaining the monotonicity property. Recall that for initializing $c_{p,k}$, the peer *p*, which is responsible of timestamping for *k*, assigns to $c_{p,k}$ the value of $c_{q,k}$ where *q* is the last peer that has generated a timestamp for *k*. But, the question is how *p* can acquire $c_{q,k}$. To answer this question, we propose two *initialization algorithms*: *direct* and *indirect*. The direct algorithm is based on transferring directly the counters from a responsible of timestamping to the next responsible. The indirect algorithm is based on retrieving the value of the last generated timestamp from the DHT.

### *4.2.1 Direct Algorithm for Initializing Counters*

With the direct algorithm, the initialization is done by directly transferring the counters from a responsible of timestamping to the next one at the end of its responsibility. This algorithm is used in situations where the responsible of timestamping loses its responsibility in a normal way, *i.e.* it does not fail.

Let *q* and *p* be two peers, and $K' \subseteq K$ be the set of keys for which *q* is the current responsible of timestamping and *p* is the next responsible. The direct algorithm proceeds as follows. Once *q* reaches the end of its responsibility for the keys in *K'*, *e.g.* before leaving the system, it sends to *p* all its counters that have been initialized for the keys involved in *K'*. Let *C* be an empty set, *q* performs the following instructions at the end of its responsibility:

```
for each $c_{q,k} \in VCS_q$ do
    if ($k \in K'$) then
        C := C + {$c_{q,k}$};
Send C to p;
```

At the beginning of its responsibility for the keys in *K'*, *p* initializes its counters by performing the following instructions:

```
for each $c_{q,k} \in C$ do begin
  new($c_{p,k}$);
  $c_{p,k}$.value := $c_{q,k}$.value;
  $VCS_p$ := $VCS_p$ + {$c_{p,k}$};
end;
```

#### *4.2.1.1 Application to CAN and Chord*

The direct algorithm initializes the counters very efficiently, in O(1) messages, by sending the counters from the current responsible of timestamping to the next responsible at the end of its responsibility. But, how can the current responsible of timestamping find the address of the next responsible? The DHT's lookup service does not help here because it can only lookup the current responsible for *k*, *i.e.* $rsp(k, h_{ts})$, and cannot return the address of the next responsible for *k*. To answer the question, we observe that, in DHTs, the next peer that obtains the responsibility for a key *k* is typically a neighbor of the current responsible for *k*, so the current responsible of timestamping has the address of the next one. We now illustrate this observation with CAN and Chord, two popular DHTs.

Let us assume that peer *q* is *rsp(k,h)* and peer *p* is *nrsp(k,h)* where $k \in K$ and $h \in H$. In CAN and Chord, there are only two ways by which *p* would obtain the responsibility for *k* wrt *h*. First, *q* leaves the P2P system or fails, so the responsibility of *k* wrt *h* is assigned to *p*. Second, *p* joins the P2P system which assigns it the responsibility for *k* wrt *h*, so *q* loses the responsibility for *k* wrt *h* despite its presence in the P2P system. We show that in both cases, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. In other words, we show that both CAN and Chord have the important property that *nrsp(k,h) is one of the neighbors of rsp(k,h) at the time when rsp(k,h) loses the responsibility for k wrt h.*

**CAN.** We show this property by giving a brief explanation of CAN's protocol for joining and leaving the system [19]. CAN maintains a virtual coordinate space partitioned among the peers. The partition which a peer owns is called its zone. According to CAN, a peer *p* is responsible for *k* wrt *h* if and only if *h(k)*, which is a point in the space, is in *p*'s zone. When a new peer, say *p*, wants to join CAN, it chooses a point *X* and sends a join request to the peer whose zone involves *X*. The current owner of the zone, say *q*, splits its zone in half and the new peer occupies one half, then *q* becomes one of *p*'s neighbors. Thus, in the case of join, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. Also, when a peer *p* leaves the system or fails, its zone will be occupied by one of its neighbors, *i.e.* the one that has the smallest zone. Thus, in the case of leave or fail, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)* , and that neighbor is known for *rsp(k,h)*.

**Chord.** In Chord [29], each peer has an m-bit identifier (ID). The peer IDs are ordered in a circle and the neighbors of a peer are the peers whose distance from *p* clockwise in the circle is $2^i$ for $0 \le i \le m$. The responsible for *k* wrt *h* is the first peer whose ID is equal or follows *h(k)*. Consider a new joining peer *p* with identifier $ID_p$. Suppose that the position of *p* in the circle is just between two peers $q_1$ and $q_2$ with identifiers $ID_1$ and $ID_2$, respectively. Without loss of generality, we assume that $ID_1 < ID_2$, thus we have $ID_1 < ID_p < ID_2$. Before the entrance of *p*, the peer $q_2$ was responsible for *k* wrt *h* if and only if $ID_1 < h(k) \le ID_2$. When *p* joins Chord, it becomes responsible for *k* wrt *h* if and only if $ID_1 < h(k) \le ID_p$. In other words, *p* becomes responsible for a part of the keys for which $q_2$ was responsible. Since the distance clockwise from *p* to $q_2$ is $2^0$, $q_2$ is a neighbor of *p*. Thus, in the case of join, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. When, a peer *p* leaves the system or fails, the next peer in the circle, say $q_2$, becomes responsible for its keys. Since the distance clockwise from *p* to $q_2$ is $2^0$, $q_2$ is a neighbor of *p*.

Following the above discussion, when a peer *q* loses the responsibility for *k* wrt *h* in Chord or CAN, one of its neighbors, say *p*, is the next responsible for all keys for which *q* was responsible. Therefore, to apply the direct algorithm, it is sufficient that, before losing its responsibility, *q* sends to *p* its initialized counters, *i.e.* those involved in $VCS_q$.

### *4.2.2 Indirect Algorithm for Initializing Counters*

With the direct algorithm, the initialization of counters can be done very efficiently. However, in some situations the direct algorithm cannot be used, *e.g.* when a responsible of timestamping fails. In those situations, we use the indirect algorithm. For initializing the counter of a key *k*, the indirect algorithm retrieves the most recent timestamp which is stored in the DHT along with the pairs (*k, data)*. As described in Section 3.2, peers store the timestamps, which are generated by KTS, along with their data in the DHT.

The indirect algorithm for initializing the counters proceeds as follows (see Figure 5). Let *k* be a key, *p* be the responsible of timestamping for *k*, and $H_r$ be the set of replication hash functions

```
Indirect_Initialization(k, var $c_{p,k}$)
begin
   $ts_m$ := -1;
   for each  $h \in H_r$  do begin
      {data, ts} := DHT.get$_h$(k);
       if ($ts_m$ < ts) then
          $ts_m$ := ts;
   end;
   $c_{p,k}$.value := $ts_m$ + 1;
end;
```

**Figure 5. Indirect algorithm for initializing counters**

which are used for replicating the data in the DHT as described in Section 3.2. To initialize $c_{p,k}$ , for each $h \in H_r$, $p$ retrieves the replica (and its associated timestamp) which is stored at *rsp(k, h)*. Among the retrieved timestamps, $p$ selects the most recent one, say $ts_m$, and initializes $c_{p,k}$ to $ts_m + 1$. If no replica and timestamp is stored in the DHT along with $k$, then $p$ initializes $c_{p,k}$ to 0.

If $p$ is at the beginning of its responsibility of timestamping for $k$, before using the indirect algorithm, it waits a while so that the possible timestamps, which are generated by the previous responsible of timestamping, be committed in the DHT by the peers that have requested them.

Let $c_{ret}$ be the number of messages which should be sent over the network for retrieving a data from the DHT, the indirect algorithm is executed in $O(|H_r| * c_{ret})$ messages.

Let us now compute the probability that the indirect algorithm retrieves successfully the latest version of the timestamp from the DHT. We denote this probability as $p_s$. Let $t$ be the time at which we execute the indirect algorithm, and $p_t$ be the probability of currency and availability at $t$ (see Section 3.3 for the definition of the probability of currency and availability). If at least one of the peers, which are responsible for $k$ wrt replication hash functions, owns a current replica then the indirect algorithm works successfully. Thus, $p_s$ can be computed as follows:

*$p_s$ = 1 – (the probability that no current replica is available at peers which are responsible for k wrt replication hash functions)*

Thus, we have:

$$p_s = 1 - (1 - p_t)^{|H_r|}$$

In this equation, $|H_r|$ is the number of replication hash functions. By increasing the number of replication hash functions, we can obtain a good probability of success for the indirect algorithm. For instance, if the probability of currency and availability is about 30%, then by using 13 replication hash functions, $p_s$ is more than 99%.

By adjusting the number of replication hash functions, the probability of success of the indirect algorithm is high but not 100%. Thus, there may be some situations where it cannot retrieve the latest version of timestamp, in which case the counter of the key is not initialized correctly. To deal with these situations in a correct way, we propose the following strategies:

- **Recovery**. After restarting, the failed responsible of timestamping contacts the new responsible of timestamping, say $p$, and sends it all its counters. Then, the new responsible of timestamping compares the received counters with those initialized by the indirect algorithm and corrects the counters which are initialized incorrectly (if any). In addition, if $p$ has generated some timestamps with an incorrect counter, it retrieves the data which has been stored in the DHT with the latest value of the incorrect counter and reinserts the data into the DHT with the correct value of the counter.
- **Periodic inspection**. A responsible of timestamping which takes over a failed one, and which has not been contacted by it, periodically compares the value of its initialized counters with the timestamps which are stored in the DHT. If a counter is lower than the highest timestamp found, the responsible of timestamping corrects the counter. Furthermore, it reinserts the data which has been stored in the DHT with the latest value of the incorrect counter (if any).

## 4.3 Validity of Counters

In Section 4.1, the third rule for managing VCSs states that if a peer $p$ loses the responsibility for a key $k$ wrt $h_{ts}$, then $p$ should remove $c_{p,k}$ from $VCS_p$ (if it is there). We now discuss what $p$ should do in order to respect the third rule for $VCS_p$. If the reason for losing responsibility is that $p$ has left the P2P system or failed, then there is nothing to do, since when $p$ rejoins the P2P system, it sets $VCS_p=\varnothing$. Therefore, we assume that $p$ is present in the P2P system and loses the responsibility for $k$ wrt $h_{ts}$ because some other peer joins the P2P system and becomes responsible for $k$.

We can classify DHT protocols in two categories: *Responsibility Loss Aware (RLA)* and *Responsibility Loss Unaware (RLU)*. In an RLA DHT, a peer that loses responsibility for some key $k$ wrt $h$ and is still present in the P2P system detects its loss of responsibility. A DHT that is not RLA is RLU.

Most DHTs are RLA, because usually when a new peer $p$ becomes *rsp(k, h)*, it contacts *prsp(k,h)*, say $q$, and asks $q$ to return the pairs *(k, data)* which are stored at $q$. Thus, $q$ detects the loss of responsibility for $k$. Furthermore, in most of DHTs, $p$ is a new neighbor of $q$ (see Section 4.2.1), so when $p$ arrives $q$ detects that it has lost the responsibility for some keys. For the DHTs that are RLA, the third rule of VCS can be enforced as follows. When a peer $p$ detects that it has lost the responsibility for some keys wrt $h_{ts}$, it performs the following instructions:

```
For each $c_{p,k} \in VCS_p$ do
      If $p \neq$rsp(k,$h_{ts}$) then
        remove $c_{p,k}$ from $VCS_p$
```

If the DHT is RLU, then Rule 3 can be violated. Let us illustrate with the following scenario. Let $k$ be a key and $p$ the peer that is *rsp(k,$h_{ts}$)* which generates some timestamp for $k$, *i.e.* $c_{p,k}$ is in $VCS_p$. Suppose another peer $q$ joins the P2P system, becomes *rsp(k, $h_{ts}$)* and generates some timestamps for $k$. Then $q$ leaves the DHT, and $p$ becomes again *rsp(k,$h_{ts}$)*. In this case, if $p$ generates a timestamp for $k$ using $c_{p,k} \in VCS_p$, the generated timestamp may be equal or less than the last generated timestamp for $k$, thus violating the monotonicity property as a result of violating Rule 3. To avoid such problems in a DHT that is RLU, we impose that *rsp(k,$h_{ts}$)* assumes that after generating each timestamp for $k$, it loses its responsibility for $k$ wrt $h_{ts}$. Thus, after generating a timestamp for $k$, it removes $c_{p,k}$ from $VCS_p$. Therefore, Rule 3 is enforced. However, by this strategy, for generating each timestamp for $k$ we need to initialize $c_{p,k}$, and this increases the cost of timestamp generation.

## 5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our Update Management Service (UMS) through implementation and simulation. The implementation over a 64-node cluster was useful to validate our algorithm and calibrate our simulator. The simulation allows us to study scale up to high numbers of peers (up to 10,000 peers).

The rest of this section is organized as follows. In Section 5.1, we describe our experimental and simulation setup, and the algorithms used for comparison. In Section 5.2, we first report experimental results using the implementation of UMS and KTS on a 64-node cluster, and then we present simulation results on performance by increasing the number of peers up to *10,000*. In Sections 5.3, we evaluate the effect of the number of replicas, which we replicate for each data in the DHT, on performance. In Section 5.4, we study the effect of peers' failures on performance. In Section 5.5, we study the effect of the frequency of updates on performance.

### 5.1 Experimental and Simulation Setup

Our implementation is based on Chord [29] which is a simple and efficient DHT. Chord's lookup mechanism is provably robust in the face of frequent node fails, and it can answer queries even if the system is continuously changing. We implemented UMS and KTS as a service on top of Chord which we also implemented. In our implementation, the keys do not depend on the data values, so changing the value of a data does not change its key.

We tested our algorithms over a cluster of 64 nodes connected by a 1-Gbps network. Each node has 2 Intel Xeon 2.4 GHz processors, and runs the Linux operating system. We make each node act as a peer in the DHT.

To study the scalability of our algorithms far beyond 64 peers, we implemented a simulator using SimJava [27]. To simulate a peer, we use a SimJava entity that performs all tasks that must be done by a peer for executing the services KTS and UMS. We assign a delay to communication ports to simulate the delay for sending a message between two peers in a real P2P system. Overall, the simulation and experimental results were qualitatively similar. Thus, due to space limitations, for most of our tests, we only report simulation results.

The simulation parameters are shown in Table 1. We use parameter values which are typical of P2P systems [25]. The latency between any two peers is a normally distributed random number with a mean of 200 ms. The bandwidth between peers is also a random number with normal distribution with a mean of 56 (kbps). The simulator allows us to perform tests up to *10,000* peers, after which simulation data no longer fit in RAM and makes our tests difficult. Therefore, the number of peers is set to be *10,000*, unless otherwise specified.

In each experiment, peer departures are timed by a random Poisson process (as in [21]). The average rate, *i.e.* $\lambda$, for events of the Poisson process is $\lambda$=1/second. At each event, we select a peer to depart uniformly at random. Each time a peer goes away, another joins, thus keeping the total number of peers constant (as in [21]).

Peer departures are of two types: normal leave or fail. Let *failure rate* be a parameter that denotes the percentage of departures which are of fail type. When a departure event occurs, our simulator must decide on the type of this departure. For this, it generates a random number which is uniformly distributed in [0..100]; if the number is greater than *failure rate* then the peer departure is considered as a normal leave, else as a fail. In our tests, the default setting for *fail rate* is 5%.

In our experiments, each replicated data is updated by update operations which are timed by a random Poisson process. The default average rate for events of this Poisson process is $\lambda$=1/hour.

In our tests, unless otherwise specified, the number of replicas of each data is 10, *i.e.* $|H_r|=10$.

**Table 1. Simulation parameters**

| Simulation parameter | Values |
|---|---|
| Bandwidth | Normally distributed random number, Mean = 56 Kbps, Variance = 32 |
| Latency | Normally distributed random number, Mean = 200 ms, Variance = 100 |
| Number of peers | 10,000 peers |
| $|H_r|$ | 10 |
| Peers' joins and departures | Timed by a random Poisson process with $\lambda$=1/second |
| Updates on each data | Timed by a random Poisson process with $\lambda$=1/hour |
| Failure rate | 5% of departures |

Although it cannot provide the same functionality as UMS, the closest prior work to UMS is the BRICKS project [13]. To assess the performance of UMS, we compare our algorithm with the BRICKS algorithm, which we denote as BRK. We tested two versions of UMS. The first one, denoted by UMS-Direct, is a version of UMS in which the KTS service uses the direct algorithm for initializing the counters. The second version, denoted by UMS-Indirect, uses a KTS service that initializes the counters by the indirect algorithm.

In our tests, we compare the performance of UMS-Direct, UMS-Indirect and BRK in terms of response time and communication cost. By response time, we mean the time to return a current replica in response to a query *Q* requesting the data associated with a key. The communication cost is the total number of messages needed to return a current replica in response to *Q*. For each experiment, we perform 30 tests by issuing *Q* at 30 different times which are uniformly distributed over the total experimental time, *e.g.* 3 hours, and we report the average of their results.

### 5.2 Scale up

In this section, we investigate the scalability of UMS. We use both our implementation and our simulator to study the response time and communication cost of UMS while varying the number of peers.

Using our implementation over the cluster, we ran experiments to study how response time increases with the addition of peers. Figure 6 shows the response time with the addition of peers until 64. The response time of all three algorithms grows logarithmically with the number of peers. However, the response time of UMS-Direct and UMS-Indirect is significantly better than BRK. The reason is that, by using KTS and determining the last generated timestamp, UMS can distinguish the currency of replicas and return the first current replica which it finds while BRK needs to retrieve all available replicas, which hurts response time. The response time of UMS-Direct is better than UMS-Indirect because, for determining the last timestamp, UMS-Direct uses a version of KTS that initializes the counters by the direct algorithm which is more efficient than the indirect algorithm used by UMS-Indirect. Note that the reported results are the average of the results of several tests done at uniformly random times.

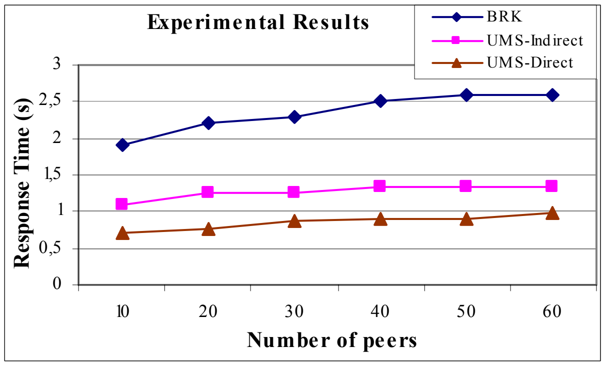

**Figure 6. Response time vs. number of peers**

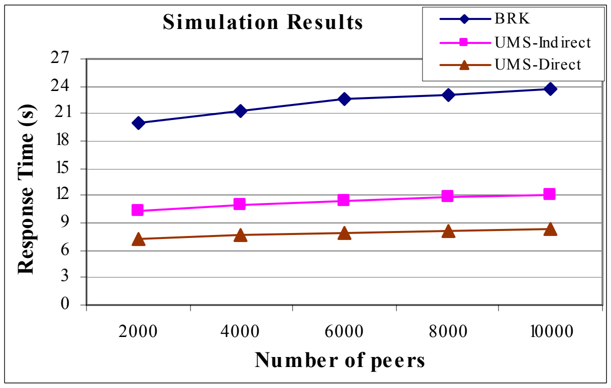

**Figure 7. Response time vs. number of peers**

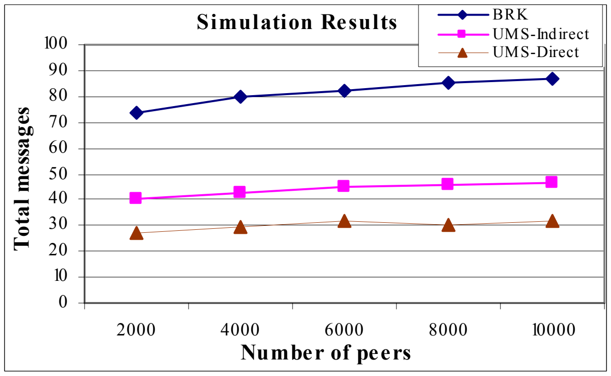

**Figure 8. Communication cost vs. number of peers**

Using simulation, Figure 7 shows the response time of the three algorithms with the number of peers increasing up to 10000 and the other simulation parameters set as in Table 1. Overall, the experimental results correspond qualitatively with the simulation results. However, we observed that the response time gained from our experiments over the cluster is slightly better than that of simulation for the same number of peers, simply because of faster communication in the cluster.

We also tested the communication cost of UMS. Using the simulator, Figure 8 depicts the total number of messages while increasing the number of peers up to 10,000 with the other simulation parameters set as in Table 1. The communication cost increases logarithmically with the number of peers.

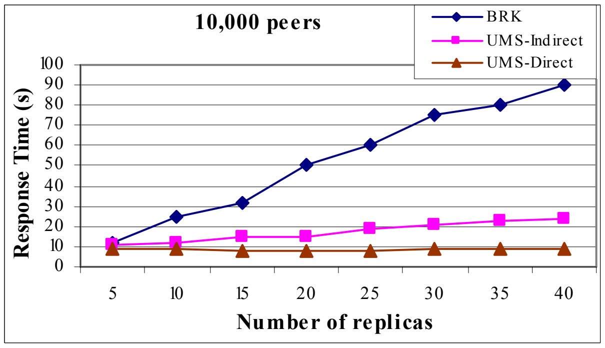

**Figure 9. Response time vs. number of replicas**

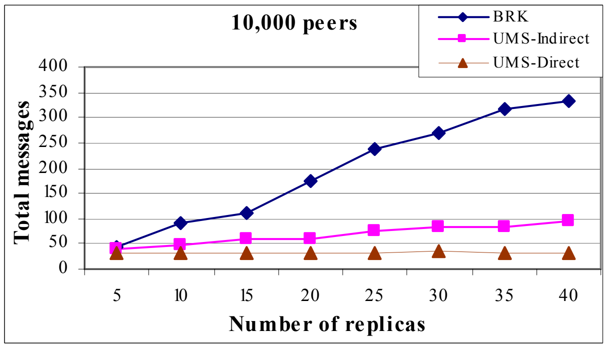

**Figure 10. Communication cost vs. number of replicas**

## 5.3 Effect of the Number of Replicas

In this section, we study the effect of the number of replicas, which we replicate for each data in the DHT, on the performance of MUS.

Using the simulator, Figures 9 and 10 show how respectively response time and communication cost evolve while increasing the number of replicas, with the other simulation parameters set as in Table 1. The number of replicas has a strong impact on the performance of BRK, but no impact on UMS-Direct. It has a little impact on the performance of UMS-Indirect because, in the cases where the counter of a key is not initialized, UMS-Indirect must retrieve all replicas from the DHT.

## 5.4 Effect of Failures

In this section, we investigate the effect of failures on the response time of UMS. In the previous tests, the value of failure rate was 5%. In this section, we vary the value of fail rate and investigate its effect on response time.

Figure 11 shows how response time evolves when increasing the fail rate, with the other parameters set as in Table 1. An increase in failure rate decreases the performance of Chord's lookup service, so the response time of all three algorithms increases. For the cases where the failure rate is high, *e.g.* more than 80%, the response time of UMS-Direct is almost the same as UMS-Indirect. The reason is that if a responsible of timestamping fails, both UMS-Direct and UMS-Indirect need to use the indirect algorithm for initializing the counters at the next responsible of timestamping, thus their response time is the same.

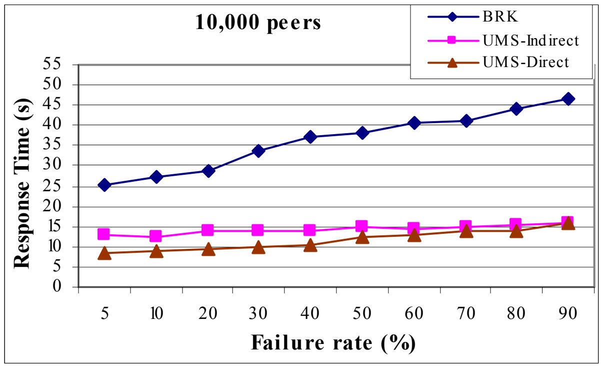

**Figure 11. Response time vs. failure rate**

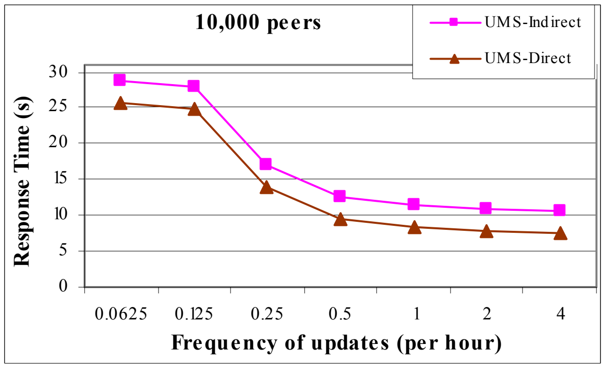

**Figure 12. Response time vs. frequency of updates**

## 5.5 Effect of Update Frequency

In this section, we study the effect of the frequency of updates on the performance of UMS. In the previous experiments, updates on each data were timed by a Poisson process with an average rate of 1/hour. In this section, we vary the average rate (*i.e.* frequency of updates) and investigate its effect on response time.

Using our simulator, Figures 12 shows how response time evolves while increasing the frequency of updates with the other simulation parameters set as in Table 1. The response time decreases by increasing the frequency of updates. The reason is that an increase in the frequency of updates decreases the distance between the time of the latest update and the retrieval time, and this increases the probability of currency and availability, so the number of replicas which UMS retrieves for finding a current replica decreases.

## 6. RELATED WORK

In the context of distributed systems, data replication has been widely studied to improve both performance and availability. Many solutions have been proposed in the context of distributed database systems for managing replica consistency [17], in particular, using eager or lazy (multi-master) replication techniques. However, these techniques either do not scale up to large numbers of peers or raise open problems, such as replica reconciliation, to deal with the open and dynamic nature of P2P systems.

Data currency in replicated databases has also been widely studied, *e.g.* [1], [10], [11], [14], [22] and [26]. However, the main objective is to trade currency and consistency for performance while controlling the level of currency or consistency desired by the user. Our objective in this paper is different, *i.e.* return the current (most recent) replica as a result of a get request.

Most existing P2P systems support data replication, but without consistency guarantees. For instance, Gnutella [9] and KaZaA [12], two of the most popular P2P file sharing systems allow files to be replicated. However, a file update is not propagated to the other replicas. As a result, multiple inconsistent replicas under the same identifier (filename) may co-exist and it depends on the peer that a user contacts whether a current replica is accessed.

PGrid is a structured P2P system that deals with the problem of updates based on a rumor-spreading algorithm [7]. It provides a fully decentralized update scheme, which offers probabilistic guaranties rather than ensuring strict consistency. However, replicas may get inconsistent, *e.g.* as a result of concurrent updates, and it is up to the users to cope with the problem.

The Freenet P2P system [3] uses a heuristic strategy to route updates to replicas, but does not guarantee data consistency. In Freenet, the query answers are replicated along the path between the peers owning the data and the query originator. In the case of an update (which can only be done by the data's owner), it is routed to the peers having a replica. However, there is no guarantee that all those peers receive the update, in particular those that are absent at update time.

Many of existing DHT applications such as CFS [4], Past [24] and OceanStore [20] exploit data replication for solving the problem of hot spots and also improving data availability. However, they generally avoid the consistency problem by restricting their focus on read-only (immutable) data.

The BRICKS project [13] deals somehow with data currency by considering the currency of replicas in the query results. For replicating a data, BRICKS stores the data in the DHT using multiple keys, which are correlated to the key *k* by which the user wants to store the data. There is a function that, given *k*, determines its correlated keys. To deal with the currency of replicas, BRICKS uses versioning. Each replica has a version number which is increased after each update. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica. In addition, to return a current replica, all replicas need be retrieved in order to select the latest version. In our solution, concurrent updates raise no problem, *i.e.* this is a consequence of the monotonicity property of timestamps which are generated by KTS. In addition, our solution does not need to retrieve all replicas, and thus is much more efficient.

## 7. CONCLUSION

To ensure high data availability, DHTs typically rely on data replication, yet without currency guarantees for updateable data. In this paper, we proposed a complete solution to the problem of data availability and currency in replicated DHTs. Our main contributions are the following.

First, we proposed a new service called Update Management Service (UMS) which provides efficient retrieval of current replicas. For update operations, the algorithms of UMS rely on timestamping. UMS supports concurrent updates. Furthermore, it has the ability to determine whether a replica is current or not without comparing it with other replicas. Thus, unlike the solution in [13], our solution does not need to retrieve all replicas for finding a current replica, and is much more efficient.

Second, we gave a probabilistic analysis of UMS's communication cost by computing the expected number of replicas which UMS must retrieve. We proved that this number is less than the inverse of the probability of currency and availability. Thus, except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of replicas are current and available then this number is less than 3.

Third, we proposed a Key-based Timestamping Service (KTS) which generates monotonically increasing timestamps in a completely distributed fashion, using local counters. The dynamic behavior of peers causes no problem for KTS. To preserve timestamp monotonicity, we proposed a direct and an indirect algorithm. The direct algorithm deals with the situations where peers leave the system normally, *i.e.* without failing. The indirect algorithm takes into account the situations where peers fail. Although the indirect algorithm has high probability of success in general, there are rare situations where it may not be successful at finding the current replica. We proposed two strategies to deal with these situations.

Fourth, we validated our solution through implementation and experimentation over a 64-node cluster and evaluated its scalability through simulation over 10,000 peers using SimJava. We compared the performance of UMS and BRK (from the BRICK project) which we used as baseline algorithm. The experimental and simulation results show that using KTS, UMS achieves major performance gains, in terms of response time and communication cost, compared with BRK. The response time and communication cost of UMS grow logarithmically with the number of peers of the DHT. Increasing the number of replicas, which we replicate for each data in the DHT, increases very slightly the response time and communication cost of our algorithm. In addition, even with a high number of peer fails, UMS still works well. In summary, this demonstrates that data currency, a very important requirement for many applications, can now be efficiently supported in replicated DHTs.

## REFERENCES

[1] Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R., and Tamma, P. Relaxed-currency serializability for middle-tier caching and replication. *Proc. of SIGMOD*, 2006.

[2] Bromwich, T.J.I. *An Introduction to the Theory of Infinite Series*. 3rd edition, Chelsea Pub. Co., 1991.

[3] Clarke, I., Miller, S.G., Hong, T.W., Sandberg, O., and Wiley, B. Protecting Free Expression Online with Freenet. *IEEE Internet Computing 6(1)*, 2002.

[4] Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., and Stoica, I. Wide-Area Cooperative Storage with CFS. *Proc. of ACM Symp. on Operating Systems Principles*, 2001.

[5] Dabek, F., Zhao, B.Y., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a Common API for Structured Peer-to-Peer Overlays. *Proc. of Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[6] Daswani, N., Garcia-Molina, H., and Yang, B. Open Problems in Data-Sharing Peer-to-Peer Systems. *Proc. of ICDT*, 2003.

[7] Datta A., Hauswirth M., and Aberer, K. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. *Proc. of IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2003.

[8] Elson, J., Girod, L., and Estrin, D. Fine-grained Network Time Synchronization Using Reference Broadcasts. *SIGOPS Operating Systems Review, 36(SI)*, 2002.

[9] Gnutella. http://www.gnutelliums.com/.

[10] Guo, H., Larson, P.Å., Ramakrishnan, R., and Goldstein, J. Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. *Proc of SIGMOD*, 2004.

[11] Guo, H., Larson, P.Å., and Ramakrishnan, R. Caching with 'Good Enough' Currency, Consistency, and Completeness. *Proc. of VLDB*, 2005.

[12] Kazaa. http://www.kazaa.com/.

[13] Knezevic, P., Wombacher, A., and Risse, T. Enabling High Data Availability in a DHT. *Proc. of Int. Workshop on Grid and P2P Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE)*, 2005.

[14] Lindstrom, G., and Hunt, F. Consistency and Currency in Functional Databases. *Proc. of INFOCOM*, 1983.

[15] Luby, M. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.

[16] Mills., D.L. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications, 39(10)*, 1991.

[17] Özsu, T., and Valduriez, P. *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.

[18] PalChaudhuri, S., Saha, A.K., and Johnson, D.B. Adaptive Clock Synchronization in Sensor Networks. *Proc. of Int. Symp. on Information Processing in Sensor Networks*, 2004.

[19] Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., and Shenker, S. A Scalable Content-Addressable Network. *Proc. of SIGCOMM*, 2001.

[20] Rhea, S.C., Eaton, P.R., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. Pond: the OceanStore Prototype. *Proc. of USENIX Conf. on File and Storage Technologies (FAST)*, 2003.

[21] Rhea, S.C., Geels, D., Roscoe, T., Kubiatowicz, J. Handling Churn in a DHT. *Proc. of USENIX Annual Technical Conf.*, 2004.

[22] Röhm, U., Böhm, K., Schek, H., and Schuldt,H. FAS: a Freshness- Sensitive Coordination Middleware for a Cluster of OLAP Components. *Proc. of VLDB*, 2002.

[23] Rowstron, A. I.T., and Druschel, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Proc. of Middleware*, 2001.

[24] Rowstron, A., and Druschel, P. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. *Proc. of ACM Symp. on Operating Systems Principles*, 2001.

[25] Saroiu, S., Gummadi, P.K., and Gribble, S.D. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc. of Multimedia Computing and Networking (MMCN)*, 2002.

[26] Segev, A., and Fang, W. Currency-based Updates To Distributed Materialized Views. *Proc. of ICDE*, 1990.

[27] Simjava. http://www.dcs.ed.ac.uk/home/hase/simjava/

[28] Spivak, M. *Calculus*. 3rd edition, Cambridge University Press, 1994.

[29] Stoica I., Morris, R., Karger, D.R., Kaashoek, M.F., and Balakrishnan, H. Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proc. of SIGCOMM*, 2001.