Enabling High Data Availability in a DHT*

Predrag Knežević[1], Andreas Wombacher[2], Thomas Risse[1]

[1]Fraunhofer IPSI
Integrated Publication and Information Systems Institute
Dolivostrasse 15, 64293 Darmstadt, Germany
{knezevic|risse}@ipsi.fraunhofer.de

[2]University of Twente
Department of Computer Science
Enschede, The Netherlands
a.wombacher@cs.utwente.nl

Abstract

Many decentralized and peer-to-peer applications require some sort of data management. Besides P2P file-sharing, there are already scenarios (e.g. BRICKS project [3]) that need management of finer-grained objects including updates and, keeping them highly available in very dynamic communities of peers. In order to achieve project goals and fulfill the requirements, a decentralized/P2P XML storage on top of a DHT (Distributed Hash Table) overlay has been proposed [6]. Unfortunately, DHTs do not provide any guarantees that data will be highly available all the time.

A self-managed approach is proposed where availability is stochastically guaranteed by using a replication protocol. The protocol recreates periodically missing replicas dependent on the availability of peers. We are able to minimize generated costs for requested data availability. The protocol is fully decentralized and adapts itself on changes in community maintaining the requested availability. Finally, the approach is evaluated and compared with replication mechanisms embedded in other decentralized storages.

1 Introduction

The research presented in this paper is motivated by the BRICKS[1] project, which aims to design, develop and maintain a user and service-oriented space of digital libraries that share knowledge and resources in the Cultural Heritage domain. The project defines a decentralized, service-oriented infrastructure that uses the Internet as a backbone and fulfills the requirements of expandability, scalability and interoperability. At the same time, the membership in the BRICKS community is very flexible; parties can join or leave the system at any time.

*This work is partly funded by the European Commission under BRICKS (IST 507457)

[1]BRICKS - Building Resources for Integrated Cultural Knowledge Services, `http://www.brickscommunity.org`

BRICKS community needs to have service descriptions, administrative information about collections, ontologies and some annotations globally available all the time [12]. An important aspect is that data are changeable during the run-time, i.e. updates must be allowed. Therefore, the data management is based on our recently proposed decentralized XML data store [6]. The store is based on top of a DHT (Distributed Hash Table) overlay, i.e. large XML documents are splitted into sets of XML nodes stored then as DHT values. DHTs are low-level structured P2P systems that provide a consistent way of routing information to the final destination, can handle the changes in topologies and have an API similar to the hash table data structure.

Unfortunately, a DHT layer does not guarantee the availability of data it manages. Whenever a peer goes offline, locally stored **(key, value)** pairs become inaccessible. Thus, it is needed to build a wrapper around the DHT layer, so the high data availability is provided.

Data availability can be reached by enabling redundancy in two ways: replicating data many times, or using erasure coding to code data and dividing them into many blocks. Although erasure coding provides in general lower storage costs [16], it has some drawbacks that are particularly important for our storage [13]. Namely, we manage smaller object, and need to enable decentralized query processing. Since the erasure coding on smaller object has higher cost than the replication and makes local data processing impossible, we choose replication as a way to achieve data availability in a DHT.

Usually, community of peers are highly dynamic, i.e. peers are frequently offline. According to some measurements [15], an average peer online probability in the Gnutella network is between 12 and 16%. Applying a simple replication protocol, where a number of replicas is created, and requested object availability is 95%, it turns out that 23 replicas are needed - a fairly high overhead. Therefore, we must come up with a protocol that generates lower costs and keeps high data availability.

Also, when a peer is offline, locally stored replicas are inaccessible. Therefore, an update might not address all repli-

Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)
1529-4188/05 $20.00 © 2005 **IEEE**

cas, leaving some of them unmodified. Further, uncoordinated concurrent updates of an object result in unpredictable values of object replicas. As a consequence, different object replicas may have different values. Thus, the main issues are (1) how to ensure that the correct value is read, (2) to synchronize offline replicas after going online again, and (3) to handle concurrent updates on the same data.

The presented research extends existing DHTs in a way that they are able to self-manage data availability with the requested probabilistic guarantees. The approach also implements a decentralized concurrency control that gives probabilistic guarantees that a correct object value will be read at any point in time.

The paper is organized in the following way. The next Section gives details about applied replication and how it adapts on changes in community. The approach is compared with some others in the field in Section 3. Related work to the idea presented in the paper is given in Section 4. Finally, Section 5 gives conclusions and some ideas for the future work.

2 Approach

As it is already mentioned, our decentralized XML storage uses a DHT layer for managing XML data. Unfortunately, all DHT implementations do not guarantee data availability, i.e. when a peer is offline, then locally stored data are offline too. Therefore, enabling high data availability in a DHT can be done by making a wrapper that implements a replication protocol around it and provides the same API to upper layers.

2.1 Distributed Hash Tables

As suggested in [5], a common DHT API should contain at least the following methods: **route(Key, Message)** (deterministic routing of a message according to the given key to the final destination), **store(Key, Value)** (store a value with the given key in DHT), and **lookup(Key)** (returns the value associated with the key).

Every peer is responsible for a portion of the key space, so whenever a peer issues a **store** or **lookup** request, it will end up on the peer responsible for that key. When the system topology is changed, i.e. peers go offline, some peers will be now responsible for the keyspace that has belonged to the offline peers. Also, peers joining the system will take responsibility for a part of the keyspace that has been under control of other peers until that moment. All **(key, value)** pairs stored on an offline peer are not available until the peer comes back again.

Reasons of using a DHT implementation as the ground layer of the decentralized XML storage are twofolds: (1) XML documents have a tree representation that can be easily mapped on a hash table (DHTs are essentially hash tables), and (2) all replication protocols assume implicitly or explicitly that there is a way to locate data, i.e. the existence of a directory.

Every value and its replicas are associated with a key that is used for **store** and **lookup** operation. The first replica key is generated using a random number generator. All other replica keys are correlated with the first one, i.e. they are derived from it by using the following rule:

$$replicaKey(i) = \begin{cases} c & : \quad i = 1 \\ hash(replicaKey(1) + i) & : \quad i \geq 2 \end{cases} \tag{1}$$

where c is a random byte array, $hash$ is a hash function with a low collision probability. $replicaKey$ and i are observed as byte arrays, and $+$ is an array concatenation function.

2.2 Operations

In order to add data availability feature to the existing DHT, every stored value must be replicated R number of times. Every peer calculates it from measured average peer online probability and the requested data availability. During joining phase, a peer can get an initial value for R from other peers in the system, or it can assume some default one.

High data availability in a DHT is achieved by self-adaptive replication protocol, i.e. missing replicas of locally stored values are recreated within refreshment rounds. The approach is proactive; a peer wants to secure that values from its storage will be available even if the peer goes offline at any point in time. Remembering the key generation schema in Formula 1, recreation of replicas would require access to the first replica key. Therefore, it must be attached to the stored value.

Another important aspect of the protocol are updates. As it has already been mentioned, ensuring consistency is the main issue. Basically, there are two possible groups of approaches [10]: pessimistic and optimistic.

Pessimistic approaches are based on locking and a centralized lock management. When a peer in decentralized/P2P environment goes offline, it and its data are not reachable. In addition, this may cause network partition, thus not even all online peers are reachable. All this unreachable peers cannot receive a lock, thus the pessimistic approach is not applicable.

In an optimistic approach, objects are not locked, but when a conflict occurs, the system tries to resolve it, or they are resolved manually. Optimistic approaches are simpler to implement and they are good if the probability for updating the same object with different values at the same time is low.

In order to determine the latest value version, we need to track it. To summarize, a DHT value will be wrapped in an instance of the following class:

```
class Entry {
  Key first;
```


Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)
1529-4188/05 $20.00 © 2005 **IEEE**

```
  long version;
  Object value;
}
```

Since the wrapper around DHT implements common DHT API introduced in Section 2.1, **store** and **lookup** operations must be re-implemented. Further, the mechanism for self-managing, i.e. refreshment rounds and rejoins of peers are introduced.

lookup(Key) When a peer wants to get a value, it is not sufficient to return any available replica. Instead of that, we must return the replica with the highest version number to ensure that the peer gets the most up-to-date available version. However, if two or more replicas with the same version (e.g. as a result of network partitioning, but with different values are found), it is a conflict that could be resolved by applying some heuristic if data semantic is known or it must be resolved manually. Currently, we do not assume any heuristic, i.e. a failure is returned, which has to be compensated by the requester.

store(Key, Value) When a value is created, it is wrapped in R instances of `Entry` class, appropriate keys are generated and version is assigned to 1. With every update, the version number is incremented by 1. During an update, replicas are modified in sequence, i.e. first the original, then 1^{st} replica, 2^{nd} replica until R^{th} replica. If the update of any replica fails, the update stops and the rest of the replicas are not touched. The update fails if a peer that receives the update request already has a replica with a higher version or the same version containing different value. The proposed write operation ensures that in case of concurrent updates only one peer completes the operation. The rest of them must compensate the request.

In order to know what should be the next version number, the replication layer must keep a log of (key, version) pairs of successful lookups. The log size and its organization are part of our future work.

During a **refreshment round**, a peer iterates over locally stored data, checks for missing replicas and recreates them. Every peer proceeds independently, there are no global synchronization points in time. Another important aspect of refreshment is that peers get more recent data versions from other peers and if there are no topology changes, the system will eventually stabilize. Also, at the beginning of a refreshment round, a peer can measure the average online probability of replicas, and compute the average data availability. If the obtained value is above a specified threshold, refreshment round can be made longer, so bandwidth utilization is saved, and/or number of replicas can decrease saving storage space . If the data availability is below the threshold, a peer should recreate replicas often, and/or create more replicas, trying to catch up requested data availability.

Measuring the average replica online probability could be done by checking all replicas in the system. Unfortunately, this is not feasible, because we simply do not know how many replicas are out there. Even if we knew that, measuring would be very inefficient and unscalable. Therefore, we use the confidence interval theory [1] to find out what is the minimal number of replicas that has to be checked, so the computed average replica online probability is accurate with some degree of confidence. For example, to achieve an accuracy with an error of 15% in a community of 1000 peers, we have to check only 12 randomly chosen replicas. It can also be shown that in large communities the approach is scalable.

In practice, a peer selects on random basis a few locally stored replicas, generates needed number of replica keys, check if they are available, and computes the average replica online availability.

When a peer **rejoins** the community, it does not change its ID, so peer will be now responsible for a part of the keyspace that intersects with previously managed. Therefore, the peer keeps previously stored data, but no explicit data synchronization with other peers is required. Upcoming requests are answered using the latest locally available versions. With a new refreshment round or update, old replicas will be eventually overwritten. Replicas, whose keys are not anymore in the part of keyspace managed at rejoined peer, can be removed or sent to peers that should manage them.

2.3 Self-adaptation

Self-adaptation of the protocol is an optimization problem based on its costs, i.e. keeping the requested data availability with minimal cost in any point of time. Cost of a replication protocol can be described by a cost function:

$$cost(S) = w \cdot comm(S) + (1 - w)storage(S) \qquad (2)$$

where S is a vector of system parameters, such as the desired object availability, the peer online probability, and/or the average object access rate. With w ($w \in [0, 1]$) the importance of a particular term can be favored (e.g. when $w = 0.9$, the communication costs are much more important than the storage costs). We take as the communication costs only messages needed for replica maintenance, and issued from a peer that triggers the maintenance.

Before doing the analysis, we define a peer view on the environment in which different replica control protocols will be observed: (1) peers are independent (2) measured peer average online probability is p (3) because of the DHT properties, at any point in time, every object can have at most R accessible replicas, (4) availability a is defined as the probability that at least one of the replicas is present in the system at the moment when the value is going to be accessed.

We start the analysis without taking into consideration refreshment rounds. Let us denote with p_r the probability of a replica being online. Since peer IDs are generated

Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)
1529-4188/05 $20.00 © 2005 **IEEE**

randomly and keys according to Formula 1, we are ensuring that replica keys are in different parts of the key space, i.e. chances that they will be stored at different peers in a large community are very high. Thus, we can assume that replica online probability is equal to peer online probability i.e. $p_r = p$.

Let us denote with Y a random variable that represents the number of replicas being online, and $P(Y \geq y)$ is the probability that at least y replicas are online. Since peers are independent, the probability follows Binomial distribution. The probability a that a DHT value is available is equal to the probability that at least one replica is online ($P(Y \geq 1)$):

$$a = P(Y \geq 1) = 1 - (1 - p_r)^R \tag{3}$$

and the average number of online replicas is expectation of the random variable Y:

$$\mathsf{E}(Y) = Rp \tag{4}$$

Let us now denote with q_i the probability that the value V_i is refreshed, and q is the average of all q_i. The probability q that an object will be refreshed is defined as

$$q = \frac{T_b}{T_{refresh}} = \frac{1}{k} \tag{5}$$

where $T_{refresh}$ is the period of refreshment round and T_b is a basic system time unit, and $T_{refresh} = kT_b, k \in \mathbb{N}$.

Having this in mind, the average number of replicas can be calculated as the sum of two terms:

$$\begin{aligned} R_{avg} &= qR\sum_{y=1}^{R} P(Y = y) + (1 - q)\mathsf{E}(Y) \\ &= R\underbrace{((1 - (1 - p)^R)q + (1 - q)p)}_{p_r} \end{aligned} \tag{6}$$

The first term represents the case when the value is refreshed; when at least one copy is online ($P(Y \geq 1)$), R replicas will be recreated. In case of none of peers is online ($P(Y = 0)$) the available number of replicas is zero, thus, does not influence the summation. Otherwise, the average number of replicas remains the same as in Formula 4.

The replica availability now depends in addition on the average refreshment probability q, and the number of replicas R.

We can now define the cost function (Formula 2) as

$$cost(p, q) = wqR + (1 - w)R \tag{7}$$

Finding an optimal solution requires looking for a minimum of the cost function under defined constraints. During the system life-time, we would like to keep the data availability above some given level. Therefore, the inequation $a \geq A$ (Formula 3), where p_r expressed as in Formula 6 is the constraint for the cost function.

The constraint is not a convex function and we cannot apply the Kuhn-Tucker theorem [11]. Thus, the solutions for q and R cannot be expressed in a symbolic form, but the numerical solutions are easy to find.

When p is being increased, R is going down to zero. When p is high, the cost functions becomes linear, i.e. it depends only on the number of replicas R. This is quite expected; in the communities with high online probability, refreshments need to happen rarely. Also, when an update happens, the refreshment rate could be reduced. Detailed analysis of updates is left as a future work.

In communities with low online probabilities, occupied space is drastically reduced (up to 90%). If minimizing traffic cost is dominant, then the space is reduced only when peer online probability is low.

3 Evaluation

Since there is a level of freedom (i.e. refreshment round period), the proposed replication protocol can adapt itself on the community properties. Of course, it has limitations; it is bounded by assumed minimal p, available bandwidth, and the size of stored data. However, by choosing carefully the protocol parameters, the protocol can deliver guaranteed object availability in very dynamic communities and at the same time minimize the system cost.

PAST [14] and CFS [4] are P2P systems that use a simple replication protocol: files/blocks are replicated R times within final destination's leafset/successor list. When the peer detects that a neighbor is offline, it recreates all needed replicas on a new peer. As long as the peer that manages some replicas is online (p), it can keep the number of replicas on high level (R). When it goes offline ($1 - p$), then the average number of replicas follows Formula 4. If we adopt the same terminology as for the analysis presented in the paper, and applying the similar reasoning as in Formula 6 (i.e. computing $R_{avg_{PAST}}$), then the probability of a replica being online in PAST or CFS system would be:

$$\begin{aligned} R_{avg_{PAST}} &= Rp_{r_{PAST}} \\ &\Rightarrow \quad p_{r_{PAST}} = 2pR - p^2R \end{aligned} \tag{8}$$

Comparing $p_{r_{PAST}}$ with p_r, it can be seen that our protocol provides higher data availability in low peer online communities with the same number of replicas. With increasing the number of replicas, the difference is even more obvious. We will further evaluate the preliminary results by using simulations.

Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)
1529-4188/05 $20.00 © 2005 **IEEE**

4 Related Work

Current popular P2P file-sharing systems (e.g. KaZaA, eDonkey, or Gnutella [8]) do not have any built-in support for replication. A file will be replicated every time when it is downloaded on a new peer. Therefore, the file availability depends on its popularity and this works fine for media file exchange. However, unlike in our approach, none of the peers knows how many times a particular file is replicated, so file removal or update is impossible.

P2P file-storing system like CFS [4] or PAST [14] have recognized the need for replication. CFS splits every file in a number of blocks that are then replicated a fixed number of times. PAST applies the same protocol, without chunking files into blocks. These approaches are evaluated against ours in Section 3.

Oceanstore [7] is a P2P file-storing system trying to provide a wide-area storage system. Oceanstore is not fully decentralized, it makes a distinction between clients and powerful, highly available servers (i.e. super-peer network). Within the storage, the data are replicated by using erasure coding (Reed-Solomon codes). Erasure coding is also used in TotalRecall [2]. Erasure coding offers lower storage costs compared to replication, if managed data are large in size. However, our DHT layer handles smaller data items. Additionally, if locally stored data are encoded, then it is not possible to process them without retrieving all needed pieces from the network, which would decrease the system performances.

Peer-to-peer filesystem Ivy [9] provides both read and write operations, and should manage data availability, but more details about applied replication mechanism are not given, nor simulation were performed with replication.

5 Conclusion and Future Work

The paper presents an approach of adding high data availability feature to any DHT overlay network by implementing a wrapper around it. The wrapper is self-adaptive to changes in community, decentralized and manages data availability with the requested probabilistic guarantees. The cost of the approach have been analyzed, it has been shown how to find optimal values of parameters that minimize traffic and storage costs.

An implementation of the protocol will be integrated in our decentralized XML datastore. Future research will be focused on updates and consistency in decentralized/P2P systems under the influence of the proposed protocol, and on system performance evaluation.

References

[1] D. A. Berry and B. W. Lindgren. *Statistics: Theory and Methods*. Duxbury Press, October 1995.

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *First ACM/Usenix Symposium on Networked Systems Design and Implementation*, pages 337–350, March 29-31 2004.

[3] BRICKS Consortium. *BRICKS - Building Resources for Integrated Cultural Knowledge Services (IST 507457)*, 2004. http://www.brickscommunity.org/.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 202–215. ACM Press, 2001.

[5] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[6] P. Knežević. Towards a reliable peer-to-peer xml database. In W. Lindner and A. Perego, editors, *Proceedings ICDE/EDBT Joint PhD Workshop 2004*, pages 41–50, P.O. Box 1527, 71110 Heraklion, Crete, Greece, March 2004. Crete University Press.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[8] D. Milojičić, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, HP, 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf.

[9] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[10] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[11] A. L. Peressini, F. Sullivan, and J. Uhl. *The Mathematics of Nonlinear Programming*. Springer Verlag, 1993.

[12] T. Risse and P. Knežević. A self-organizing data store for large scale distributed infrastructures. In *International Workshop on Self-Managing Database Systems(SMDB)*, April 8-9 2005.

[13] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *IPTPS '05: International Workshop on Peer-to-Peer Systems*, February 24-25 2005.

[14] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 188–201. ACM Press, 2001.

[15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[16] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338. Springer-Verlag, 2002.

Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)
1529-4188/05 $20.00 © 2005 **IEEE**