Highly Available DHTs: Keeping Data Consistency After Updates*

Predrag Knežević[1], Andreas Wombacher[2], and 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. The research in the paper is motivated by building a decentralized/P2P XML storage on top of a DHT (Distributed Hash Table). The storage must provide high data availability and support updates. High data availability in a DHT can be guaranteed by data replication. However, DHTs can not provide a centralized coordination guaranteeing data consistency upon updates. In particular, replicas may have different values due to concurrent updates or partitioning of the P2P network. An approach based on versioning of replica values is presented proposing a decentralized concurrency control system, where probabilistic guarantees can be provided for retrieving a correct replica value. This paper presents the protocol as well as a statistical analysis of the lower bound of the probabilistic guarantees.
Keywords: Peer-to-Peer Computing, Decentralized Data Management, DHT

1 Introduction

The research presented in this paper is motivated by the BRICKS[3] 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)

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

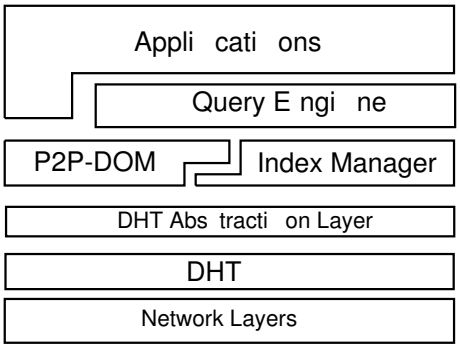

Fig. 1. Decentralized XML Storage Architecture

BRICKS community needs to have service descriptions, administrative information about collections, ontologies and some annotations globally available all the time [1]. An important aspect is that data are changeable during the runtime, i.e. updates must be allowed. Therefore, the data management is based on our recently proposed decentralized XML data store [2]. The store is based on top of a DHT (Distributed Hash Table) overlay, i.e. large XML documents are split 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.

Figure 1 presents the proposed system architecture. All layers exist on every peer in the system. The datastore is accessed through the P2P-DOM component or by using the query engine (e.g. XPath or XQuery). The query engine could be supported by an optional index manager that would maintain indices. The P2P-DOM exports a large portion of the DOM [3] interface to the upper layers, and maintains parts of a XML tree in a DHT. More details about the storage and selected research issues can be found in [2].

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. The research done in [4] proposed a wrapper around the DHT that is able to self-manage data availability by using replication within the requested probabilistic guarantees.

The research presented here investigates in detail data consistency, i.e. ensuring it after an update is preformed. When a peer is offline, locally stored replicas are inaccessible. Therefore, an update might not address all replicas, 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 how to:

– Ensure that the correct value is read

– Synchronize offline replicas after going online again
– Handle concurrent updates on the same data

The approach presented in the paper gives probabilistic guarantees on accessing correct data at any point in time. Also, replicas are updated in a predefined sequence, and they are assigned a higher version number.

The paper is organized in the following way. The next Section introduces the DHT with high data availability and update features. Consistency issues are analyzed afterwards in Section 3. Some related work is presented in Section 4. Finally, Section 5 gives conclusions and some ideas for the future work.

2 Highly Available Distributed Hash Table

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 key space that has belonged to the offline peers. Also, peers joining the system will take responsibility for a part of the key space 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.

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. Thus, the key of the original is determiend to be a random value c, while the key of the i^{th} replica is calculated by combining the randoam value c with the order number of the replica i and using the resulting value as a basis for a hash function delivering the key. The above rule enables uniqueness of all replica keys in system, and since distance between keys is high, it increases a probability that keys are placed on different peers in system. At the same time by knowing the first replica key, all other replica keys can be generated with no communication costs.

2.1 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 which 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 [6]:

- **Pessimistic**
  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 locking-based approach is not applicable.
  Quorum-based replica protocols require presence of quorums both for a read or write operation. In environments with low online probability, such quorums are hard to get and this makes the quorum-based protocols not a very good candidate [7].
- **Optimistic**
  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;
  long version;
  Object value;
}
```

Since the wrapper around DHT implements common DHT API introduced in Section 2, **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 heuristics, data semantics, or has to be resolved manually. Currently, we do not assume any heuristics, i.e. a failure is returned, which has to be compensated by the requester.

It is important to notice that during a **lookup** operation, all online replicas must be checked in order to find the latest available. Doing this by broadcasting the request would be fully inefficient: the whole network would be flooded. Using DHT overlay makes communication more efficient: the request is routed only to peers that could potentially have a replica. Although, obviously the required communication for deriving the value is higher then in DHTs without high data availability.

store(Key, Value) When a value is created, it is wrapped in R instances of `Entry` class, appropriate keys are generated and the version number 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 1^{st} replica, then 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 rounds 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 [8] 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, checks if they are available, and computes the average replica online availability.

When a peer **rejoins** the community, it does not change its ID, so the peer will be now responsible for a part of the key space that intersects with the 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 key space managed at the rejoined peer, can be removed or sent to peers that should manage them.

3 Data Consistency

The **store** operation from the previous section has been defined so that it is able to update data, even if not all replicas are online. Namely, a **store** will update all online replicas and recreate a higher version of replicas that are at that moment offline. From then, some replicas will be represented in the system with multiple versions, some of them will be online, some of them offline; some replicas will be up-to-date (i.e. correct), some odd (incorrect). This Section analyzes the probability that the correct data are found with every **lookup** operation.

Obviously, a correct data version will be read by a peer only if at least one correct replica is available. In order to compute the probability of this case, we need to model the life cycle of a replica, after it is initially created: the replica can be online or offline, and correct (up-to-date) or wrong (containing an older version). Therefore, during its life cycle, the replica can be in the following states: online and correct, online and wrong, offline and correct, and offline and wrong.

3.1 Settings

Before doing the analysis, we define the environment in which the proposed update protocol will be analyzed:

- Peers are independent
- Measured peer average online probability is p
- Because of the DHT properties, at any point in time, every object can have at most R accessible replicas.
- Success of an update is represented by a random variable U, i.e. successful update ($U = 1$) and unsuccessful update ($U = 0$). An unsuccessful update could change some replicas, but not all of them. At least one replica is correct.
- Reading of a correct object version is represented by a random variable C, i.e. correct read ($C = 1$) and incorrect read ($C = 0$)

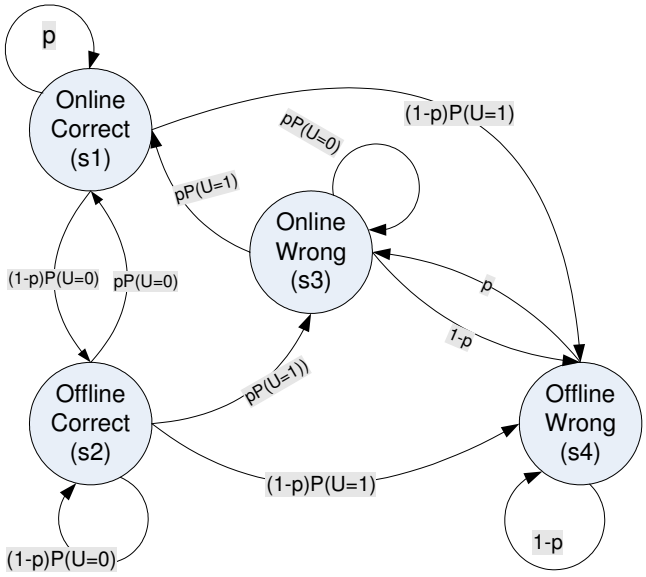

Fig. 2. The life cycle of replica after creation

- Random variable S represents the replica state, i.e. online-correct ($S = S1$), offline-correct ($S = S2$), online-wrong ($S = S3$), and offline-wrong ($S = S4$). Therefore, $p = P(S = S1) + P(S = S3)$.
- No partitions are assumed
- There is no need for recovery, i.e. going offline does not destroy locally stored data

The correct object version is going to be read, if at least one correct replica is online. In other words, it is the counter probability that none of online-correct replicas ($P(S = S1)$) is available. Therefore, the probability of reading the correct object ($P(C = 1)$) can be expressed as

$$P(C = 1) \geq 1 - (1 - P(S = S1))^R \tag{2}$$

After an update R replicas are online, but there might be some other offline replicas in the system. Therefore, Formula 2 is the lower-bound of the correct read probability.

3.2 Life Cycle of Replica

In order to describe the life cycle of a single replica we define a discrete-time Markov chain, represented by the state diagram on Figure 2. Transition probabilities are displayed next to each arc.

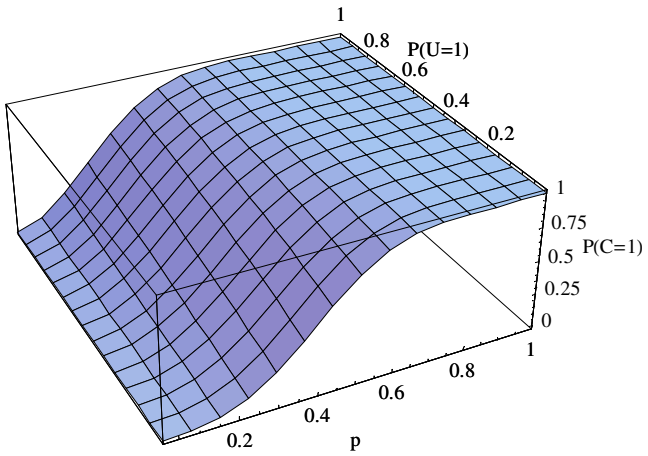

Fig. 3. The probability that correct value is read after update

An online-correct replica stays in this state as long as it is online. Also, a replica can come to this state, if it has been offline-correct and comes online again while no update has happened in the meantime. An online-wrong replica becomes correct after a successful update is performed.

A replica stays offline-correct if there are no successful updates. If they are successful, the replica is not up-to-date anymore and the state is changed to offline-wrong. Also, a replica comes into offline-correct state if it has been online-correct before and no updates happen when it goes offline.

A replica remains in offline-wrong state until it goes online. Then, when it is back online again, it goes to online-wrong state.

The probability of every described state ($P(S = S1, S2, S3, S4)$) can be calculated by applying long-run analysis of discrete-time Markov chains (i.e. equilibrium analysis) [9]. To compute the probability of correct read $P(C = 1)$, we need to determine the probability for being in state S1 ($P(S = S1)$, see Formula 2). Figure 3 shows the probability for reading an up-to-date object for number of replicas $R = 10$. This number has been taken from a previous analysis of the replication protocol [4], where it has been shown that it guarantees an average object availability $a \geq 99,9\%$, if peer online probability p is higher than 50%. It can be seen that the lower bound of the correct reading probability depends only weakly on the successful update probability $P(U = 1)$, because with or without successful update, the system will contain at least one correct replica (but maybe offline). For example, for a peer online probability of 50%, the correct reading probability with 100% successful updates ($P(U = 1) = 1$) is only 25% higher compared with a system with a probability of zero for successful updates ($P(U = 1) = 0$).

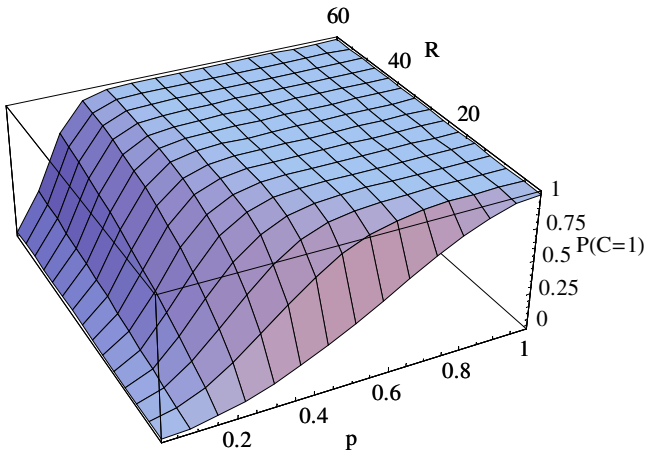

Fig. 4. The probability that correct value is read after update in function of the number of replicas

Figure 4 shows how the read probability changes with the increase of the number of replicas ($P(U = 1) = 0.9$). It can be seen that when the number of replicas is greater than 30 ($R \geq 30$), good reading probability ($P(C = 1) \geq 0.9$) can be achieved, even for low-online probabilities of peers ($p \geq 0.28$).

Allowing updates has introduced a possibility that in the system exist many different versions of the same object. As a consequence, our correct reading probability ($P(C = 1)$) is equal to data availability a. In the rest of the cases ($1 - P(C = 1)$), some replicas could be available, but with outdated values. Thus, applications built on top should handle scenarios when returned data are obsolete. However, incorrect replicas will not stay in the system forever; with every new update, or during refreshment round, they will be eventually overwritten with the correct version.

The presented analysis has shown how data availability depends on the given system parameters, i.e. the number of replicas R and the probability that an update was successful $P(U = 1)$. Future research will investigate how $P(U = 1)$ behaves in different application scenarios, and how it depends on the system parameters as well.

4 Related Work

Updates in replicated distributed databases are a widely researched field. As it has already mentioned in Section 2, both optimistic and pessimistic approaches for resolving updates exist [6]. However, they all assume high peer online probability and global system view, e.g. [10] proposes hierarchy-less data distribution,

but the approach requires high peer online probability. Our approach is fully decentralized and works under any peer online probability.

The popular P2P filesharing systems (e.g. KaZaA, Gnutella, eDonkey) [11] do not consider updates at all. If a file update occurs, it is not propagated to other replicas. There is no way that a peer that wants to get a file can conclude what is the freshest version.

Oceanstore [12] supports updates and does versioning of objects. An update request is sent first to the object's inner ring (primary replicas), which performs a Byzantine agreement protocol to achieve fault-tolerance and consistency. When the inner ring commits the update, it multicasts the result of the update down to the dissemination tree. To our knowledge, analysis of consistency guarantees has not been published so far. Also, the inner ring consists of super peers that are highly available.

The paper [13] addresses updates in P2P system, but the aim of the research is to reduce communication costs, data consistency has not been addressed.

Ivy [14] is a peer-to-peer file system that enables writes by maintaining log of changes at every peer with write access. Reading up-to-date file version requires consulting all longs, and that is not very efficient. Additional tool has been provided that can be run manually in order to resolve conflict that could occur during concurrent updates. Our approach does not need to contact all peers in order to find the freshest replica version.

TotalRecall [15] has a peer-to-peer storage system with update support. Files are immutable, so every new version is stored separately in the system, and some garbage collection is needed for removing old version. The system distinguishes master and slave replica copies, and therefore an update is first performed on a master responsible for an object. Then, the master updates all other slaves. If some slaves are offline, new slave peers will be selected and the update will be repeated. Our approach is simpler, we do not distinguish master and slaves, so there is no need to elect new master when the old one goes offline. Even during an update, peers could go offline, and if there is no conflict, the update is successful.

Om [16] is a peer-to-peer file system that achieves data high availability through online automatic regeneration while still preserving consistency guarantees. File access is done by using read-one/write-all quorum, i.e. implicitly high peer online probability is assumed. All writes are first performed at primary replicas that update later secondary replicas.

5 Conclusion and Future Work

The work presented in the paper is adding high data availability feature to any DHT overlay network under consideration of data consistency issues. In particular, versioning and replication of data stored in a DHT are introduced and a preliminary analysis has derived lower-boundaries for the probabilistic data availability guarantees providing consistency in the system. A good probability can be achieved even with moderate costs, i.e. number of replicas ($R = 10$),

and with moderate peer online probability ($p > 0.5$), whereas more replicas are needed for systems where peer online probability is low.

The future work will investigate in more details some of the parameter introduced in the model (e.g. update successfulness); their dependency on application patterns and other system parameters. Also, the approach will be investigated in situations when network partitions are allowed. Finally, the approach will be implemented and tested in practice.

References

1. Risse, T., Knežević, P.: A self-organizing data store for large scale distributed infrastructures. In: International Workshop on Self-Managing Database Systems(SMDB). (2005)
2. Knežević, P.: Towards a reliable peer-to-peer xml database. In Lindner, W., Perego, A., eds.: Proceedings ICDE/EDBT Joint PhD Workshop 2004, P.O. Box 1527, 71110 Heraklion, Crete, Greece, Crete University Press (2004) 41–50
3. : Document Object Model. (2002) http://www.w3.org/DOM/.
4. Knežević, P., Wombacher, A., Risse, T., Fankhauser, P.: Enabling high data availability in a dht. In: Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05) (submitted). (2005)
5. Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common api for structured peer-to-peer overlays. In: 2nd International Workshop on Peer-to-Peer Systems. (2003)
6. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice Hall (1999)
7. Jiménez-Peris, R., Patiño-Marténez, M., Alonso, G., Kemme, B.: Are quorums an alternative for data replication? ACM Trans. Database Syst. **28** (2003) 257–294
8. Berry, D.A., Lindgren, B.W.: Statistics: Theory and Methods. Duxbury Press (1995)
9. Tijms, H.C.: Stochastic Models: An Algorithmic Approach. John Wiley (1994)
10. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In: The VLDB Journal. (2000) 134–143
11. Milojičić, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z.: Peer-to-peer computing. Technical report, HP (2002) http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf.
12. Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., Kubiatowicz, J.: Maintenance-free global data storage. IEEE Internet Computing **5** (2001) 40–49
13. Datta, A., Hauswirth, M., Aberer, K.: Updates in highly unreliable, replicated peer-to-peer systems. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society (2003) 76
14. Muthitacharoen, A., Morris, R., Gil, T., Chen, B.: 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 (2002)
15. Bhagwan, R., Tati, K., Cheng, Y.C., Savage, S., Voelker, G.M.: Total recall: System support for automated availability management. In: First ACM/Usenix Symposium on Networked Systems Design and Implementation. (2004) 337–350
16. Yu, H., Vahdat, A.: Consistent and automatic replica regeneration. ACM Transactions on Storage **1** (2005) 3–37