XOROS: A mutable Distributed Hash Table

Antony Chazapis and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{chazapis, nkoziris}@cslab.ece.ntua.gr

Abstract. Data redundancy in DHTs is commonly accomplished through automatic replication of values to a set of close participating nodes. Such copies, once written, should not be modified, as there is no inherent DHT function that can operate on a dynamic set of mutable replicas. In this paper, we present XOROS — a system based on the Kademlia routing scheme, that addresses the problem by implementing a Byzantine-tolerant protocol for serializable data updates directly at the peer-to-peer level. Based upon related works that study distributed replica synchronization, mutual exclusion and communication in the presence of Byzantine behavior, we propose a unified DHT-based algorithm, that ties corresponding practices together in order to consistently propagate changes to all primary replicas of any key-value pair stored in the network. A multitude of applications may benefit from the resulting distributed read/write storage substrate as it retains all the advanced features of DHTs and is backwards compatible with existing put/get semantics.

1 Introduction

Distributed Hash Tables represent a major class of peer-to-peer designs that abstract the aggregate overlay as a key-value based storage facility. *Putting* and *getting* named data items requires only a small number of steps, proportional to the logarithm of the total network size. To achieve this, all DHT implementations assume a predefined structure for a vast, virtual identifier space, where both peers and data are uniformly *hashed* within. Each physical node stores values and manages lookup queries that refer to data with IDs "close" to its own. The notion of closeness depends on the details of each specific DHT implementation, as does the arrangement of the key space [1].

DHTs have allowed a large number of distributed applications to be layered on top, instantly inheriting their ability to scale to millions of nodes while remaining resilient to unadvertised subsystem or network failures. For instance, applications like distributed filesystems [2] and metadata management services [3] directly use a structured peer-to-peer deployment as a distributed storage substrate. Nevertheless, DHTs provide a write-once/read-only data repository. Any changes to a value result in a new key — a new object that is unique and will be placed at a different address. Furthermore, to guarantee availability, values

are copied to a range of nodes, usually via internal automatic replication mechanisms [4]. If an external system tries to update a value in-place (using the same key), the results will be unpredictable. An inherent DHT *update* function should properly resolve all consistency issues by uniformly modifying all instances of a value in a single operation, even if the particular nodes responsible for storing the copies continuously change due to network churn.

The absence of an *update* operation has led research initiatives to propose designs that work around the constrain: Ivy [5], a peer-to-peer filesystem, uses a DHT to store read-only state changes of namespace and file contents (logs), while Eliot [6], relies on centralized servers to handle metadata changes and index read-only file blocks. Updates are of significant importance to applications that can encode internal semantics to specific keys. Consider the aforementioned examples that could use DHT IDs to represent native objects (i.e. i-nodes or filename hashes). Similarly, mutable values would enable the implementation of common data structures over DHTs.

In this paper, we present a novel protocol that allows in-place, automatic, serializable data updates in the *primary-replica* set of any structured peer-to-peer system. We define *primary replicas* as the values stored at the closest nodes of a data item's key — the nodes that will be contacted by a *get* operation assuming that it does not use cached data. We argue that a consistent update operation requires implementing a secure and fault-tolerant mutual exclusion protocol directly at the DHT layer, accompanied by a value propagation process that will synchronize all primary replicas in a single, unified transaction. Moreover, we consider the DHT as an environment where any peer may exhibit arbitrary behavior, either on purpose or due to network instability.

Distributed mutual exclusion and conflicting decision resolution in the context of dynamic and Byzantine environments has been an active research target for many years. In this paper, we elaborate on the specific steps required to incorporate respective knowledge into a DHT implementation that inherently and transparently handles all corresponding issues. Peers of the resulting overlay, namely XOR Object Store (XOROS), use a Kademlia [7] routing scheme, along with a modified protocol to *get* and *put* data, although the additions and changes presented here could have been equally applied to any DHT design.

2 Related Work

The steps involved in constantly tracking the locations of replicas and synchronizing relevant data changes have been addressed both in the context of unstructured [8] and structured [9] peer-to-peer networks. The basic idea behind all proposed algorithms is to build a distributed replica location index that is dynamically updated, either as lookup commands follow the path from a node to a replica maintainer, or by a subscription protocol. When an update is issued, the new value is pushed to all members of the index. Some systems also require a periodic pull phase. Such practices require cooperation between nodes, that can either be enforced or propelled by establishing corresponding incentives.

Furthermore, it is assumed that there is one *primary node* that holds the latest value of an item and a *replica set* that needs to be updated.

In contrast, we consider an environment where multiple *primary replicas* of a key-value pair must be updated in a consistent and secure manner. Primary replicas are always available and their position is well-known (the dynamic set of closest nodes to an ID), so there is no need to maintain special indices in order to locate them. The algorithms proposed for maintaining loose replica consistency can be applied along our protocol to support *secondary replica* synchronization — i.e. to track down cached key-value pairs that may be distributed anywhere in the overlay. The main objective of keeping a replica index is to track *availability* of data. Thus, we argue that corresponding protocols not only do not overlap, but may also benefit from our work.

A distributed system known to be coordinating changes between primary and secondary replica sets is Oceanstore [10]. Oceanstore uses a top-level, well-connected server group that decides on the update commit order, before sending the new values to the rest of the nodes. Our algorithm aims at providing similar functionality, though inherently in a DHT, where all participating nodes have equal privileges and responsibilities. The need to provide atomic data operations across all replicated data items at the DHT level has been pointed out by Lynch *et. al* [11]. However, their proposed algorithms require a reliable message passing substrate and a network of trusted nodes that thoroughly comply to the respective protocols for joining and leaving the network.

Much related work can also be found in the context of distributed mutual exclusion algorithms [12], although to our knowledge, only the Sigma protocol [13] presents a method for implementing permission-based mutual exclusion directly at the peer level of DHTs. According to Sigma, in order to lock a given resource, a client must acquire the majority of its logical "replicas". Each such replica, is represented by a unique identifier which maps to a physical node responsible for granting one permission for the corresponding resource. The protocol addresses node failures and high lock contention, but requires trusting the involved parties. We argue that there is no need to have separate commands for mutual exclusion and data updates, especially when the semantics of obtaining the lock are well known beforehand. Moreover, there is a need to protect the mutual exclusion from peers that could exploit the update mechanism by not performing the necessary steps orderly. Our approach to handle these cases has been influenced by related studies on the diffusion of messages in Byzantine environments [14, 15].

Similar goals are targeted by Rosebud [16], a DHT-based storage system that employes a Byzantine-aware protocol for changing values of "public-key" data objects. To perform an update, the issuer first contacts replica holders to obtain a new version identifier and then issues a signed write request. In contrast to XOROS, these two phases are disjoint. By using a lock protocol instead of a version vector, XOROS supports a richer set of data manipulation operations. Rosebud also tracks DHT membership through a distributed "configuration service" and makes heavy use of certificates, practices which could contradict the requirements of some extremely dynamic or anonymity-preserving deployments.

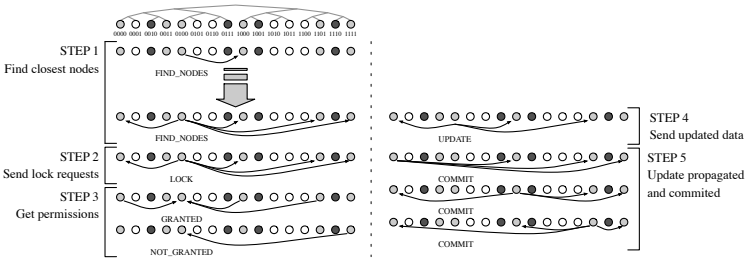


Fig. 1. A step-by-step representation of the update process

3 Design

3.1 Defining the update transaction

In Kademlia, any peer wishing to *get* a value sends `FIND_VALUE` RPCs to contacts from its routing table that are closest to the value's key. Nodes receiving the requests can either reply with data or return routing information. The process is iterative and continues until the key-value pair is found or no more routing options exist. Each query step involves issuing multiple concurrent requests (up to α), which reduces the total time needed for completion and helps in identifying and bypassing faulty peers. *Put* operations are based on a similar query loop, using `FIND_NODE` RPCs instead. Replies contain a predefined number of close nodes — equal to κ, a system-wide parameter which specifies the number of replicas maintained for each data item and controls the size of routing tables. The key-value pair is replicated to the κ closest nodes to the ID via `STORE` RPCs. Kademlia suggests that all key-value pairs are republished in this way every hour, and expire in 24 hours from their initial publication.

In order to support serializable data updates, XOROS requires multiple closely related messaging phases that form a single transaction. When a peer is instructed to *update* data, it first performs a `FIND_NODE` loop in order to find the κ nodes that store replicas of the specified key-value pair (Fig. 1). We consider these same nodes as the ones responsible for giving permission to update the corresponding key. The next step is to issue `LOCK` RPCs to all members of the quorum in order to request mutual exclusion. Nodes may reply to incoming lock requests with either a `GRANTED` or `NOT_GRANTED` message, depending on whether they have already granted their vote to another peer. Moreover, a `GRANTED` message signifies the permission to send an updated value. A requesting node can only send respective `UPDATE` RPCs to the nodes that have voted in its favor. We require that `UPDATE` RPCs are sent only after a node has successfully collected μ_{lock} permissions — a number that guarantees that no other, competing update issuer may gain mutual exclusion. `GRANTED` RPCs also contain the item's current value, in order to implement atomic read-modify-write transactions.

When a replica maintainer receives an `UPDATE` RPC, there is no assurance that the quorum has indeed concurred on the peer that should perform the update. The last step of the algorithm requires the closest nodes of the key-value

pair to communicate with each other via `COMMIT` RPCs in order to complete the transaction and propagate the correct data even to parts of the quorum that may have given their permission elsewhere. Quorum members consider a value as updated and store it, only if they receive μ_{store} `COMMIT` messages (including an initial `UPDATE` if applicable), all referring to the same data from the same issuer. Similar to μ_{lock}, μ_{store} protects the consistency of the transaction in the case of competing updates. Supposing that there are no malicious peers and no failures, it is evident that $\mu_{lock} = \mu_{store} = \kappa/2 + 1$. However, in every other case, μ_{store} has a different value and μ_{lock} is larger than μ_{store} by the maximum number of nodes that may exhibit arbitrary behavior or fail.

We assume that each peer has a good knowledge of its close peers and thus will know the quorum members of each data item it stores. Nevertheless, depending on the way each particular DHT implementation manages routing tables, this may require an extra messaging step. In XOROS, quorum members check their routing tables to certify that a peer is indeed part of the key-value pair's closest nodes, once they receive a `COMMIT` message. If the sender is not found, respective `FIND_NODE` queries are issued. Different RPCs are used for updating and committing a value, to be able to distinguish the respective semantics if the issuer is also a member of the quorum. Also, XOROS peers ignore `STORE` RPCs for key-value pairs already in their registry.

3.2 Tolerating incompatible behavior

Let us now consider a scenario where λ of the κ nodes forming the quorum are "corrupt", meaning they may execute the protocol in an arbitrary way. Let σ denote the number of nodes that can be trusted to conform to the protocol ($\kappa = \sigma + \lambda$). If we instruct all κ nodes to report the same value to each other, any node will receive at least σ consistent values. In the following discussion, we require that no message can fake its originator, which can be asserted by hashing the peers' network addresses into their corresponding IDs.

In every lock round, σ votes may be distributed among issuers competing for an update, while up to λ additional votes may be simultaneously granted to all. In the next step, assuming all issuers send `UPDATE` RPCs, σ nodes will forward a correct value, while λ may send anything or nothing at all. They may even retransmit `COMMIT` RPCs received, as if they have granted a permission to the issuer, producing an additional λ commit messages for each value.

Trustful quorum members that receive μ consistent update requests may safely store a value when they can be certain that no other competing value can be successfully propagated during the same transaction. From the μ messages received, at least $\mu - \lambda$ can be trusted. This leaves other issuers with the remaining $\sigma - (\mu - \lambda)$ trustful peers available to forward their update, plus an additional λ that may propagate any value. We set μ_{store} to the smallest possible integer value, so that even if λ out of μ_{store} values are reported from corrupt nodes, no other competitor can violate the system's consistency by producing μ_{store} conflicting updates. Intuitively, if any quorum member receives μ_{store} consistent update messages, the issuer has successfully gained mutual exclusion.

$$\mu_{store} > \sigma - (\mu - \lambda) + \lambda \overset{(\kappa=\sigma+\lambda)}{>} \frac{\kappa + \lambda}{2} \Rightarrow \mu_{store} = \lfloor \frac{\kappa + \lambda}{2} \rfloor + 1 \quad (1)$$

From an issuer's point of view μ_{store} is the minimum number of quorum members that must vote to his favor in order to proceed with the update. Nevertheless, there is no way to know how many corrupt nodes are contained within those μ_{store} peers and as a result how many will indeed forward the corresponding `COMMIT` commands. We *suggest* that each peer receives at least μ_{lock} `GRANTED` RPCs before proceeding with an update. Moreover, if we want the system to *guarantee* that an issuer conforming to the protocol will be able to commit an update, we must bind μ_{lock} to κ, which allows us to infer an upper bound for λ.

$$\mu_{lock} \geq \mu_{store} + \lambda \overset{(1)}{>} \frac{\kappa + 3\lambda}{2} \Rightarrow \mu_{lock} = \lfloor \frac{\kappa + 3\lambda}{2} \rfloor + 1 \quad (2)$$

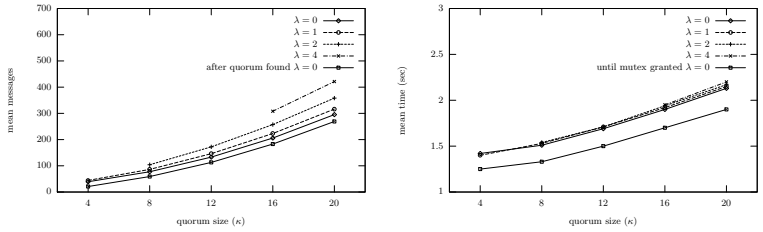
$$\kappa \geq \mu_{lock} \overset{(2)}{>} \frac{\kappa + 3\lambda}{2} \Rightarrow \lambda < \frac{\kappa}{3} \quad (3)$$

Note that XOROS, for a quorum of $3\lambda + 1$, can sustain up to λ corrupt members, which has been proven optimal [14]. Even if there are no harmful peers in the network, the behavior expected from up to λ peers may address node failures or arrivals. Involuntary bad behavior can also be expected when peers join the network or when the overlay is experiencing high levels of churn.

3.3 Protocol enhancements

XOROS also uses a modified *get* procedure. As up to λ closest nodes may report any value, a peer must locate and query at least $2\lambda + 1$ primary replicas in order to get a consistent (majority) reply, unless the specific key is currently updated, the update process is in its last phase and some quorum members are "slower" than others. We currently backoff and repeat reading a key-value pair for a predefined number of retries. Note that this implementation offers only loose consistency semantics and in some cases Byzantine replicas may influence the result by collectively reporting old values along trustful but slow quorum members. Although this will not yield an invalid result, an alternative *get* policy could instruct contacting more primary replicas, or even locking the quorum before sending `FIND_VALUE` messages.

To protect the mutual exclusion from greedy peers and avoid starvation, we require that all granted votes expire and issuers exponentially backoff after a round of unsuccessful `LOCK` RPCs. Both conditions are regulated by the quorum members themselves. Upon the receipt of a `GRANTED` message, the peer requesting the lock must issue an `UPDATE` or `YIELD` RPC before the lease expires. Failure to do so results in the node being banned from further lock requests for a large time interval by the respective voter. `UPDATE` RPCs also trigger a similar time counter. A non-Byzantine quorum member automatically releases a lock when the corresponding value is finally committed or any of the two timeouts expires. Thus, neither the update issuer, nor part of the closest nodes can deliberately or accidentally stall any phase of the update transaction.



(a) Mean messages per update (b) Mean time per update

Fig. 2. Results for the same request pattern and different values of κ and λ

4 Evaluation

The XOROS prototype is implemented in way that allows us both to deploy a network of peers and perform large-scale distributed packet-level simulations using the same codebase. In order to get an insight into the performance characteristics of XOROS, we created a scenario where 1024 peers, storing an equal number of items perform random updates at a constant rate of 1 $\frac{update}{sec}$. Results depicted in Fig. 2 correspond to multiple runs of the same scenario for different values of κ and λ. The messaging roundtrip latency is distributed between 120 and 180 milliseconds and Kademlia's α is set to 3.

The number of messages produced by each update operation follow a quadratic increase in respect to the quorum size. Looking back at Fig. 1, we can see that steps 2, 3 and 4 of the protocol produce at most κ messages each, while the commit round requires $\mu_{lock}(\kappa-1)$ messages, which contains a κ^2 term. On the other hand, it is evident from Fig. 2(b) that the messaging cost is distributed across the network. The first four lines of Fig. 2(b) show the total time required for each update transaction, measured from the moment the command is issued until the last quorum member commits the value. Also, the time needed for the issuers to complete steps 1 to 3 of the protocol is marked as "until mutex granted". There is a constant difference between the latter and the total time consumed by each command, representing the time needed to propagate an update. Notice that this is equal to the maximum message roundtrip latency. The mean time required for each transaction is affected by the quorum size only because of the protocol's first step, where we locate the closest nodes of a key-value pair. In Fig. 2(a) we also show the number of messages required for steps 2 to 5 of the update transaction (marked as "after quorum found"). Although the message count for the first step is relatively small, `FIND_NODE` RPCs are sent in waves of α parallel requests — a process that requires more time as κ increases.

5 Conclusion

DHTs are widely used by a multitude of large-scale, high-throughput applications. Their distinctive feature set has allowed many systems to seamlessly scale to Internet-wide deployments. Nevertheless, we believe that the DHT design

has not yet reached its full potential, as it restricts upper-layer services to a read-only repository. A mutable DHT can meet a broader range of application requirements and may prove extremely important for future implementations.

In this paper, we have presented a protocol in the context of an actual DHT implementation we call XOROS, that tackles the problem of performing consistent updates to the primary data replicas in DHTs. XOROS's foundation is previous work on distributed mutual exclusion and Byzantine-tolerant communication. We propose an algorithm, where the actions of locking a resource and committing a change are bound into the same transaction, in turn controlled and driven by the closest nodes of each key-value pair. The quorum responsible for mutual exclusion consists of the nodes that also hold the data item to be updated, thus we can implement fine-grain locking. Moreover, by having quorum members communicate with each other before committing data items to storage, we can handle cases where there are malicious or faulty peers in the overlay.

References

1. Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Looking Up Data in P2P Systems. Communications of the ACM **46** (2003) 43–48
2. Busca, J.M., Picconi, F., Sens, P.: Pastis: A Highly-Scalable Multi-User Peer-to-Peer File System. In: Proc. of the 11th Inter. Euro-Par Conference. (2005)
3. Chazapis, A., Zissimos, A., Koziris, N.: A Peer-to-Peer Replica Management Service for High-Throughput Grids. In: Proc. of the 34th ICPP. (2005)
4. Ghodsi, A., Alima, L.O., Haridi, S.: Symmetric Replication for Structured Peer-to-Peer Systems. In: Proc. of the 3rd DBISP2P. (2005)
5. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: A Read/Write Peer-to-Peer File System. In: Proc. of the 5th OSDI. (2002)
6. Stein, C., Tucker, M., Seltzer, M.: Building a Reliable Mutable File System on Peer-to-Peer Storage. In: Proc. of the RPPDS. (2002)
7. Maymounkov, P., Mazières, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: Proc. of the 1st IPTPS. (2002)
8. Wang, Z., et al.: Update Propagation Through Replica Chain in Decentralized and Unstructured P2P Systems. In: Proc. of the 4th IEEE P2P. (2004)
9. Roussopoulos, M., Baker, M.: CUP: Controlled Update Propagation in Peer-to-Peer Networks. In: Proc. of the USENIX Annual Technical Conference. (2003)
10. Kubiatowicz, J., et al.: OceanStore: An Architecture for Global-Scale Persistent Storage. In: Proc. of the 9th ASPLOS. (2000)
11. Lynch, N., Malkhi, D., Ratajczak, D.: Atomic Data access in Content Addressable Networks, A Position Paper. In: Proc. of the 1st IPTPS. (2002)
12. Velasquez, M.G.: A Survey of Distributed Mutual Exclusion Algorithms. Technical Report CS-93-116, Colorado State University (1993)
13. Lin, S.D., Lian, Q., Chen, M., Zhang, Z.: A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems. In: Proc. of the 3rd IPTPS. (2004)
14. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401
15. Malkhi, D., Mansour, Y., Reiter, M.K.: On Diffusing Updates in a Byzantine Environment. In: Proc. of the 18th IEEE SRDS. (1999)
16. Rodrigues, R., Liskov, B.: Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture. Technical Report LCS TR/932, MIT (2003)