ATOMIC COMMITMENT IN TRANSACTIONAL DHTS

Monika Moser
Zuse Institute Berlin (ZIB)
Berlin, Germany
moser@zib.de

Seif Haridi
Royal Institute of Technology (KTH)
Stockholm, Sweden
haridi@kth.se

Abstract We investigate the problem of atomic commit in transactional database systems built on top of Distributed Hash Tables. DHTs provide a decentralized way to store and look up data. To solve the atomic commit problem we propose to use an adaption of Paxos commit as a non-blocking algorithm. We exploit the symmetric replication technique existing in the DKS DHT to determine which nodes are necessary to execute the commit algorithm. By doing so we achieve a lower number of communication rounds and a reduction of meta-data in contrast to traditional Three-Phase-Commit protocols. We also show how the proposed solution can cope with dynamism due to churn in DHTs. Our solution works correctly relying only on an inaccurate failure detection of node failure which is necessary for systems running over the Internet.

Keywords: Atomic Commit, Database, Transactions, DHT, Paxos

1. Introduction

DHTs provide the ability to store and lookup data in a fully decentralized manner. They can be utilized to build a distributed database on top of it. We consider such a database which provides the user with an interface to perform transactions on its data, and where all operations on data are done in a transactional manner. Therefore an atomic commit mechanism is needed to build such a DHT-based database. database. Atomic commit guarantees that either all operations of the transactions take place or none of them. This means that only committed states are made visible. Another important mechanisms of distributed transactional systems is concurrency control, which ensures that concurrent transaction cannot interfere with each other. This paper is concerned with the atomic commit problem.

A typical transaction is a sequence with an arbitrary number of operations on different items. This sequence of operations is enclosed by a *Begin of Transaction (BOT)* and an *End of Transaction (EOT)*. BOT signals that a client or application wants to start a transaction. The end of the transaction is marked with the EOT. At this point the system has to ensure that either all of the operations contained in the transaction take place or none of them will affect the system. Therefore a node receiving EOT starts a distributed commit protocol where it determines whether all the nodes which are responsible for items that are involved in the transaction can do the operations. If all those nodes confirm that they can commit the transaction will be committed.

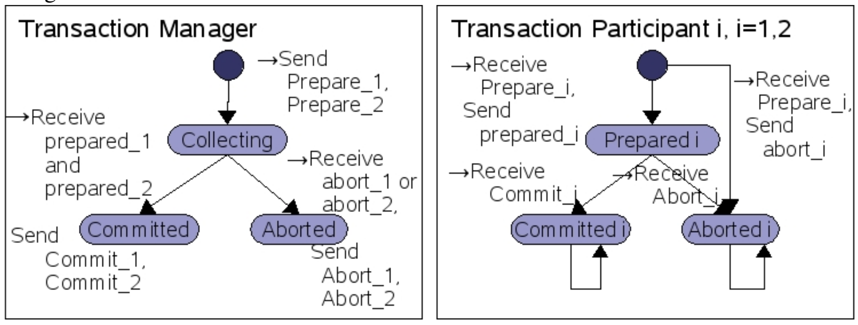
We propose a solution for atomic commit which is based on the Paxos commit algorithm introduced in [5]. We show how it can be adapted for a DHT-based database. The Paxos commit algorithm defines different roles for nodes running the protocol. We use the specific structure and services of the DHT to determine which nodes have to act in which role. As DHTs are systems that are highly dynamic, we show how we can cope with the dynamism and when we have to fix the group of nodes involved in the protocol. Another advantage of the Paxos commit algorithm is that it can handle a number of failures among the nodes without relying on a perfect failure detector, which is an important property for distributed systems running on the Internet.

Outline. Section 2 gives the problem description for this paper. In section 3 we describe the architecture of our system. Our approach for atomic commit in a transactional DHT-based database system is presented in 4. Section 5 lists some related work. As this paper summarizes some work in progress, we add an outlook on our future work to the final conclusions presented in 6.

2. Problem Description

We consider a storage system that is built on top of a DHT. DHTs are utilized to efficiently find data items stored in a P2P system. They use a hashing function to assign each data item consisting of (Key, Value) an identifier in a typically large identifier space. Each node that is part of the DHT is responsible for at least one subrange in the identifier space. Examples for DHTs are Chord [1], DKS [3], CAN [9] and Pastry [10].

Figure 1. State-charts for a 2-Phase-Commit Protocol with 2 Participants and 1 Transaction Manager


Transaction Manager
→Send Prepare_1, Prepare_2
→Receive prepared_1 and prepared_2
Collecting
→Receive abort_1 or abort_2,
Send Commit_1, Commit_2
Committed
Aborted
Send Abort_1, Abort_2
Transaction Participant i, i=1,2
→Receive Prepare_i, Send prepared_i
Prepared i
→Receive Prepare_i, Send abort_i
→Receive Commit_i
→Receive Abort_i
Committed i
Aborted i

There exist a number of storage systems which are built on DHTs, e.g. Bamboo[1] which is based on Pastry and DHash[2] which is based on Chord. Mostly items in such systems are replicated for a higher degree of availability and reliability. These systems are typically read-only storage systems.

Atomicity is one of the four ACID properties of transaction. A transaction will be executed either completely or will have no effects on the data at all. Changes on data made by a transaction will be made persistent when it reaches its *commit* point at EOT. A transaction will either end with *commit* or with *abort*, in which case the data modification are canceled, and the transaction has no effect. In distributed databases items involved in a transaction may be spread over different nodes. There is one node that acts as the *Transaction Manager (TM)*, which is responsible for coordinating the transaction. Nodes that are responsible for items which are involved in the transaction are the *Transaction Participants (TP)*. A transaction can only be committed if each of the TPs is able to commit its part of the transaction. All the TPs have to agree on the same outcome of the transaction. Well known solutions to this problem are *Two-Phase-Commit (2PC)* algorithms. In the first phase (voting phase) the TM initially asks all the TPs to prepare. The TPs answer whether they are able to commit. In the second phase (decision phase) the TM tells the TPs to commit if all the TPs are able to commit and make their changes durable. Figure 1 shows the possible states of a 2PC protocol with one Transaction Manager and two Transaction Participants.

One Problem with the basic 2PC is that it is a blocking protocol. If the TM fails in the decision phase (state Collecting) after the TPs have voted prepared, they are not able to receive the outcome of the transaction and are blocked. A number of non-blocking algorithms were introduced. *Three-Phase-Commit*

[1]http://www.bamboo-dht.org/
[2]http://pdos.csail.mit.edu/chord/

(3PC) algorithms introduce an extra phase to circumvent a blocking state. For DHT-based systems adding an extra phase might be very costly in terms of latencies. In particular if nodes are distributed worldwide. Most of them are also relying on timeouts, which might impact the performance for Internet-based systems with fluctuating link delays. We therefore use the Paxos based commit algorithm introduced in [5]. Instead of using an extra phase, votes of the TPs are sent to a number of so called acceptors. The non-blocking property is introduced at the cost of a higher number of messages, instead of an additional communication round. We think that in a P2P environment it is more important to reduce latency than reducing the number of messages sent to achieve an acceptable performance. Besides the size of the messages needed for the protocol is small. Another important property of the Paxos commit protocol is that it does not rely on a perfect failure detector.

Next we will describe the architecture of the system for which our solution is designed for.

3. Architecture of the Transactional System

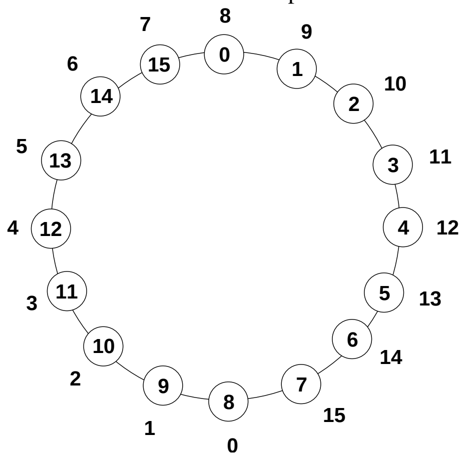
In DHT-based transactional database systems each node can act as TM and as TP. Clients and applications which invoke transactions are connected to arbitrary nodes in the DHT. Any such node will act as a TM for the transaction started by the associated client. During the commit phase all nodes which are responsible for an item that is involved in the transaction act as TPs. Items in our DHT are replicated. Our solution is illustrated with the symmetric replication scheme of the DKS DHT as mentioned below. With symmetric replication replicas can be accessed concurrently.

3.1 Symmetric Replication and Data Consistency

We consider symmetric replication as described in [4, 2]. The storage system replicates each item with the replication factor f. An identifier of an item is associated with $f - 1$ other identifiers. This corresponds to a partition of the identifier space in $\frac{N}{f}$ equivalence classes. Figure 2 shows an example of an identifier space of size $N = 16$ and a replication factor $f = 2$. For $f = 2$ items are replicated on the opposite side of the ring. Replicas are shown outside the ring. The identifiers for the replicas of an item with identifier id are determined using the following function: $r_i(id) = (id + (i-1)\frac{N}{f}) mod N \, for 1 \leq i \leq f$. Using symmetric replication, items can be accessed concurrently by determining their associated identifiers.

Our system maintains strong consistency among operations on data by including at least a majority of replicas in an operation on an item. All operations related to data enforce the invariant that a majority of replicas for a certain data item is up to date. Here, a majority must contain at least $\lfloor \frac{f}{2} \rfloor + 1$ replicas.

Figure 2. Associated identifiers from an identifier space with size $N = 16$ and $f = 2$ [4].



As write and store operations are performed on a majority, a read operation includes a majority as well, to ensure to get the latest version of an item. As a consequence join, leave and node failure handling have to maintain the replication factor. Especially they have to ensure that the number of replicas never becomes larger than f. When a new node joins the system, it gets the data it will be responsible for, and then takes over the responsibility from its successor node. There is no point where they are both responsible for the transfered items in order to ensure that the number of replicas for each item does not exceed f. When a node leaves it transfers the responsibility for its items to its successor node and thus again does not change the number of replicas for an item. When a node failure is detected, another node in the system becomes responsible this node's item. It will read the items from the remaining replicas. Here the number of replicas is restored to f after some time, but it does not increase the number of replicas

According to Brewer's conjecture [12] we will only be able to maintain availability until partitioned overlays merge. It is impossible to maintain consistency, availability and partition-tolerance at the same time. Our emphasis is on consistency.

3.2 System Properties

A DHT-based database system differs from a traditional distributed database system in a number of points that are important for the design of the commit algorithm. Traditional distributed database systems usually consist of a number of reliable nodes connected through a LAN. In contrast a DHT is built on unreliable nodes. The MTTF (Mean Time to Failure) of a node in a DHT system is typically much smaller. The need for a non-blocking atomic commit algorithm therefore is higher than in a traditional database system. Traditional database systems often are optimized for the failure-free case as failures occur quite seldom.

Another point is latency. In DHT-based database systems latencies are high due to the WAN communication paths and the routing structure of a DHT. A non-blocking atomic commit algorithm implemented in a DHT has to be low in the number of communication rounds to achieve acceptable performance.

The number of nodes involved in a transaction is typically much higher for a DHT-based system as items are distributed over a larger number of nodes. Traditional distributed database systems often consist of a server and some backup servers. In a DHT we even have two levels of distribution. First items are distributed to multiple nodes in the system and second items are replicated and again spread over the whole system. The number of nodes involved in the transaction depends on the number of items which are part of the transaction. An atomic commit algorithm for a DHT therefore has to be scalable in the number of participants.

The failure model for a traditional database system is normally based on a crash-recovery process model. In contrast there are several possible failure models for DHT-based database systems. In this paper we consider a DHT database system that is based on a crash-stop process model. When a node crashes and later recovers, it joins as a new node. Therefore it does not need to remember any previously stored data, nor logs of uncommitted transactions. Here we rely on the majority of nodes holding replicas of items involved in ongoing transactions will survive. Which is a consequence of our majority based consistency mechanisms.

The atomic commit algorithm we present in the next section assumes the crash-stop DHT model and symmetric replication. It is tailored for high latencies, high distribution of items and it can handle the failure of the TM.

4. Atomic Commit Protocol for a DHT

As mentioned above nodes of the DHT can act as TMs and as TPs. A client that invokes a transaction is connected to a node in the DHT. This node will be the TM for that particular transaction. Invoking a transaction will result in the creation of a transaction item, such that the key of the transaction item results in an identifier that belongs to the responsibility of the TM and which we refer to as the transaction-ID. This item consists of a transaction record and will be stored in the transaction manager and also symmetrically stored in the DHT.

As failures of nodes in DHTs may occur quite often, a non-blocking atomic commit protocol is needed. Gray and Lamport [5] introduce a commit protocol built on the Paxos consensus algorithm[7–8]. Our solution is an adaptation of this commit protocol to work for DHTs. The Paxos commit protocol uses a number of nodes that collect the votes of the TPs. These are called acceptors. In the case of a TM's failure the decision for the transaction can be requested from the associated set of acceptors. We adapt this protocol by having the set of nodes responsible for the replicated transaction item as our set of acceptors. Therefore the number of acceptors is determined by the replication factor of the whole system.

As mentioned above the Paxos commit algorithm provides an ability to circumvent the blocking problem of a Two-Phase-Commit protocol. In the next section we will briefly introduce the properties of the Paxos consensus algorithm and thereafter Paxos commit.

4.1 The Paxos Protocol

Paxos is an algorithm which guarantees uniform consensus. Consensus is necessary when a set of processes has to decide on a common value. Uniform consensus satisfies the following properties: 1. *Uniform agreement*, which means that no two processes decide differently regardless of whether they fail after the decision was taken; 2. *Validity* describes the property that the value which is decided can only be a value that has been proposed by some process; 3. *Integrity*, meaning no process may decide twice and finally 4. *Termination*, every process eventually decides some value [6]. Paxos assumes an eventual leader election to guarantee termination. Eventual leader election can be built by using inaccurate failure detectors.

Paxos defines different roles for the processes. There are *Proposers*, which propose a value, and *Acceptors*, which either accept a proposal or reject it in a way that guarantees uniform agreement. Paxos as described in [8] assumes that each process may act as both proposer and acceptor. In our solution presented below we use different processes as proposers and acceptors.

The above mentioned properties of uniform agreement can be guaranteed by Paxos whenever a majority of acceptors is alive. That means, it tolerates the failure of F acceptors out of initially $2F + 1$ acceptors.

Paxos basically consists of two phases called the read and write phase. In the *read phase* a process makes a proposal and tries to get a promise that his value will be accepted by a majority or it gets a value that it must adopt for the write phase. In the *write phase* a process tries to impose the value resulting from the read phase on a majority of processes. Either the read or write phase may fail. Proposals are ordered by proposal numbers. By using an eventual leader to coordinate different proposals, the algorithm will eventually terminate.

4.2 Atomic Commit with Paxos

Uniform consensus alone is not enough for solving atomic commit. Atomic commit has additional requirements on the value decided. If some process proposes abort or is perceived to have crashed by other nodes before a decision was taken, then all processes have to decide on abort. To decide on commit, all processes have to propose commit.

In the Paxos Commit protocol [5] we have a set of acceptors, with a distinguished leader, and a set of proposers. The set of acceptors play the role of the coordinator and the set of proposers are those who have to decide in the atomic commit protocol.

Each proposer creates a separate instance of the Paxos algorithm with itself as the only proposer to decide on either prepared or abort. All instances share the same set of acceptors It can be noted that the Paxos consensus can be optimized, because there is only one proposer for each instance. If a proposer fails, one of the acceptors, normally the leader, acts on behalf of that proposer in the particular Paxos instance and proposes abort.

Acceptors store the decisions of all proposers. Whenever an acceptor collects all decisions it sends commit or abort to the leader. A leader needs to receive the decision of a majority of acceptors to do the final decision. Thereafter the final abort/commit is sent to the initial proposers. If the leader fails by the eventual failure detector, another leader will take over and can extract the decision from a majority of acceptors and complete the protocol.

The state-chart of a proposer is similar to the state-chart of a TP in the original 2PC protocol, as shown in figure 1. Also the state-chart of an acceptor is similar to that of the TM, referring to the same figure. But instead of sending the decision commit to the participants, the acceptors send the outcome to the leader.

4.3 Adapted Paxos Commit for a DHT

Paxos is designed for a static environment with a fixed number of participants and acceptors. However each transaction involving items of a DHT has different nodes involved. Every node responsible for an item in a transaction becomes a TP for that particular transaction. In fact the TM initially does not know which nodes are TPs. The number of nodes varies according to whether or not the node is responsible for an item that is involved in the transaction. As mentioned earlier, each transaction has a certain transaction item. We therefore use a certain group of acceptors for each particular transaction, that can be easily determined from the transaction-ID of the transaction item. We use symmetric replication to determine the set of acceptors. The set of acceptors consists of the nodes responsible for a replica of the transaction item. One advantage is that we create a pseudo static group of acceptors. The group of acceptors is fixed temporarily by the TM just before the prepare request is sent to the TPs. With the prepare request the TM informs the nodes responsible for items in the transaction about the set of acceptors. When such a node receives the prepare request it becomes a TP and starts its Paxos instance.

At this stage the group of TPs and the group of acceptors are fixed. It will remain fixed during the atomic commit phase. If a node joins/leaves in a DHT, the responsibility of certain items has to be transferred. The transfer of the responsibility of items involved in an active commit protocol is deferred until the protocol instance terminates.

One modification to the Paxos commit is that the acceptors collect the votes from the TPs and classify them per item. When a majority of replicas of an item votes prepared, the acceptors record a prepared vote for this specific item. If the decision is prepared for all items the transaction commits.

When a TM knows the decision for the transaction, it can store this information in the transaction item. This item can then be replicated in the DHT just like regular data items. Whenever a TP does not receive the result of the transaction from the TM it can query the result of the transaction by reading the transaction item stored in the DHT.

In contrast to a number of 3-Phase-Commit protocols TPs do not need to know each other. Therefore the meta-data for transactions can be kept small. Also the overall amount of meta-data kept in the DHT is smaller, since only acceptors have to keep a record of the transaction. As mentioned earlier, the number of communication steps is also smaller.

Another issue is garbage collection of transaction items. As information on previous transactions grows by time, garbage collection is needed to throw away information which is no longer needed. This can be done in different ways either by acknowledgment messages or expiry date associated with transaction items.

Most of the operations mentioned in this particular DHT-based Paxos commit are operations on a set of identifiers. This is supported efficiently by bulk operations in DHTs as described in the DKS system[4, 2].

5. Related Work

In [11] Paxos is used to achieve consensus in DHTs. The authors present a middleware service called PaxonDHT which provides a mean to guarantee strong consistency among a set or replicas. In contrast to PaxonDHT our work is providing an approach for atomic commit with replicas of several items involved.

OceanStore [13] provides the ability to concurrently update data stored in a global persistent data store. A master replica is required which consists of a set of nodes which run a Byzantine agreement protocol to cooperate with each other.In [13] the authors mention that transactions could be built on top of the API of OceanStore. Our work considers a system that provides transactions in its own interface and provides strong consistency among operations on data.

6. Conclusion and Future Work

We presented a framework for having transactions on DHTs and consequently strong notion of data consistency in DHTs. We focus on the atomic commit problem. Our solution is based on the Paxos commit algorithm. We showed why Paxos commit is suitable for DHT-based systems and how we can adapt it for transactional DHT-based databases. Among processes Paxos commit defines a set of acceptor and a set of proposers. Our approach uses the symmetric replication scheme for DHTs to determine a pseudo static group of acceptors. The non-blocking property of this commit protocol is important as failures in DHTs occur quite often. Another advantage is a lower number of communication rounds compared to traditional non-blocking algorithms in distributed database systems like Three-Phase-Commit. Paxos commit can handle a number of failures among the processes which are involved in the atomic commit without violating the properties of atomic commit. Further we showed how to handle dynamism in a DHT due to churn. We defined the phases when it is necessary to fix the group of participants in the algorithm to enable a correct atomic commit.

There is a number of issues left, that will be addressed in the future. We will investigate in a concurrency control for a DHT-based database system. An optimistic concurrency control seems reasonable for this scenario. One solution will be a timestamp based ordering. Further we will evaluate the whole architecture and specify the algorithms formally.

Acknowledgments

This research work is carried out under the SELFMAN project funded by the European Commission and the Network of Excellence Core-GRID funded by the European Commission.

References

[1] I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, 149-160

[2] A. Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables KTH. Doctoral Dissertation, KTH — Royal Institute of Technology, 2006

[3] L. Onana Alima, S. El-Ansary, P. Brand and S. Haridi DKS (N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, 2003

[4] A. Ghodsi, L. Alima and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *The 3rd Int Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 2005

[5] J. Gray and L. Lamport. Consensus on transaction commit. In *ACM Trans. Database Syst.*, ACM Press, 2006, 31, 133-160

[6] R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag, 2006

[7] L. Lamport. Paxos Made Simple. 2001

[8] L. Lamport. The part-time parliament. In *ACM Trans. Comput. Syst.*, ACM Press, 1998, 16, 133-169

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ACM Press, 2001, 161-172

[10] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001, 329-350

[11] B. Temkow, A. Bosneag, X. Li and M. Brockmeyer. PaxonDHT: Achieving Consensus in Distributed Hash Tables In *SAINT '06: Proceedings of the International Symposium on Applications on Internet, IEEE Computer Society*, 2006, 236-244

[12] S. Gilbert and N. Lynch Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services In *SIGACT News*, 2002

[13] J. Kubiatowicz , et al. OceanStore: An Architecture for Global-scale Persistent Storage In *Proceedings of ACM ASPLOS*, 2000