Project no. 034084
Project acronym: SELFMAN
Project title: *Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components*

European Sixth Framework Programme
Priority 2, Information Society Technologies

Deliverable reference number and title:	D.3.1 First report on formal models for transactions over structured overlay networks
Due date of deliverable:	July 15, 2007
Actual submission date:	July 15, 2007
Start date of project:	June 1, 2006
Duration:	36 months
Organisation name of lead contractor for this deliverable:	ZIB
Revision:	1
Dissemination level:	PU

Contents

1 Executive summary

In this document the work done in workpackage 3 of the SELFMAN project is presented. The aim of workpackage 3 is to design and build a self-managing storage service, which will provide data replication and the ability to perform transactions. It will be built on top of a structured overlay network. This report contains first results on the development of formal models for transactions over structured overlay networks (D3.1).

When building a transactional storage service on top of a structured overlay network (SON) we have to deal with different system properties than in classic distributed database systems. Nodes which are part of the structured overlay network crash more frequently than rather highly reliable nodes of classic database systems. In an Internet-based SON we even cannot rely on having perfect failure detectors. It is impossible to distinguish between a node which has failed and a node with a long communication delay. Also the system has to be scalable to be able to handle a growing number of nodes. Protocols for the transactional storage service on top of a SON have to deal with dynamically changing set of nodes which constitute the system.

In this report we present an outline of a framework for DHTs which provide strong data consistency and transactions on their data. Two different approaches for replication and consistency of data are introduced. The first one is based on symmetric replication and the other one on so called cells. To provide transactions on the data basically two mechanisms are needed. One mechanism guarantees that either all operations of a transaction are performed or none of them will affect the storage system - **Atomic commit**. We developed an atomic commit protocol which is non-blocking and does make process despite a number of failures. The second mechanism ensures that concurrent transactions do not interfere with each other - **Concurrency control**. Different concurrency control mechanisms which take into account different characteristics of transactions are presented in this report.

2 Contractors contributing to the Deliverable

ZIB (P5) and KTH (P2) have contributed to this deliverable.

ZIB (P5) ZIB has contributed on the transaction model. The deliverable mainly includes content of two papers [6], included in Appendix B, and [5], included in Appendix A, together with additional work done on the topic.

KTH (P2) KTH contributed on the transaction model, especially to the paper [5], included in Appendix A.

3 Results

3.1 Introduction

The storage service is a core service for self-managing applications. Research in structured overlay networks (SONs) and Distributed Hash Tables (DHTs) has lead to systems with basic primitives on which a self managing storage service can be built. They provide efficient routing and communication algorithms, fault tolerance, dynamic behavior and the ability to store data in a distributed and replicated way. However most storage systems built with DHTs concentrate on providing low latency for operations on their data items. Our focus is on building a storage system with strong consistency guarantees on its data and on providing the ability to perform transactions on the data.
In the following we show how a transaction can look like. A typical example for a transaction are bank accounts. The following one includes operations on instances of three different bank accounts A, B and C.

```
Begin of Transaction
    a.withdraw(100);
    b.deposit(100);
    c.withdraw(200);
    b.deposit(200);
End of Transaction
```

Assume the bank accounts were distributed on three different nodes. Distributed transactions require the coordination of all nodes which are responsible for data addressed in the transaction. One issue of transactions is to ensure that either all nodes can execute the operations on their data or none of them will make changes to the data (*atomicity property*). Another issue is to ensure that concurrent transactions do not interfere with each other. They may not see intermediate results made by other transactions (*isolation property*).

DHTs store data as key-value-pairs which are referred to as items. Transactions considered in this report include operations on such items, which are distributed over the nodes of the DHT.

To ensure the *atomicity property* we developed an atomic commit protocol which is based on Paxos atomic commit. Paxos atomic commit is a protocol which can handle a number of failures and does not rely on a perfect failure detector. We provide an adaption of this protocol which takes into account the consistency mechanisms of the underlying storage system and utilizes the replication scheme to create pseudo static groups of nodes which run the atomic commit protocol.

To ensure the second property of transactions - *isolation* - concurrency control mechanisms are needed. For centralized and distributed database systems there already exist a number of concurrency control mechanisms. To utilize them in structured overlay networks, they have to be adapted to take into account the systems properties. We worked on two general approaches of concurrency control, a simple protocol that is well suited for low-conflict scenarios and a more complex protocol which is simultaneously targeted at long-running update and simple read-only transactions. Both protocols are based on optimistic concurrency control.

This is a natural choice for structured overlays that helps reducing the impact of replication on communication overhead related to concurrency control.

Atomic commit and concurrency control are considered in the context of two different replication mechanisms. One is based on symmetric replication as described in DKS [1, 2]. To ensure consistency of the data items all operations on them have to include a majority of nodes. Besides the model with symmetric replication which provides consistency and replication on top of a DHT, we additionally present an outline of a system where a DHT is built on top of replicated nodes, which are called cells. These cells consist of a number of nodes which constitute a replicated state machine and thereby provide consistency and replication within the cell. On top of this cell the DHT is built.

Finally we show how the storage system can be utilized to build a distributed Wiki on it. A distributed Wiki is one of the usage scenarios which are presented in the deliverable of workpackage 5.

Outline. In section 3.2 we introduce differences between transactional DHTs and classic databases. Section 3.3 describes replication and consistency mechanisms of a storage system to which transactions are related to. An outline of how to do transactions in a DHT is given in section 3.4. The usage scenario is presented in section 3.5.

3.2 Transactional DHTs vs. Databases

Classic distributed database systems usually consist of reliable servers. As a consequence they consider the failure of a server as a seldom event and optimize their protocols for the failure-free case. However in transactional DHTs the failure of a node is a normal event. Protocols for these systems have to make progress despite the failure of nodes and provide a correct execution even if a node is falsely suspected to have crashed by an imperfect failure detector.

Whereas in classic databases the failure model usually is crash-recovery, we consider a crash-stop failure model. If a node of a DHT fails it would rejoin the system as a new node.

Due to the dynamic behavior of nodes in a DHT - frequent joins, leaves, crashes - the protocols for transactional DHTs have to deal with dynamic groups of nodes involved in a transaction. In classic distributed database systems the set of nodes is rather static.

3.3 Replication and Consistency Mechanisms

In this section we give an outline of the replication scheme and consistency mechanisms for a transactional DHT. Models for transactions have to be considered in conjunction with these mechanisms.

We will present two different replication models. One which uses symmetric replication within the DHT and another one which is built on so called cells.

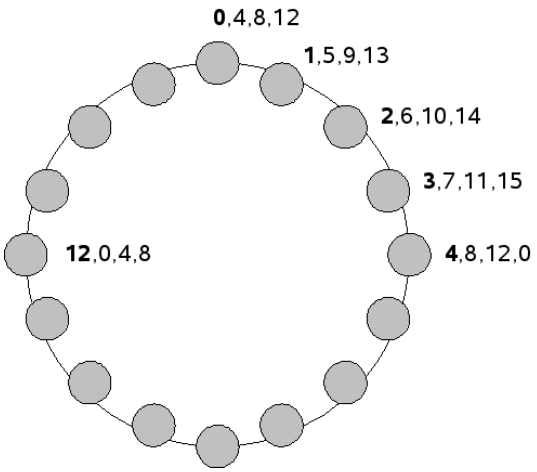


Figure 1: Identifier space of size $N = 16$ with replication factor $f =$, using symmetric replication

3.3.1 System Model with Symmetric Replication

Replication

Items which are stored in the DHT are replicated according to the DKS symmetric replication scheme [2, 1]. The number of replicas for each item is determined by the system's replication factor f. Symmetric replication describes how to find the nodes which are responsible for a replica. There each identifier is associated with $f-1$ other identifiers. Figure 1 illustrates part of an identifier space of a DHT with size $N = 16$ and replication factor $f = 4$.

Consistency mechanisms

A short outline of our consistency mechanisms is included in "Atomic Commitment in Transactional DHTs" of Appendix A. In the following we give a more detailed outline of them. Basically our consistency mechanisms are including a majority of nodes holding a replica in each operation on an item. The number of nodes constituting a majority is dependent on the replication factor. A majority has to include $\lfloor f/2 \rfloor + 1$ nodes. The system makes progress as long as a majority of nodes storing a replica is available.

A second requirement is that each item gets a version number. Operations on items work as follows:

- **Write:** A write operation first has to read the latest version number of the item it wants to write. Therefore it has to query at least a majority of the nodes being responsible for a replica. In a second phase the write operation tries to impose the write on at least a majority of nodes together with a timestamp which is higher than the one from the current version.
- **Read:** A read operation queries at least a majority of nodes being responsible for a replica of the item. The result will be the value of all returned replicas with the highest timestamp.

By including a majority in both the write and the read operation we ensure always to include the latest version of the item. However instead of using a majority one could also implement another quorum based mechanism, where e.g. read an write operations do require a different number of nodes.

In addition to operations on items, mechanisms to handle node join, leave and failure on the data level also have to maintain the invariant that at least a majority of nodes is storing the latest version of an item.

- **Node join:** Whenever a node joins the system and becomes responsible for a certain range of identifiers in the identifier space, it has to retrieve the items it will be responsible for. Either it will get them from the node which was responsible for the replicas before and therefore just replace this node without violating the majority invariant. Another approach would be to retrieve the items by reading from a majority being responsible for replicas of an item. Here we would possibly even increase the number of nodes holding a replica with the current version of the item.

- **Node leave:** This is done accordingly to the node join. Either the node which will take over the responsibility for the items of the leaving node gets the replicas from the leaving node or it will read from a majority of nodes.

- **Node failure handling:** If a node is considered to have crashed another node in the system has to take over the responsibility of the crashed node's replicas. Therefore it reads the items from a majority of nodes being responsible for them and thereby possibly increases the number of nodes storing a current version of the item as the crashed node could have stored an old version.

A consequence of providing strong consistency on data is that we cannot optimize our system for availability and network partition tolerance at the same time according to Brewer's conjecture [3]. A data item will only be available as long as a majority of nodes that is responsible for a replica is available. During a network partition the part with a minority of nodes won't be able to operate on the data.

3.3.2 System Model based on Cells

As an alternative approach to replication and consistency, we have developed the cell model. The general idea of the cell model is to build the DHT on top of groups of replicated nodes (cells) in contrast to doing replication on top of the DHT.

Instead of constructing the overlay from single nodes it is built from cells. Each cell is a group of physical nodes that together constitute a replicated state machine [8]. Cells are implemented using suitable total order broadcast or atomic group multicast protocols. Such protocols are usually based on some variant of consensus and provide replicated, atomic, and totally ordered operations over the state of the cell. This state is stored at each physical node and kept synchronized through the chosen protocol.

Cells have to use a *dynamic* atomic group multicast protocol (e.g. [7]). Dynamicity allows adding new and removing old nodes from the cell. When a new node enters the system, it is added to a randomly selected cell. The chosen protocol has

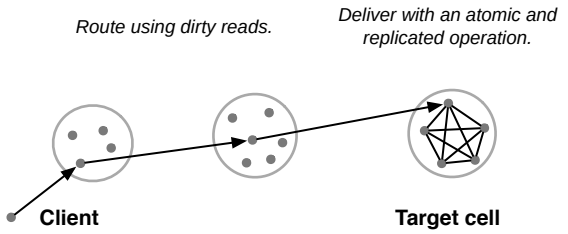


Figure 2: Routing in the cell model

to ensure that the new node is provided with the current cell state before it gets activated as a full member of the cell.

When cells consist of too many nodes they split into separate new cells to reduce communication overhead. Cells lacking enough nodes, are either dissolved by merging remaining nodes and their data with an adjacent cell (topology maintenance) or are healed by adding new replacement nodes from neighboring cells (node re-provisioning). Adjacency/Neighborship here depends on the chosen overlay topology. For the future, we intend to look deeper into these dynamic aspects of the cell model, especially the assignment of nodes to cells (e.g. using simulation techniques).

By basing cells on the aforementioned protocols, they already provide (per-cell) atomicity as a building block for a full transaction processing scheme. Additionally, the use of atomic group multicast protocols increases availability through replication. Together with topology maintenance and node re-provisioning, this can be exploited to mask churn from node failures *completely*.

Node re-provisioning reduces the number of necessary topological changes. Topology maintenance ensures that the one-on-one mapping between cells and key-space partitions always is complete and non-disjoint. Through this, atomic delivery of replicated data operations only occurs at a well-defined and currently responsible cell. Thus the cell model allows the creation of reliable overlay networks that guarantee lookup consistency – an important requirement for proper resource management in any transaction processing scheme.

Topology maintenance requires atomically updating the routing table of multiple, adjacent cells. This can be implemented using transaction processing. The cell model approach here leads to a circular synergy: Topology maintenance using cells provides lookup consistency. Lookup consistency is a necessary precondition for transaction processing. Transaction processing in turn can be used to implement topology maintenance.

The main disadvantage of the cell model is the high communication cost in terms of messages and, more important, latency that is associated with replicated operations. Thus any overlay built using the cell model must minimize the number of such operations. For regular routing, this can be done by using dirty reads. Intermediate routing steps are executed using dirty reads at single nodes of the routing cell. Final message delivery is executed using a replicated operation. If this delivery attempt fails because the presumed target cell is not responsible for the message, routing continues. This scheme has the benefit that alternative intermediate routing hops can be selected using some latency estimation metric.

The main advantages of the cell model are its ability to mask churn and thus ensure lookup consistency, saving unnecessary replica lookups (cell nodes know each other), providing atomicity, exchangeable failure models and replication schemes at the level of single cells, and supporting the aforementioned latency minimizing node selection strategies. In addition, the cell model is topology agnostic and only depends on a notion of cell neighborship. Whether these advantages outweigh the heavy cost induced of state machine replication protocols is an open question for future work.

3.4 Distributed Transactions over Structured Overlay Networks

The storage service built on top of a SON will provide a transactional interface. The kind of transactions we consider here are simple transaction with a sequence of operations on a number of items. Items which are involved in a transaction can be distributed over a number of nodes. However we have even two levels of distributions as items are replicated and replicas again are distributed over a different nodes. The set of nodes which are involved in a particular transaction protocol can differ for each transaction.

3.4.1 Properties of Distributed Transactions

Transactions provide the so called ACID-Properties:

- **Atomicity:** Operations of a transaction are performed in an all-or-nothing manner.
- **Consistency:** A transaction does not violate the system's consistency rules. After completion the system is in a legal state.
- **Isolation:** Each transaction is independent from other transactions. Concurrent transactions can't observe one another's intermediate results.
- **Durability:** Once a transaction is committed the results are persistent.

For the atomicity protocol we have to ensure that all nodes that are responsible for an item have to decide on the same outcome of the transaction. Basically this requires an atomic commit protocol which is a sort of consensus among them. The consistency property will be ensured by enforcing the majority invariant of the consistency model described in 3.3.1. To isolate transactions from each other we have to provide means to detect concurrent transaction and prevent them from interfering with each other. This is covered by the concurrency control mechanism. Durability has to be considered dependent on the assumption that a majority of nodes storing a replica will survive, which is an assumption of the whole framework in order to work correctly.

3.4.2 Transaction System

We define two different roles for nodes involved in a transaction. One role is the transaction manager (TM), which is responsible for coordinating the transaction. The other role is the transaction participant (TP). A TP is a node which is responsible for a replica of an item that is involved in the transaction. Each node in the system can both act as TM and as TP.

In the following we describe two important mechanisms for transactions: an atomic commit protocol and concurrency control mechanisms. We give different approaches for concurrency control which are optimized for transactions with different characteristics.

3.4.3 Atomic Commit

An atomic commit protocol ensures that all nodes which are involved in the transaction decide on the same outcome of the transaction. The outcome of the protocol will be commit, if all nodes are prepared and could commit their part of the transaction. The overall outcome will be abort if there exists at least one node that wants to abort the transaction. This constitutes a special case of consensus. All nodes have to decide on the same outcome, even if they fail. The last property is called uniform agreement.

Adapted Paxos Commit

In "Atomic Commit for Transactional DHTs" included in Appendix A we describe an atomic commit protocol for transactional DHTs. There we provide an outline for a DHT that provides strong data consistency and describe how atomic commit can be implemented there. The atomic commit protocol is based on the Paxos atomic commit [4] which was introduced by Lamport and Gray. Paxos is a majority based consensus protocol. It uses a set of acceptors which collect the votes of the participants and it can handle the failure of a minority of the acceptors. We adapted this protocol to a DHT with symmetric replication. We utilize the replication scheme to determine a group of acceptors which are responsible for the transaction management. The decisions of the transaction participants whether they are prepared and could commit their operations or whether they have to abort are distributed to this set of acceptors. In contrast to the most common atomic commit protocol, 2-Phase-Commit (2PC), this protocol is non-blocking. 2PC is blocking as it uses a single transaction manager. The failure of the TM after having collected the decisions of the participants causes the participants to be blocked as they cannot retrieve the outcome of the overall transaction and do not know whether they have to abort or make the changes permanent. By introducing a set of acceptors the outcome can still be retrieved from the acceptors as long as a majority of them is alive. As in our system the acceptors have the role of the TM we call them replicated transaction managers, while there exists one of them acting as the leader. The leader usually is the node to which the client or application is connected to. Figure 3 illustrates how the set of nodes for a transaction is determined. For simplification we assume an ID space of size $N = 16$, with replication factor $f = 4$, where for each ID there exists one node. On the top left-hand of Figure 3 nodes which participate in the

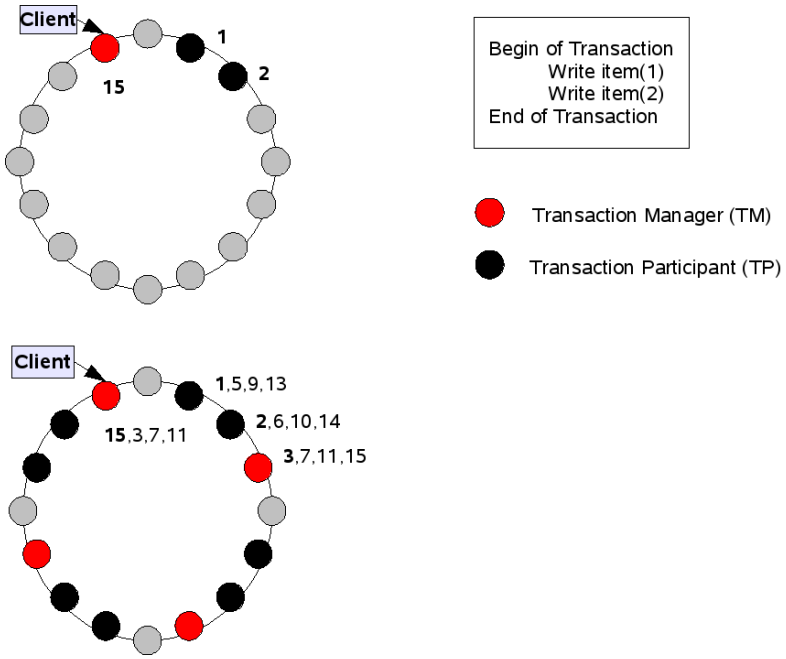


Figure 3: Transaction system for the transaction example shown on the right side. The top left side is without replication, the bottom left side with replication

protocol for the transaction example of the top right-hand part of the figure are shown without replication. There exists one transaction manager which is the node the client issuing the transaction is connected to. The nodes which are responsible for the items involved in the transaction become transaction participants. The part of the figure below illustrates the roles the nodes take when taking into account replication. There are three additional nodes which act as replicated transaction managers and three additional transaction participants for each item. For each replica of the item exists one transaction participant.

Further we adapt the protocol to the consistency mechanisms of the storage system. The set of TPs which vote in the protocol is composed by the nodes which are responsible for a replica of an item involved in the transaction. As our read and write operations require a majority of nodes we also decide on prepared or abort for each item by requiring a majority of nodes being responsible for a replica to vote for the same decision. This means that the decision for one item can be commit even if there exists one node holding a replica that votes to abort, as long as a majority votes to commit. The overall decision for the transaction is retrieved from the respective decisions for each item. It will be abort, if for at least one item the decision is abort, otherwise it will be commit, if for all items the decision is commit.

Figure 4 illustrates the whole protocol as it is described in the paper in Appendix A. First the leading TM sends a *Prepare* message to all participants. This message basically starts the transactions. Each TP decides whether it is prepared and could validate its part of the transaction or whether it's decision is to abort the transaction. Then each TP sends a *Prepared* or an *Abort* message to each replicated TM. Whenever a replicated TM has collected for each item *Prepared* messages from a majority of nodes being responsible for a replica of the item, it sends *Commit* to

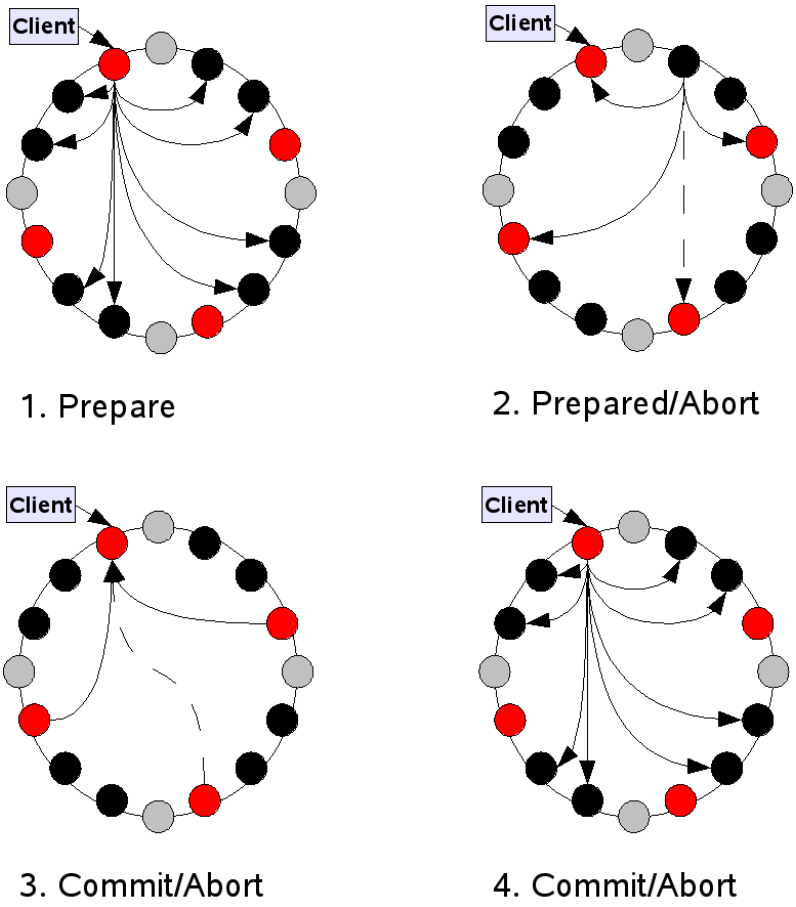


Figure 4: Communication steps of the adapted Paxos atomic commit protocol

the leading TM. Otherwise it sends *Abort* to the leading TM. The leading TM can send the decision of the transaction to the TPs as soon as it has collected the decision from a majority of replicated TMs. As we are requiring only a majority of nodes to answer for certain phases we have used dashed arrows for the messages which aren't necessary to switch to the next step.

Evaluation of Paxos Atomic Commit

Paxos Atomic Commit needs one more message delay than 2PC for the failure free case [4]. This is the theoretically minimal message delay for a non-blocking protocol. It requires around $2fi$ (f : replication factor, i : number of items involved in the transaction) more messages than 2PC, with a negligible size of messages.

We implemented Paxos Commit in Erlang and ran it on PlanetLab to get some latency measurements. We simulated a failure-free execution of the protocol with a varying replication factor and a varying number of items involved. The nodes which participated in the execution of the distributed protocol were distributed all over the world. Figure 5 shows the latencies for the commit protocol and the round trip time measurements between TPs and TMs. Round trip times where gained asynchronously, therefore they only reflect an approximation of the round trip times during the atomic commit protocol. The implementation of the the protocol does

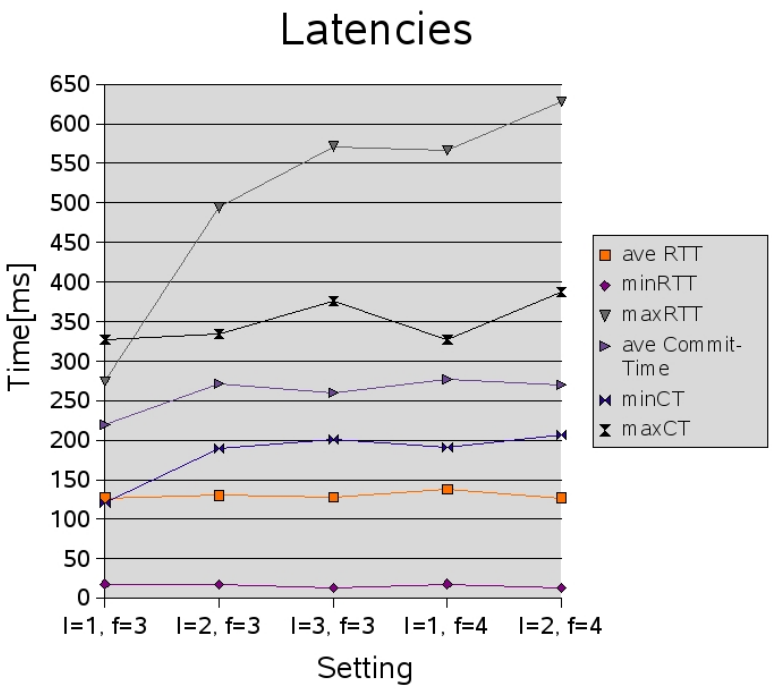


Figure 5: Latency Measurements for the failure free execution of Paxos Atomic Commit on PlanetLab, I : number of items involved, f : replication factor

not yet reflect the consistency mechanisms. It required that all replicas for an item have to be able to commit instead of a majority of replicas. The simulation included 20 runs for each particular setting. For each setting we extracted the average, maximum and minimum commit latency as well as the average, maximum and minimum round trip time (RTT). As expected the average commit time was not increased with a larger number of nodes involved in the protocol, since this does not lead to a higher number of communication steps. For example the setting with $I = 2, f = 3$ involves 9 nodes. 3 nodes for each item and 3 nodes for the replicated transaction managers, whereas the setting $I = 2, f = 4$ involves 12 nodes. However we can see that the maximum round trip time seems not to be reflected in the maximum commit time. Since the leading transaction manager can make an overall decision on the outcome of the transaction if it got the single decisions from a majority of the replicated transaction managers, it does not have to wait for the minority of the replicated transaction managers which introduce the highest communication delays.

3.4.4 Hybrid optimistic concurrency control

In the Wiki scenario, some hotspot pages (e.g. pages covering current events) will be frequently read and written by many clients. This can lead to high abort rates of update transactions. In order to support such long running update transactions we are investigating the adaptation of hybrid optimistic concurrency control (HOCC, [9]) techniques to structured overlay networks. Additionally, we have enhanced HOCC with support for read-only multiversioning (ROMV). ROMV provides fast read-only

transactions that can be utilized to read Wiki pages quickly.

This work is part of an ongoing effort to develop a suitable transaction processing protocol for the cell model as outlined in the article "Distributed Wikis on Structured Overlays". Therefore, in the following, we assume that atomic operations (e.g. for locking) can be executed for each replica group without specifying a concrete mechanism that achieves this.

Overview

HOCC is an optimistic multi-phase concurrency control method that allows guaranteed progress of update transactions under the assumption of access invariance. Access invariance is given if read and write sets of failed transactions do not grow during their re-execution.

HOCC transactions start in the work phase. All objects are accessed optimistically. Reads are performed against a single, possibly outdated replica. Writes are performed in a local transaction workspace.

At commit time, the transaction first enters a validation phase. Validation starts by getting all required locks. Locks are kept until the transaction finally aborts or commits (Strong two-phase locking, SS2PL). Next, it is verified that no conflicts exist between operations on accessed objects accessed and operations of other concurrently executed and already committed transactions. Conflicts arise when a concurrently executing transaction (a) overwrites already written objects, (b) overwrites previously read objects, or (c) reads objects that are overwritten later by the validated transaction. If no such conflicts are detected, the transaction is committed using a distributed atomic commit protocol.

If validation fails, the TM aborts the transaction and initiates transaction re-execution without releasing any locks (re-execution phase). All locks are kept and thus validation of the re-execution is guaranteed to be successful if the transaction is access invariant.

The distributed execution of HOCC requires handling one major problem: In order to avoid deadlocks, validation starts of transactions must be executed in the same order at all replica groups.

In a LAN scenario this can easily be ensured by using a central node or a suitable message broadcast medium. For a structured overlay in a WAN, these solutions are unsuited. However, a different approach using timestamps can be used. This is explained next.

Validation synchronization with timestamps

We assume that every node has access to a loosely synchronized local clock that can be used for the generation of timestamps. Uniqueness of timestamps is easily ensured by appending a node id. For every replica group or cell a monotonically increasing validation timestamp is maintained. This timestamp is always updated atomically. In the cell model, this can be done using replicated operations.

When validation starts, the TM suggests a global transaction commit timestamp to a TP of each replica group. This timestamp is selected larger than the local time and the largest known validation timestamp. To support this, the current validation time stamp of replica groups is communicated as part of regular control messages.

Upon receipt of the initially suggested commit timestamp, each TP tries to update its replica groups validation timestamp. This is only successful, if the suggested timestamp is actually larger than the replica group's current validation timestamp. After that, validation with locking is performed for the replica group. In any case, the TM is informed of the outcome. If all replica groups acknowledge the commit timestamp, regular commit can be performed. Otherwise, a new timestamp is chosen by the TM. Already validated replica groups discard previous validation results and drop associated locks upon receipt of a newly suggested commit timestamp. They then continue as if this timestamp would have been the initially suggested commit timestamp and repeat the protocol.

Read-Only Multiversioning

HOCC can easily be extended with support for read-only transactions using multiversioning. Read-only multiversioning (ROMV) has the advantage, that it does not require the execution of an atomic commitment protocol.

Multiple versions of all objects are stored in the system. Versions are identified by the commit timestamp of the transaction that wrote them. In addition, for every object a *current* version is maintained.

Update transactions execute using HOCC on the current version of objects. The current version is implicitly associated with the replica group's validation timestamp. At commit, new versions of all written objects are created using the transactions suggested commit timestamp.

Read-only transactions are associated with their start timestamp. Read operations return the youngest object version that is older than the reading transaction. If this version is the object's current version, two cases arise. (a) If no update transaction has entered validation at this replica group, it is sufficient to update the group's validation timestamp to the start timestamp of the read-only transaction and then proceed with reading. (b) Otherwise, the read operation blocks until either all update transactions finish (case as above) or until the commit timestamps of currently validating update transactions are larger than the start time of the read-only transaction (possibly conflicting transactions have finished execution).

Next steps

The above outline of a possible concurrency control approach for the cell model needs to be refined. Thorough treatment of possible failure situations and integration with the presented commit protocol are required next steps.

An analysis of the communication overhead of the different techniques would allow us to select an optimal approach.

3.5 Usage Scenario: Distributed Wiki

In this section we show how the transaction model fits for the usage scenario which is part of deliverable 5.1. Instead of running Wiki on several racks full of servers, it could be run on a decentralized storage service as it was introduced before. Figure 6 illustrates such a distributed Wiki. The nodes of the structured overlay network store the content of the Wiki as items where the key is the name of the page and the

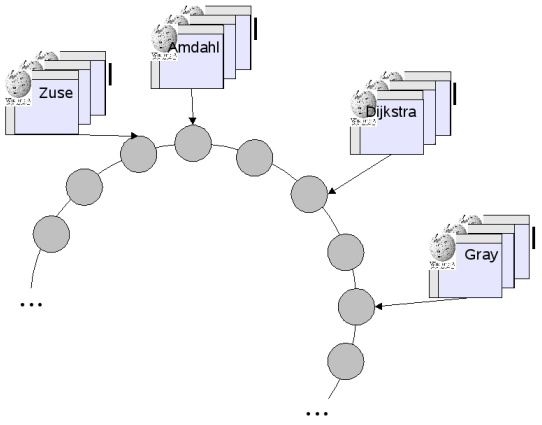


Figure 6: Distributed Wiki using a decentralized storage service built on top of a SON

value is the content of the page. For simplicity Figure 6 does not include hashing of the keys.

3.5.1 Transactions in a distributed Wiki

Here we illustrate a simple transaction sample for a distributed Wiki. It includes an update of the Wiki page on *Konrad Zuse* and an update of the category page for *Famous computer scientists*. The content of the category page should only be updated if both pages can be updated.

```
Begin of Transaction
    Read wiki page (Konrad Zuse)
    Update wiki page (Konrad Zuse)
    Update wiki category page (Famous computer scientists)
End of Transaction
```

When a user wants to change a wiki page, he first reads the page and then makes some changes to it. Between reading and submitting the changes, a different user could have made changes to the same page. The transaction has to check whether the read wiki page is still the current one and changes were based on the current version. Additionally in this example updates on the wiki page *Konrad Zuse* should only be made if changes in the wiki category page *Famous computer scientists* can be made at the same time.

4 Papers and publications

Atomic Commitment in Transactional DHTs M. Moser, S. Haridi. Atomic Commitment in Transactional DHTs. In *proceedings of the CoreGRID Symposium* (to appear).

Abstract: We investigate the problem of atomic commit in transactional database systems built on top of Distributed Hash Tables. Therefore we present a framework for DHTs to provide strong data consistency and transactions on data stored in a decentralized way. To solve the atomic commit problem within distributed transactions, 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 in contrast to applying 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, what is necessary for systems running over the Internet.

Distributed Wikis on Structured Overlays S. Plantikow, A. Reinefeld, and F. Schintke. Distributed Wikis on Structured Overlays. Presented in *CoreGRID Workshop on Grid Programming Model, Grid and P2P System Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, June 2007.

Abstract: We present a transaction processing scheme for structured overlay networks and use it to develop a distributed Wiki application that is based on a relational data model. The Wiki supports rich metadata and additional indexes for navigation purposes. Ensuring consistency and durability requires handling of node failures. We mask such failures by providing high availability of nodes by constructing the overlay from replicated state machines (Cell model). Atomicity is realized using two phase commit with additional support for failure detection and restoration of the transaction manager. The developed transaction processing scheme provides the application with a mixture of pessimistic, hybrid optimistic and multiversioning concurrency control techniques to minimize the impact of replication on latency and optimize for read operations. We present pseudocode of the relevant Wiki functions and evaluate the different concurrency control techniques in terms of message complexity.

References

[1] A. Ghodsi, L.O. 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.

[2] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[4] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[5] S. Haridi M. Moser. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[6] F. Schintke S. Plantikow, A. Reinefeld. Distributed wikis on structured overlays. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P System Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, June 2007.

[7] André Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, 2006.

[8] Fred B. Schneider. The state machine approach: A tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986.

[9] Alexander Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, January/February 1998.

A Atomic Commitment in transactional DHTs

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. Therefore we present a framework for DHTs to provide strong data consistency and transactions on data stored in a decentralized way. To solve the atomic commit problem within distributed transactions, 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 in contrast to applying 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, what is necessary for systems running over the Internet.

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

*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.

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 distributed data are done in a transactional manner. For distributed transactions an atomic commit protocol is needed to guarantee that either all operations of the transaction take place or none of them. Only committed states are made visible. Another important mechanism of distributed transactional systems is concurrency control, which ensures that concurrent transactions cannot interfere with each other. We present a framework for having transactions on DHTs and consequently strong notion of data consistency in DHTs. Our focus in this paper is on 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 a transaction is marked with 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 nodes, which are responsible for items that are involved in the transaction, can execute the operations. If all those nodes confirm that they can do so, 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

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 DKS [3], Chord [1]and CAN [9].

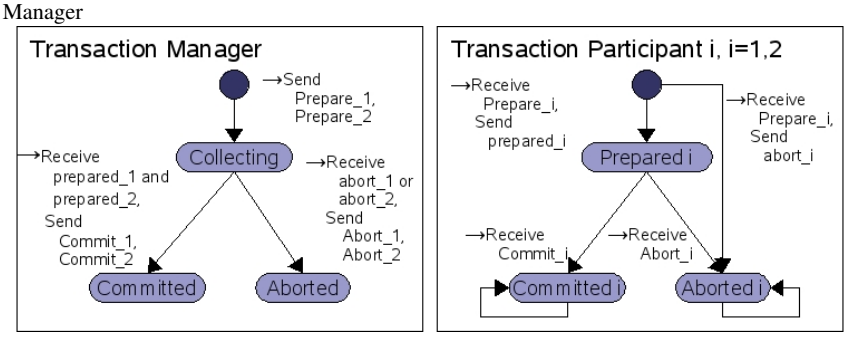
There exist a number of storage systems which are built on DHTs, e.g. Bamboo[1] which is based on Pastry and DHash[2] which is based on Chord. Mostly items in such systems are replicated for a higher degree of availability and reliability. These systems are typically read-only storage systems.

Atomicity is one of the four ACID properties of a 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 modifications 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 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 prepared and were able to commit. In the second phase (decision phase) the TM tells the TPs to commit if all TPs are prepared and are able to 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), the TPs are not able to receive the outcome of the transaction and are blocked. A number of non-blocking algorithms were introduced. *Three-Phase-Commit (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

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

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



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. The identifiers for 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 these operations. All operations related to data enforce the invariant that a majority of replicas for a certain data item is up to date. A majority contains at least $\lfloor \frac{f}{2} \rfloor + 1$ replicas. 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 exceeds f. When a new node joins the system, it gets the data it will be responsible for, and then takes over the responsibility from the node formerly responsible for those items. 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 for this node's items. 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 [11], 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. There are even two levels of distribution. Additionaly distributed items are replicated and again spread over the whole system. The number of nodes involved in a

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. This 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 will contain the result of the transaction 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 nodes has to decide on a common value. Uniform consensus satisfies the following properties: 1. *Uniform agreement*, which means that no two nodes 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 node; 3. *Integrity*, meaning no node may decide twice and finally 4. *Termination*, every node 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 nodes. 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 node may act as both proposer and acceptor. In our solution presented below we use different nodes 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 node 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 node tries to impose the value resulting from the read phase on a majority of nodes. 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 node proposes abort or is perceived to have crashed by other nodes before a decision was taken, then all nodes have to decide on abort. To decide on commit, all nodes have to propose prepared.

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 has collected 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, by using symmetric replication. 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. It has to be noted that a node could be responsible for several items involved in the transaction. The TP runs a separate Paxos instance for each item it is responsible for.

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 TPs holding a

replica 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.

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 [10] 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 [12] 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 [12] 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 nodes 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 nodes 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.

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] 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

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

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

B Distributed Wikis on Structured Overlays

DISTRIBUTED WIKIS ON STRUCTURED OVERLAYS*

Stefan Plantikow
Zuse Institute Berlin

Alexander Reinefeld
Zuse Insitute Berlin

Florian Schintke
Zuse Insitute Berlin

Abstract We present a transaction processing scheme for structured overlay networks and use it to develop a distributed Wiki application that is based on a relational data model. The Wiki supports rich metadata and additional indexes for navigation purposes.

Ensuring consistency and durability requires handling of node failures. We mask such failures by providing high availability of nodes by constructing the overlay from replicated state machines (Cell Model). Atomicity is realized using two phase commit with additional support for failure detection and restoration of the transaction manager. The developed transaction processing schema provides the application with a mixture of pessimistic, hybrid optimistic and multiversioning concurrency control techniques to minimize the impact of replication on latency and optimize for read operations. We present pseudocode of the relevant Wiki functions and evaluate the different concurrency control techniques in terms of message complexity.

Keywords: Distributed transactions, content management systems, structured overlay networks, consistency, concurrency control.

1. Introduction

Structured overlay networks provide a scalable and efficient means for storing and retrieving data in distributed environments without central control. Un-

*This work was supported by the EU Network of Excellence Core-GRID and the EU SELFMAN project

fortunately, in their most basic implementation, structured overlays do not provide any guarantees on the ordering of concurrently executed operations.

Transaction processing provides concurrently executing clients with a single, consistent view of a shared database. This is done by bundling client operations together in a transaction and executing them as if there was a global, serial transaction execution order. Enabling structured overlays to provide transaction processing support is a sensible next step for building *consistent* decentralized, self-managing storage virtualization services.

We propose a transactional system for an Internet-distributed content management system built on a structured overlay. Our emphasis is on supporting transactions in dynamic decentralized systems where nodes may fail with a relatively high rate. The chosen approach provides clients with different concurrency control options to minimize latency.

The article is structured as follows: Section 2 describes a general model for distributed transaction processing in structured overlay networks. The main problem addressed is handling the unreliability of nodes. Section 3 presents our transaction processing schema with a focus on concurrency control. This schema is extended to the relational model and exemplified using the distributed Wiki in Section 4. Finally, in Section 5, we evaluate the different proposed transaction processing techniques in terms of message complexity.

2. Transactions on Structured Overlays

Transaction processing is used to guarantee the four ACID properties: Atomicity (transactions are either executed completely or aborted and any effects undone), consistency (transaction processing will never corrupt the database state), isolation (data operations of concurrently executing transactions do not interfere with each other), durability (results of successfull transactions survive system crashes). These ACID properties can be separated into two aspects: *Concurrency control* is responsible for isolation and consistency by proper scheduling of elementary operations, and database *recovery* ensures atomicity and durability of transactions.

Page model. In this paper we consider transactions in the *page model* [4] in which a database contains a set of uniquely addressable, single objects. Valid elementary operations are reading and writing of objects and transaction commit and abort. The model does not support predicate locking and thus phantoms can occur and our scheme cannot support consistent aggregation queries. The page model was chosen because it can be naturally applied to structured overlays. Objects are stored by their identifier using the overlay's policy for data placement. In Section 4.1 we show how relational data models can be mapped on top of this simple scheme.

2.1 Distributed Transaction Processing

Distributed transaction processing guarantees the ACID-properties in scenarios where clients access multiple databases or different parts of the same database located on different nodes. All accesses to local databases are controlled by *resource manager (RM)* processes in each participating node. Additionally, for each active transaction one node takes the role of the *transaction manager (TM)*. The TMs coordinate with the involved RMs to execute transactions on behalf of their clients. The TMs also plays an important role during the execution of distributed atomic commit protocols.

Distributed transaction processing in a structured overlay network requires to distribute resource- and transaction management. Transaction management can be performed by the initiating peer. For resource management it is necessary to minimize the required communication overhead between resource manager and the storing node. Therefore, in the following, we assume that each peer of the overlay performs resource management for all objects in its fraction of the keyspace. For application scenarios where certain groups of objects are accessed together, it could be preferable to perform resource management at a dedicated peer for the whole group.

2.2 The Cell Model for Handling Churn

Distributing resource management over all peers puts tight restrictions on the message delivery. Messages initiating operations under transaction control must never be delivered to the wrong node. This property is known as *lookup consistency*. Without lookup consistency, a node might erroneously grant a lock on a data item or deliver outdated data. It is an open question how lookup consistency can be efficiently guaranteed in the presence of frequent and unexpected node failures (churn). Some authors [3, 6] have proposed protocols based on atomic commit that ensure consistent lookup if properly executed by all joining and leaving nodes. Yet large scale overlays are subject to considerable amounts of churn [8]. Thus handling the unreliability of nodes is important for any transaction processing scheme.

Relational databases usually assume the *crash-recovery* model in which durability is guaranteed by a combination of persistent storage and certain restart mechanisms. For structured overlays, the crash-recovery model is not useful because it is often unknown whether a disconnected node will later rejoin again. As a consequence, traditional locking cannot be used, because unreleased locks of crashed nodes would block the system forever. Hence, for structured overlays, the *crash-stop* model is used instead. Here the positive dynamics of structured overlays (neighboring nodes take over the keyspace partition of a failed node) conflicts with transactional consistency.

Cell model. Irrespective of the chosen failure model, data loss created by terminal node failures will violate the durability property. Therefore we propose the use of *replicated state machines* (RSMs) [16] to ensure (a) lookup consistency (b) availability and (c) durability. Instead of constructing the overlay network from single nodes, the overlay is made up by *cells*. Each cell is a dynamically sized group of physical nodes [15] that constitutes a RSM. Performing replication below the overlay's topology yields the advantage of reduced communication costs. No overlay lookups are necessary to send messages between replicas.

The execution of replicated operations has considerable cost: Even modern consensus algorithms like Fast Paxos [7] require at least $N(\lfloor 2N/3 \rfloor + 1)$ messages. While this cost is hardly avoidable for consistent replication, it is also unacceptable for regular message routing. Routing using dirty reads avoids these costs but may create routing errors if node and cell state are temporarily deviating. To handle this, the presumed target cell will deliver the message using a replicated operation (Fig. 1). If during the delivery attempt it is detected that the cell is not responsible for the message, routing continues using the cell's proper routing table.

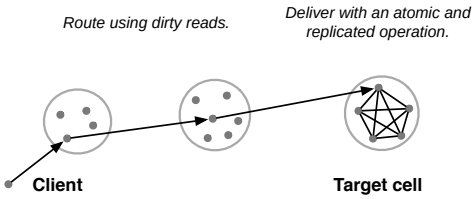


Figure 1. Cell routing using dirty reads.

We do not cover the distribution of physical nodes on cells, nor do we consider Byzantine failures. For this paper, we assume that cells either have enough nodes or are merged with topologically adjacent cells. In any case cells never fail unexpectedly and always orderly execute the overlay algorithm. If too many nodes of a cell fail, the cell destroys itself by executing the overlay's leave protocol. The freed nodes can then rejoin neighbouring cells. This has the benefit that crash-recovery of failed nodes and the use of stable storage is unnecessary. For simplification, we also assume that the keyspace partition associated to each cell does not change during transaction execution.

3. Concurrency Control and Atomic Commit in Structured Overlays

We use hybrid optimistic concurrency control and two phase commit on top of replicated state machines (cells). Additionally we support optimized read transactions using read-only multiversioning.

Atomic Operations. Using RSMs directly allows the execution of atomic and totally ordered operations. This already suffices to implement transaction processing, e.g. by using pessimistic, strong *two phase locking (2PL)* and an additional distributed atomic commit protocol. But each replicated operation is expensive. Thus any efficient transaction processing scheme for cell-structured overlays must aim at minimizing the number of replicated operations.

Optimistic concurrency control (OCC). OCC executes transactions against a *local* working copy (working phase). This copy is validated just before the transaction is committed (validation phase). The transaction is aborted if conflicts are detected during validation. As every node has (a possibly temporarily deviating) local copy of its cell's shared state, OCC is a prime candidate for reducing the number of replicated operations by executing the transaction against single nodes of each involved cell.

3.1 Hybrid Optimistic Concurrency Control

Plain OCC has the drawback that long-running transactions which need objects that are frequently accessed by short-running transactions may suffer starvation due to consecutive validation failures. This is addressed by *hybrid optimistic concurrency control (HOCC, [18])* under the assumption of *access invariance*, i.e. repeated executions of the same transaction have identical read and write sets.

HOCC works by executing strong 2PL for the transaction's read and write sets at the beginning of the validation phase. In case of a validation failure, the locks are kept and the transaction logic is reexecuted. Because of access invariance this second execution cannot fail. All necessary locks are already held by the transaction.

The use of strong 2PL has the additional benefit that no distributed deadlock detection is necessary if a global validation order between transactions with non-disjoint sets of accessed objects can be established. A possible technique for this has been described by Agrawal et. al [1]: Every cell v maintains a strictly monotonic increasing timestamp t_v for the largest, validated transaction. Before the start of validation, the transaction manager suggests a validation time stamp $t > t_v$ to all involved cells v. After every cell v has acknowl-

edged that $t > t_v$ and updated t_v to t, the validation phase is started. Otherwise the algorithm is repeated. Gruber [5] optimized this approach by including the largest validation timestamp in every message.

3.2 Distributed Atomic Commit

Distributed atomic commit (DBAC) requires consensus between all transaction participants on the transaction's termination state (committed or aborted). If DBAC is not guaranteed, all four ACID properties are violated.

We propose a blocking DBAC protocol that uses cells to treat TM failures by replicating transaction termination state.[1] A *commit record* holding the state is stored under the transaction's unique identifier (TXID) in the overlay network (for example in the same cell as the transaction manager's node). If no failures occur, regular two-phase atomic commit (2PC) is executed. But after prepared-messages have been received from and before the final commit messages are sent, the TM first writes the commit record. If the record already is set to abort, the TM aborts the transaction. If RMs suspect a TM failure, they read the commit record to either determine the termination state or initiate transaction abort.

3.3 Read-only Transactions

In many application scenarios simple read-only transactions are much more common than update transactions. Therefore we optimize and extend our transaction processing scheme for read-only transactions by applying techniques similar to read-only multiversioning (ROMV) [11].

All data items are versioned using unique timestamps generated from each node's loosely synchronized clock and globally unique identifier. Additionally for each data item we maintain a *current version*. This version is accessed and locked exclusively by HOCC transactions as described above and implicitly associated with the cell's maximum validation timestamp t_v. The current version decouples read-only multiversioning and HOCC.

Our approach moves newly created versions to the future such that they never interfere with read operations from ongoing read-only transactions. This avoids the cost associated with distributed atomic commit for read-only transactions but necessitates it to execute reads as replicated operations. Read-only transactions are associated with their start time. Every read operation is executed as a replicated operation using the multiversioning rule [14]: The result is the oldest version that is younger than the transaction start time. If this version is the current version, the maximum validation timestamp t_v is updated. This may block the read operation until a currently running validation is fin-

[1] For an alternative, non-blocking approach, see [12].

ished. Update transactions create new versions of all written objects using $t > t_v$ during atomic commit.

4. Algorithms for a Distributed Wiki

In the following sections we describe the basic algorithms of a distributed content management system that is built on a structured overlay with transaction support.

4.1 Mapping the Relational Model

So far we only considered uniquely addressable, uniform objects. In practice, many applications use more complex, relational data structures. This rises the question of how multiple relations with possibly multiple attributes can be stored in a single structured overlay. For this, we assume that the overlay supports range queries over a finite number of index dimensions.[2]

Storing multiple attributes requires mapping them on index dimensions. As the number of available dimensions is limited, it is necessary to partition the attributes into disjoint groups and map these groups instead. The partition must be chosen in such a way that fast primary-key based access is still possible. Depending on their group membership, attributes are either primary, index, or data attributes. Multiple relations can be modeled by introducing an additional primary attribute that contains a unique relation identifier.

4.2 Notation

Table 1 contains an overview of the pseudocode syntax from [13]. Relations are represented as sets of tuples and written in CAPITALS. Relation tuples are addressed using values for the primary attributes in the fixed order given by the relation. For reasons of readability, tuple components are identified using unique labels (Such labels easily can be converted to positional indexes). Range queries are expressed using labels and marked with a "?".

4.3 Wiki

A *Wiki* is a content management system that embraces the principle of minimizing access barriers for non-expert users. Wikis like www.wikipedia.org comprise millions of pages that are written in a simplified, human-readable markup syntax. Each page has a unique name which is used for hyperlinking with other Wiki pages. All pages can be read and edited by any user, which may result in many concurrent modification requests for hotspot pages. This makes Wikis a perfect test-case for our distributed transaction algorithm.

[2] Possible approaches can be found in [17, 2].

Table 1. Pseudocode notation

Syntax	Description
Procedure Proc (arg_1, arg_2, ..., arg_n)	Procedure declaration
Function Fun (arg_1, $arg_2 \stackrel{def}{=}$ "Value", ..., arg_n)	Function declaration, default for arg_2
begin transaction...**commit (abort) transaction**	Transaction boundaries
ADDRESS$_{"ZIB"}$	Read tuple from relation
ADDRESS$_{"ZIB"} \leftarrow$ ("*Takustr.* 7","*Berlin*")	Write tuple to relation
$\Pi_{attr_1, ..., attr_n}(M) = \{\pi_{attr_1, ..., attr_n}(t) \mid t \in M\}$	Projection
$\forall t \in$ tuple set : RELATION $\stackrel{+}{\leftarrow} t$ bzw. $\stackrel{-}{\leftarrow} t$	Bulk insert and delete
DHT$^{?}_{key_1="a",\ key_2}$ *or* DHT$^{?}_{key_1="a",\ key_2=*}$	Range query ($*$ asks for any value)
ADDRESS$^{?}_{"ZI"<orga<"ZZ"\#<50}$ $^{\overleftarrow{orga},\ \overrightarrow{street}}$	Sorted range query with result limit

Modern Wikis extend provide a host of additional features, particularly to simplify navigation. In this paper we exemplarily consider backlinks (list of other pages linking to this page) and recent changes (list of recent modifications of this pages). We model our Wiki using the following two relations:

Relation	Primary attributes	Index attributes	Data attributes
CONTENT	*pageName*	*ctime* (change time)	*content*
BACKLINKS	*referencing* (page), *referenced* (*page*)	-	-

All Wiki operations use transactions to maintain the following consistency invariants:

- CONTENT always contains the page's current content,
- BACKLINKS contains proper backlinks for all pages given by CONTENT,
- users cannot modify pages whose content has never been seen by them (explained below).

The function WikiRead (Alg. 4.1) delivers the content of a page and all backlinks pointing to it. This requires a single read for the content and a range query to obtain the backlinks. Both operations can be executed in parallel.

The function RecentChanges (Alg. 4.2) issues a range query to return a sorted list of the *limit* newest pages that have been changed $beforeTime$.

The function WikiWrite (Alg. 4.3) is more complex because conflicting writes by multiple users must be resolved. This can be done by serializing the write requests using locks or request queues. If conflicts are detected during (atomic) writes by comparing last read and current content, the write operation is aborted.

Algorithm 4.1 WikiRead: Read page content

```
1: function WikiRead (pageName)
2:     begin transaction read-only
3:         content ← π_content(CONTENT_pageName)
4:         backlinks ← Π_referenced(BACKLINKS?_referencing=pageName, referenced)
5:     commit transaction
6:     return content, backlinks
7: end function
```

Algorithm 4.2 RecentChanges: List of recently modified pages

```
1: function RecentChanges (beforeTime, limit)
2:     begin transaction read-only
3:         result ← {CONTENT?_pageName, ctime>beforeTime}^ctime_#<limit
4:     commit transaction
5:     return result
6: end function
```

Users may then manually merge their changes and retry. This approach is similar to the compare-and-swap instructions used in modern microprocessors and to the concurrency control in version control systems.[3] For our distributed Wiki, we realize the compare-and-swap in WikiWrite by using transactions. First, we precompute which backlinks should be inserted and deleted. Then, we compare the current and old page content and abort if they differ. Otherwise all updates are performed by writing the new page content and modifying BACKLINKS. The update operations again can be performed in parallel.

4.4 Wiki with Metadata

Often it is necessary to store additional metadata with each page (e.g. page author, category). To support this, we add a third relation METADATA with primary key attributes *pageName* and *attrName* and data attribute *attrValue*. Alternatively we could also add metadata attributes to CONTENT. But this would not be scalable as current overlays only provide a limited number of index dimensions.

Modifying page metadata requires checking that the page has not been changed by some other transaction. Otherwise new metadata could be associated wrongly to a page (This is similiar to storing the wrong backlinks). For reading page metadata, a simple range query suffices ([13] contains the algorithms).

[3] Most version control systems provide heuristics (e.g. content merging) for automatic conflict resolution that could be used for the Wiki as well.

Algorithm 4.3 WikiWrite: Write new page content and update backlinks

```
1:  procedure WikiWrite(pageName, content_old, content_new)
2:      refs_old ← Refs(content_old)
3:      refs_new ← Refs(content_new)
4:      refs_del ← refs_old \ refs_new          — precalculation
5:      refs_add ← refs_new \ refs_old
6:      txStartTime ← CurrentTimeUTC()
7:      begin transaction
8:          if π_content(CONTENT_pageName) = content_old then
9:              CONTENT_pageName = (txStartTime, content_new)
10:             ∀t ∈ {(ref, pageName) | ref ∈ refs_add} : BACKLINKS ←(+) t
11:             ∀t ∈ {(ref, pageName) | ref ∈ refs_del} : BACKLINKS ←(−) t
12:         else
13:             abort transaction
14:         end if
15:     commit transaction
16: end procedure
```

5. Evaluation

It is noteworthy that the presented algorithms for ensuring consistency mainly require the atomicity property. There are only few conditions on the serial execution order of operations. Thus in theory, a high degree of concurrency is possible. This is especially interesting for range queries like RecentChanges which can utilize the overlay's capabilities to multicast to many nodes in parallel.

Table 2. Comparison of concurrency control methods

Transaction type	**Once for N involved cells**	**Parallel ops on N cells**	**Total for k serial ops**
(1) Atomic Write	$1L$	$1R$	$1L+1R$, because $k,N=1$
(2) Read-Only Trans.	NL	NR	$NL+kNR$
(3) Pess. 2PL + 2PC	$NL+2NR$	NR	$NL+(k+1)NR$
(4) Hyb. Opt. + 2PC	$NL+2NR$	NU	$NL+(k-1)NU+2NR$
(5) Hyb. Opt. + 2PC + Validation Error	$NL+3NR$	$2NU$	$NL+(2k-2)NU+3NR$

Table 2 compares the communication overhead of the different concurrency control methods. We assume transactions consisting of k serial operations. Every such operation is executed in parallel on N cells. U is a simple, unreplicated, R is a replicated, and L is a lookup (routing) operation. The cost

is split into one-time (initial and DBAC) overhead, overhead per k operations, and total overhead. Totals include DBAC costs and respect possible combined sending of messages (e.g. combining last data operation with validate and prepare).

Table 2 contains (1) a simple, replicated operation on a single cell, (2) a read-only multiversioning transaction (Sec. 3.3), (3) a pessimistic 2PL transaction, (4) a HOCC (Sec. 3.1) transaction without validation failure, and (5) a HOCC transaction with validation failure and transaction logic reexecution. (2)-(4) all use the 2PC variant described in 3.2 (For the evaluation, we assume no failures occur during commit).

HOCC reduces the number of necessary replicated operations for $k > 1$. For $k = 1$ and a operation on a single cell, ACID is already provided by using a RSM and no DBAC is necessary. For $k = 1$ and a single operation over multiple cells, HOCC degenerates into 2PL: the data operations on the different cells are combined with validate-and-prepare messages and executed as single replicated operations.

Read-only transactions use more replicated operations but save the DBAC costs of HOCC. This makes them well-suited for quick, parallel reads. But long running read transactions might be better off by choosing HOCC if the performance gained by optimism outweights DBAC overhead and vaidation failure chance.

Using cells yields an additional benefit. If replication would be performed above the overlay layer, additional routing costs of $(r-1)N$ lookup messages would be necessary (r is the number of replicas).

6. Summary

In this article, we presented a transaction processing scheme suitable for a distributed Wiki application on a structured overlay network. While previous work on overlay transactions (e.g. [10]) has not treated handling the unreliability of nodes, we identified this as a key requirement for consistent data storage in structured overlays and proposed the cell model as a possible solution.

The developed transaction processing scheme provides applications with a mixture of concurrency control techniques to minimize the required communication effort. We showed core algorithms for the Wiki that utilize overlay transaction handling support and evaluated the different concurrency control techniques in terms of message complexity.

References

[1] A. Divyakant, A.J. Bernstein, P. Gupta, and S. Soumitra. Distributed optimistic concurrency control with reduced rollback. In: *Distributed Computing* (2), pages 45–59, 1987.

[2] A. Andrzejak, Z. Xu, Scalable, Efficient Range Queries for Grid Information Systems. In: *2nd IEEE International Conference on Peer-to-Peer Computing (P2P2002)*, pages 5–7, Sweden, Sep. 2002

[3] A. Ghodsi. Distributed k-Ary System: Algorithms for Distributed Hash Tables. PhD thesis, KTH Stockholm, Stockholm, 2006.

[4] J. Gray. The Transaction Concept: Virtues and Limitations. In: *Proceedings of the 7th International Conference on Very Large Databases*, pages 144–154, 1981.

[5] R.E. Gruber. Optimistic Concurrency Control for Nested Distributed Transactions. *Technical Report MIT-LCS/TR-453*, Laboratory of Computer Science, Massachusetts Institute of Technology, Jun. 1989.

[6] S.E. Johnson. Consistent lookup during Churn in Distributed Hash Tables. Master thesis, Norwegian University of Science and Technology, Trondheim, Sep. 2005.

[7] L. Lamport. Fast Paxos. *Technical Report MSR-TR-2005-112*, Microsoft Research, 2nd edition, Jan. 2006.

[8] J. Li, J. Stribling, T. M. Gil, R. Morris, and M.F. Kaashoek. Comparing the performance of distributed hash tables under churn. *IPTPS 2004*, Feb. 2004.

[9] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data. *Technical Report MIT-CSAIL-TR-2005-044*, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2005.

[10] V. Mesaros, R. Collet, K. Glynn, and P. van Roy. A Transactional System for Structured Overlay Networks. *Technical Report RR2005-01*, Department of Computing Science and Engineering, Université catholique de Louvain, Mar. 2005.

[11] C. Mohan, H. Pirahesh, R. Lorie. Efficient and flexible methods for transient versioning of record to avoid locking by read-only transactions. In: SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data., pages 124–133, 1992.

[12] M. Moser, and S. Haridi. Atomic Commitment in Transactional DHTs, *First CoreGRID European Network of Excellence Symposium*, Aug. 2007. To appear.

[13] S. Plantikow. Transaktionen für veteilter Wikis auf strukturierten Overlay-Netzwerken. Diploma thesis, Humboldt-Universität zu Berlin, Apr. 2007.

[14] D. Reed. Naming and Synchronization in a Decentralized Computer System. PhD thesis, In: *Technical Rerpot MIT-LCS/TR-205*., Laboratory of Computer Science, Massachusetts Institute of Technology, Sep. 1978.

[15] A. Schiper. Dynamic group communication. In: *Distributed Computing* 18 (5), pages 359–374, 2006.

[16] F.B. Schneider. The State Machine Approach: A Tutorial. *Technical Report TR-86-800*, Department of Computer Science, Cornell University, 1986.

[17] T. Schütt, F. Schintke, A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06)*, May 2006.

[18] A. Thomasian. Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing. In: *IEEE Transactions on Knowledge and Data Engineering* 10 (1), pages 173–189, Feb. 1998.

[19] G. Urdaneta, G. Pierre, and M. van Steen. A Decentralized Wiki Engine for Collaborative Wikipedia Hosting. In: *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, Mar. 2007.