Transactions for Distributed Wikis on Structured Overlays*

Stefan Plantikow, Alexander Reinefeld, and Florian Schintke

Zuse Institute Berlin, Takustrasse 7, D-14195 Berlin-Dahlem

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.

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. Unfortunately, in their most basic implementation, structured overlays do not offer 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

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

transactions in dynamic decentralized systems where nodes may fail at 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 masking the unreliability of nodes. Section 3 presents our transaction processing scheme focusing on concurrency control. This scheme 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 successful 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 has been 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. Access to local databases is 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 play an important role during the execution of distributed atomic commit protocols.

Distributed transaction processing in a structured overlay network requires distribution of 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 keyspace partition.

2.2 The Cell Model for Masking Churn

Distributing the resource management over all peers puts tight restrictions on messages delivered under transaction control. Such messages may only be delivered to nodes that are currently responsible for the data. 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, 7] have suggested protocols that ensure consistent lookup if properly executed by *all* joining and leaving nodes. Yet large scale overlays are subject to considerable amounts of churn [9] and therefore correct transaction processing requires masking it.

Cell model. Instead of constructing the overlay network using single nodes, we propose to build the overlay out of *cells*. Each cell is a dynamically sized group of physical nodes [17] that constitute a replicated state machine (RSM, [18]). Cells provide replicated, atomic operations and high availability. This can be utilized to

- mask churn and therefore guarantee lookup consistency,
- provide stable storage for transactional durability,
- ensure data consistency using atomic operations,
- minimize overhead for routing to replicas (cell nodes form a clique).

For the underlying nodes, we assume the *crash-stop* failure model. This model is common for structured overlay networks because it is usually unknown wether a disconnected node will rejoin again later. We do not cover the distribution of physical nodes on cells, nor do we consider Byzantine failures. We assume that cells never fail unexpectedly and always execute the overlay algorithm orderly. If too many cell nodes fail, the cell destroys itself by executing the overlay's leave protocol. The data items are distributed among neighboring cells. For simplification, we also assume that the keyspace partition associated to each cell does not change during transaction execution.

Cell routing. The execution of replicated operations within the cells comes at a considerable cost: Modern algorithms like Fast Paxos [8] require at least $N(\lfloor 2N/3 \rfloor + 1)$ messages for consensus between N nodes. While this cost is hardly avoidable for consistent replication, it is as well unacceptable for regular message routing. Hence we propose to use dirty reads (i.e. to read the state of one arbitrary node). When the state of a node and its cell are temporarily out of sync, routing errors may occur. To handle this, the presumed target cell will

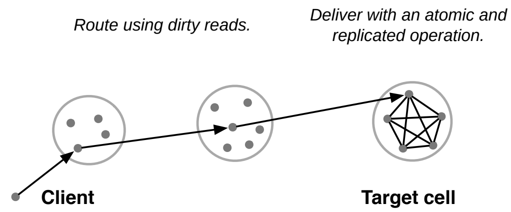

Fig. 1. Cell routing using dirty reads.

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.

3 Concurrency Control and Atomic Commit in Structured Overlays

In this section, we present appropriate concurrency control and atomic commit techniques for overlays based on the cell model.

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. However, each replicated operation is expensive, thus any efficient transaction processing scheme for cell-structured overlays should 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 using objects which are frequently accessed by short-running transactions may suffer starvation due to consecutive validation failures. This is addressed by *hybrid optimistic concurrency control (HOCC, [20])* 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 re-executed. Now, access invariance ensures that this second execution cannot fail. All necessary locks are already held by the transaction.

The use of strong 2PL adds the benefit that no distributed deadlock detection is necessary if a global validation order between transactions is established. A possible technique for this has been described by Agrawal et. al [1]: Each cell v maintains a strictly monotonic increasing timestamp t_v for the largest, validated transaction. Before starting the validation, the transaction manager suggests a validation time stamp $t > t_v$ to all involved cells. After each such cell has acknowledged 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, the ACID properties are violated.

We propose a blocking DBAC protocol that uses cells to treat TM failures by replicating transaction termination state. Every transaction is associated with a unique identifier (TXID). The overlay cell corresponding to that TXID is used to store a *commit record* holding the termination state and the address of the TM node (an arbitary, single node of the TXID cell). If no failures occur, regular two-phase atomic commit (2PC) is executed. Additionally, after all prepared-messages have been received and before the final commit messages are sent, the TM first writes the commit record. If the record is already set to abort, the TM aborts the transaction. If RMs suspect a TM failure, they read the record to either determine the termination state or initiate transaction abort. Optionally, RMs can restore the TM by selecting a new node and updating the record appropriately. Other RMs will find out about this when they reread the modified record.

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, [12]).

All data items are versioned using unique timestamps generated from each node's loosely synchronized clock and globally unique identifier. Additionally, we maintain a *current version* for each data item. 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 [15]: The result is the oldest version which is younger than the transaction start time. If this version is the current version, the maximum validation timestamp t_v will be updated. This may block the read operation until a currently running validation is finished. Update transactions create new versions of all written objects using $t > t_v$ during atomic commit.

3.4 Summary

The presented transaction processing scheme provides applications with a mixture of concurrency control techniques. Simple operations can be executed atomically by a cell, complex update transactions use hybrid optimistic concurrency control and 2PC with commit records. In addition, our scheme supports optimized read-only transactions using multiversioning that do not rely on DBAC.

4 Algorithms for a Distributed Wiki

In this section, we describe the basic algorithms of a distributed content management system 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 raises the question of how multiple relations with possibly multiple attributes can be stored in a single structured overlay. To address this, first, we assume that the overlay supports range queries [19, 2] over a finite number of index dimensions.

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

Table 1. Pseudocode notation

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

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 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.

Modern Wikis provide a host of additional features, particularly to simplify navigation. In this paper we exemplarily consider backlinks (a list of all the other pages linking to a page) and recent changes (a list of recent modifications of all Wiki 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 current content for all pages,
- Backlinks contains proper backlinks for all pages contained in Content,
- users cannot modify pages whose content they have never seen (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 $beforeTime$.

Algorithm 4.1 WikiRead: Read page content

function WikiRead (*pageName*)
  begin transaction *read-only*
    content $\leftarrow \pi_{content}(\text{CONTENT}_{pageName})$
    backlinks $\leftarrow \Pi_{referenced}(\text{BACKLINKS}^{?}_{referencing=pageName,\ referenced})$
  commit transaction
  return *content*, *backlinks*
end function

Algorithm 4.2 RecentChanges: List of recently modified pages

function RecentChanges (beforeTime, limit)
  begin transaction *read-only*
    result $\leftarrow \{\text{CONTENT}^{?}_{pageName,\ ctime>beforeTime}\}^{\overleftarrow{ctime}}_{\#<limit}$
  commit transaction
  return *result*
end function

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. Users may then manually merge their changes and retry. This approach is similar to the compare-and-swap instructions [6] used in modern microprocessors and to the concurrency control in version control systems.[1] We realize the compare-and-swap in WikiWrite by using transactions for our distributed Wiki. 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*.

[1] 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

procedure WikiWrite ($pageName,\ content_{old},\ content_{new}$)
    $refs_{old} \leftarrow$ Refs ($content_{old}$)
    $refs_{new} \leftarrow$ Refs ($content_{new}$)
    $refs_{del} \leftarrow refs_{old} \setminus refs_{new}$ — precalculation
    $refs_{add} \leftarrow refs_{new} \setminus refs_{old}$
    $txStartTime \leftarrow$ CurrentTimeUTC()
    begin transaction
        if $\pi_{content}(\textsc{Content}_{pageName}) = content_{old}$ **then**
            $\textsc{Content}_{pageName} = (txStartTime,\ content_{new})$
            $\forall t \in \{(ref,\ pageName) \mid ref \in refs_{add}\} :\ \textsc{Backlinks} \xleftarrow{+} t$
            $\forall t \in \{(ref,\ pageName) \mid ref \in refs_{del}\} :\ \textsc{Backlinks} \xleftarrow{-} t$
        else
            abort transaction
        end if
    commit transaction
end procedure

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 (Alg. 4.4) 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 ([14] contains the algorithm).

Algorithm 4.4 SetPageMetadata: Write page metadata attributes

Require: *changeEnv* environment describing changes to be made
procedure SetPageMetadata ($pageName,\ content_{old},\ changeEnv$)
    begin transaction
        if $\pi_{content}(\textsc{Content}_{pageName}) = content_{old}$ **then**
            $\forall(anAttrName \ \Leftarrow\ anAttrValue) \in changeEnv :$
                $\textsc{Metadata}_{pageName,\ anAttrName} \leftarrow anAttrValue$
        else
            abort transaction
        end if
    commit transaction
end procedure

5 Evaluation

The presented algorithms for ensuring consistency mainly require the atomicity property while only few restrictions are placed 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	$1\,L$	$1\,R$	$1\,L + 1\,R$, because $k, N = 1$
(2) Read-Only Trans.	$N\,L$	$N\,R$	$N\,L + kN\,R$
(3) Pess. 2PL + 2PC	$N\,L + 2N\,R$	$N\,R$	$N\,L + (k+1)N\,R$
(4) Hyb. Opt. + 2PC	$N\,L + 2N\,R$	$N\,U$	$N\,L + (k-1)N\,U + 2N\,R$
(5) Hyb. Opt. + 2PC + Validation Error	$N\,L + 3N\,R$	$2N\,U$	$N\,L + (2k-2)N\,U + 3N\,R$

Table 2 shows the communication overhead of various concurrency control methods. We assume transactions consisting of k serial operations. Each 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, the cost per k operations, and the total cost. Totals include DBAC costs and take the possible combined sending of messages into account (e.g. combining last data operation with validate and prepare).

Table 2 compares the following concurrency control schemes ((2) to (4) use the 2PC variant described in 3.2; for our evaluation, we assume that no failures occur during the commit):

(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 re-execution of transaction logic.

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 using HOCC if the performance gained by optimism outweights DBAC overhead and validation failure chance.

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

6 Related Work

OceanStore [16] uses a two-tier approach for multiversioning-based replication. On the first layer, a small set of replicas forms a primary ring. On the second layer, additional replicas cache object versions. Replicas are located using the Tapestry overlay network. Primary ring replicas use a Byzantine agreement protocol to serially execute atomic operations optionally guarded by a predicate.

Etna [10] is a system for executing atomic read and write operations in a Chord-like overlay network. Operations are serialized using a primary copy and replicated over k sucessors using a consensus algorithm.

Both articles do not describe how full transaction processing can be built on top of atomic operations. For OceanStore, multiversioning [15] is proposed [16]. However, the inherent cost of having to execute all operations as replicated operations is not considered.

Mesaros et. al describe a transaction processing scheme for overlays based on 2PL [11]. Lock conflicts are resolved by giving higher priority to older transactions and forcing the loosing transaction into the 2PL shrinking phase. Transactions are executed by forming a dynamic multicast group consisting of all participating nodes. The article does not address issues of lookup consistency and replication.

As an alternative to our solution for the atomic commit Moser et al. [13] describe a non-blocking approach based on Paxos commit.

7 Summary

We presented a transaction processing scheme suitable for a distributed Wiki application on a structured overlay network. While previous work on overlay transactions has not addressed node unreliability, we identified this as a key requirement for consistency 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 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. *Tech. Rep. MIT-LCS/TR-453*, Lab. of Comp. Sci., MIT, Jun. 1989.
6. M. Herlihy. Wait-Free Synchronization. In: *ACM Transactions on Programming Languages and Systems.*, 13 (1), pages 124–149, Jan. 1991.
7. S.E. Johnson. Consistent Lookup during Churn in Distributed Hash Tables. Master thesis, Norwegian University of Science and Technology, Trondheim, Sep. 2005.
8. L. Lamport. Fast Paxos. *Tech. Rep. MSR-TR-2005-112*, Microsoft Research, 2nd edition, Jan. 2006.
9. 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.
10. A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data. *Tech. Rep. MIT-CSAIL-TR-2005-044*, Comp. Sci. and AI Lab. (CSAIL), MIT, 2005.
11. V. Mesaros, R. Collet, K. Glynn, and P. van Roy. A Transactional System for Structured Overlay Networks. *Tech. Rep. RR2005-01*, Dept. of Comp. Sci. and Engineering, Université Catholique de Louvain, Mar. 2005.
12. C. Mohan, H. Pirahesh, R. Lorie. Efficient and Flexible Methods for Transient Versioning of Records 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.
13. M. Moser, and S. Haridi. Atomic Commitment in Transactional DHTs, *First CoreGRID European Network of Excellence Symposium*, Aug. 2007. To appear.
14. S. Plantikow. Transaktionen für veteilter Wikis auf strukturierten Overlay-Netzwerken. Diploma thesis, Humboldt-Universität zu Berlin, Apr. 2007.
15. D. Reed. Naming and Synchronization in a Decentralized Computer System. PhD thesis, In: *Tech. Rep. MIT-LCS/TR-205.*, Lab. of Comp. Sci., MIT, Sep. 1978.
16. S. Rhea, P. Eaton, D. Geels. Pond: The OceanStore Prototype. In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, 2003.
17. A. Schiper. Dynamic Group Communication. In: *Distributed Computing* 18 (5), pages 359–374, 2006.
18. F.B. Schneider. The State Machine Approach: A Tutorial. *Tech. Rep. TR-86-800*, Dept. of Comp. Sci., Cornell University, 1986.
19. 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.
20. 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.
21. 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.