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.

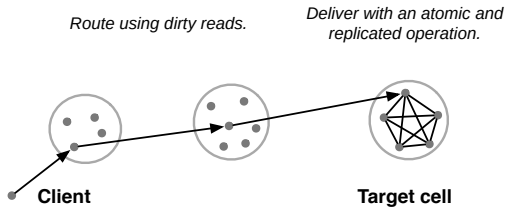

Figure 1. Cell routing using dirty reads.

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 $(arg_1, arg_2, \ldots, arg_n)$	Procedure declaration
Function Fun $(arg_1, arg_2 \stackrel{def}{=}$ "Value", $\ldots, arg_n)$	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
$\Pi_{attr_1, \ldots, attr_n}(M) = \{\pi_{attr_1, \ldots, attr_n}(t) \mid t \in M\}$	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 outweighs 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.