Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

MONIKA MOSER, SEIF HARIDI, TALLAT M. SHAFAAT,
THORSTEN SCHÜTT, MIKAEL HÖGQVIST, ALEXANDER
REINEFELD

Transactional DHT Algorithms

This work is partially funded by the European Commission through the SELFMAN project with contract number 034084.

ZIB-Report 09-34 (October 2009)

Transactional DHT Algorithms

Monika Moser[1], Seif Haridi[2], Tallat M. Shafaat[2], Thorsten Schütt[1], Mikael Högqvist[1], Alexander Reinefeld[1]

[1] Zuse Institute Berlin (ZIB), Germany
[2] Royal Institute of Technology (KTH), Sweden

Abstract. We present a framework for transactional data access on data stored in a DHT. It allows to atomically read and write items and to run distributed transactions consisting of a sequence of read and write operations on the items. Items are symmetrically replicated in order to achieve durability of data stored in the SON. To provide availability of items despite the unavailability of some replicas, operations on items are quorum-based. They make progress as long as a majority of replicas can be accessed. Our framework processes transactions optimistically with an atomic commit protocol that is based on Paxos atomic commit. We present algorithms for the whole framework with an event based notation. Additionally we discuss the problem of lookup inconsistencies and its implications on the one-copy serializability property of the transaction processing in our framework.

Table of Contents

1 Introduction

DHTs are fully decentralized and highly scalable systems that provide the ability to store and lookup data. They use the lookup service of a Structured Overlay Network for Internet-scale applications. The interface DHTs provide on their data is mostly a simple put/get interface. Often data in DHTs is immutable or consistency guarantees on data are weak. However many distributed systems require stronger guarantees like they are given by atomic data operations. We present a framework with transactional access to data stored in a DHT. It provides high availability of data and one-copy serializability for transactions on that data. A transaction consists of a sequence of one or more read and/or write operations that is executed atomically.

DHTs are dynamic systems where nodes are able to join the system or crash at any time. In order to maintain durability and availability of data, items are replicated. Each item consists of a fixed number of replicas. To tolerate the unavailability of a subset of replicas our transaction mechanisms are majority-based. This means that they are able to make progress if a majority of replicas is accessible. Therefore replication factor has to be chosen in a way that the availability of a majority of replicas is very high.

Read and write operations access at least a majority of replicas and choose the one with the highest timestamp. The atomic commit protocol that is needed to coordinate a distributed transaction also makes use of the majority idea. In order to prevent a single transaction manager from blocking the whole protocol if it fails, the framework uses a Paxos based non-blocking atomic commit protocol[7]. There, the single transaction manager is replaced by a set of nodes that all together act as the transaction manager. The protocol makes progress if the majority of these nodes does not fail until every participant in the transaction receives the outcome of the transaction.

In this paper we present the algorithms for our transactional framework. We use an event-based notation as it is well suited to present an asynchronous message-passing system. The framework builds on various techniques known from distributed database systems. The processing of transactions is done optimistically with a non-blocking atomic commit protocol in the end. Concurrency control is included in the atomic commit phase. The basic idea is to either acquire all necessary locks at the same time or to abort the transaction, thus avoiding distributed deadlock detection. Timestamps are used for each replica to determine whether items read in the read phase of the transaction are still valid in the commit phase. As the transaction processing is optimistic locks are only held during the commit phase. We combine the non-blocking Paxos atomic commit protocol with quorum techniques for access on replicated data.

2 Structured Overlay Network

In this section we introduce the model of the SON which underlies our framework. Thereby we refer to self-stabilization mechanisms that are used in Chord

[13]. However our framework is not restricted to a DHT based on Chord but can be applied to other key-based SONs.

2.1 The Overlay Model

A Model of a Ring-based SON. A DHT makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some a priori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has an unique identifier from the identifier space. Each node keeps two pointers: *succ*, to its *successor* on the ring, and *pred* to its *predecessor*. The successor of a node with identifier p is the first node found going in a clockwise direction on the ring starting at p. The predecessor of a node with identifier p is the first node met going anti-clockwise on the ring starting at p. For each node p, a *successor-list* is also maintained consisting of p's c immediate successors, where c is typically set to $\log_2(n)$, where n is the network size.

Ring-based DHTs also maintain additional routing pointers, called fingers, on top of the ring to enhance routing [1]. For our analysis, we assume that these pointers are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(\mathcal{N})$. Our results are independent of the chosen scheme for placing the fingers.

Dealing with Joins and Failures in Chord. A DHT system is a continuously running system and there is no notion of crash recovery. Whenever a node fails there is another node that becomes responsible for the items of the failed nodes. The protocols are based on a crash-stop model of nodes. This implies that if a node crashes and then reboot to re-join the network, it will be considered as a new node.

Chord handles joins and failures using a protocol called *periodic stabilization*. Figure 1 shows part of the protocol presented in [13]. Failures of predecessors are handled by having each node periodically check whether its *pred* is alive, and setting $pred := nil$ if the predecessor is found dead. Moreover, each node periodically checks to see if *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Each node periodically asks for its successor's *pred* pointer, and updates its *succ* pointer if it gets a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*.

Joins are also handled by the ring stabilization protocol. Joining nodes lookup their successor s on the ring, and sets $succ := s$. Periodic stabilization will eventually fix its predecessor and successor. Hence, any joining node is eventually properly incorporated into the ring.

Failure Detectors. DHTs provide a platform for Internet-scale systems, aimed at working on an asynchronous network. Informally, a network is asynchronous if there is no bound on message delay. Thus, no timing assumptions can be made

```
n.join(n')
  predecessor = nil;
  sucessor = n'.findsuccessor(n);

n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor])
    successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n])
    predecessor = n';

n.check predecessor()
  if (predecessor has failed)
    predecessor = nil;
```

Fig. 1: Part of the Chord protocol, presented in [13], which is related to successor and predecessor pointers.

in such a system. Due to the absence of timing restrictions in an asynchronous model, it is difficult to determine if a node has actually crashed or is very slow to respond. This gives rise to inaccurate suspicion of node failure.

Failure detectors are modules used by a node to determine if its neighbors are alive or dead. Since we are working in an asynchronous model, a failure detector can only provide probabilistic results about the failure of a node. Thus, we have failure detectors working probabilistically.

Failure detectors are defined based on two properties: *completeness* and *accuracy* [3]. In a crash-stop model, completeness requires the failure detector to eventually detect all crashed nodes. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not. A perfect failure detector is accurate all the times, while the accuracy of an unreliable failure detector is defined by its probability of working correctly.

2.2 Lookup Consistency and Responsibility

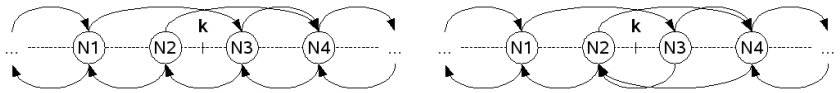
A consequence of imperfect failure detectors are inconsistent lookups and inconsistent responsibilities. We explain these terms in the following. Lookup consistency and responsibility consistency are important concepts when we reason about data consistency in our transactional DHT. Basically responsibility consistency is a requirement for guaranteeing data consistency.

In the following, we define lookup consistency and responsibility consistency, and explain how they can be violated. We use the term **configuration** of a SON to denote the set of all nodes and their pointers to neighboring nodes at a certain

point in time. A SON evolves by either changing a pointer, or adding/removing a node.

Definition 1 *A lookup for a key is consistent, if in a configuration lookups for this key made from different nodes, return the same node.*

Lookup consistency can be violated if some nodes' successor pointers do not reflect the current ring structure. Figure 2a illustrates a scenario, where lookups for key k can return inconsistent results. It shows nodes with their successor and predecessor pointers. This configuration may occur if node $N1$ falsely suspected $N2$ as failed, while at the same time $N2$ falsely suspected $N3$ as failed. A lookup for key k ending at $N2$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N1$ would return $N3$.



(a) **(b)**

Fig. 2: Lookup inconsistency and responsibility inconsistency. Nodes with successor and predecessor pointers: (a) Example with wrong successor pointers. (b) Example with wrong successor pointers and overlapping responsibilities.

Definition 2 *A node n is said to be* locally responsible *for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. We call a node* globally responsible *for a key, if it is the only node in the system that is* locally responsible *for it.*

The responsibility of a node changes whenever its predecessor is changed. If a node has an incorrect predecessor pointer, the range of keys it is responsible for can overlap with another node's key range. Thus there are several nodes responsible for a part of the key range.

Definition 3 *The responsibility for a key k is consistent if there is a node globally responsible for k.*

A configuration where responsibility consistency for key k is violated is shown in Figure 2b. Here, lookup consistency for k cannot be guaranteed and both nodes, $N3$ and $N4$, are locally responsible for k. However, in Figure 2a, $N3$ is globally responsible despite lookup inconsistency and $N4$ is not responsible. The configuration depicted in Figure 2b can arise from the configuration shown in Figure 2a with an additional wrong suspicion of node $N4$ about its predecessor $N3$.

Lookup consistency and responsibility consistency cannot be guaranteed in a SON. As we will show later responsibility consistency is an assumption of our

system in order to guarantee data consistency. However in [12] we show that the probability for a violation of responsibility consistency is very low. E.g. with a reasonable probability for a failure detector to make false positives with two percent the probability to get consistent responsibility for a replica is more than 99.999%.

2.3 Availability

Another important property in our system is the availability of a key. In order to make progress operations in our system have to be able to access a sufficient number of replicas.

Definition 4 *A key k is available if there exists a reachable node n such that n is locally responsible for k.*

Availability of a key in a SON is both affected by churn and inaccurate failure detectors. Due to churn a key is unavailable when the node that is responsible for it fails until a successor node takes over responsibility and is reachable in the system. This is illustrated in Figure 3b where the key k is unavailable because of the failure of node $N2$. A node n is said to be reachable for a node n', if there exists a path from n to n'. Also during a join process when a node is transferring responsibility for a certain key range to the joining node, keys in that range are unavailable until the joining node is reachable in the system. Figure 3c illustrates a scenario where the joining node $N2$ already took over responsibility for key k but is not yet reachable in the system as node $N1$ has not set its successor pointer to $N2$. In the second case inaccurate failure detectors cause unavailability when a node that falsely suspects its successor will remove the pointer to this node. Thus, keys for which the suspected node is responsible will temporarily become unavailable. In figure 3d node $N1$ suspects node $N2$, thus k becomes unavailable.

3 Transactions on a SON

Usually DHTs provide a simple put/get interface to store and retrieve data. Hardly they provide consistency guarantees on data and often they are restricted to immutable data. Our framework is able to provide a transactional interface on top of a SON. It provides read and write operations that are executed transactionally as well as the ability to execute transactions that consist of a sequence of different operations.

3.1 1-Copy Serializability

Our algorithms provide 1-copy serializability. In our system items are replicated and there might exist replicas with different versions. However transactions produce a serializable history as if there was only one copy available to transactions. A history H of transactions is serializable if all committed transactions in H issue the same operations and receive the same responses as in some sequential history S that consists of the transactions committed in H [8].

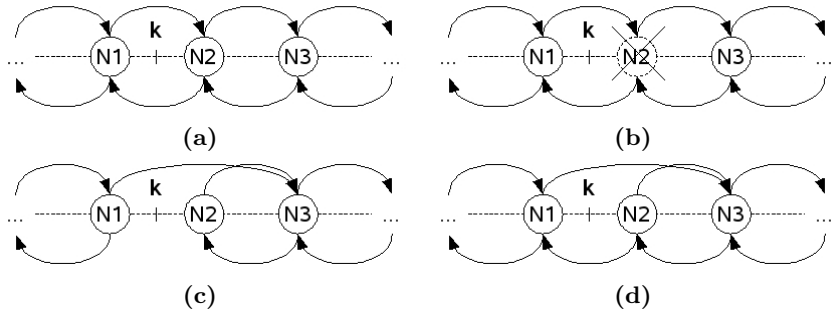


Fig. 3: Availability. Nodes with successor and predecessor pointers: (a) Example where key k is available. (b) Example where k is unavailable due to the failure of $N2$. (c) Example where k is unavailable during the joining process of $N2$. (d) Example where k is unavailable because $N1$ suspected $N2$ to have failed.

3.2 Transaction Processing

Transactions in our system are executed optimistically. They are processed in the following three phases:

- **Read phase (R):** Operations that are part of the transaction are executed within a transaction managers local workspace that is private to the transaction. Changes made by write operations are not visible to other transactions.
- **Validation phase (V):** Once a transaction should be committed, all involved data managers that are responsible for the data that is part of the transaction, check whether the operations are valid. Version numbers are used to determine if another transaction has made changes after the transaction's read phase.
- **Write phase (W):** If all data managers successfully validated the operations on their data, changes can be made permanent.

Atomic Commit. The validation phase and the write phase are executed within an *atomic commit protocol*. An atomic commit protocol coordinates all processes that are involved in a transaction. It ensures that all data managers decide on the same outcome of the transaction. The decision is commit if all data managers are able to validate the operations or abort if there exists at least one data manager that cannot validate an operation. Figure 4 shows a basic commit algorithm called the 2-Phase-Commit Protocol. There is one node called the *transaction manager (TM)* that coordinates the protocol. Data managers are called *transaction participants (TP)*. The TM asks the TPs to validate by sending a *prepare* request. The TPs either reply with *prepared* or *abort*. The TM collects the votes and sends *commit* if all TPs voted to be prepared otherwise it sends *abort*. When a TP receives commit it will make all changes permanent if promised to do when sending the prepared message. If a TP receives abort it won't make any changes permanent. Our framework executes transactions

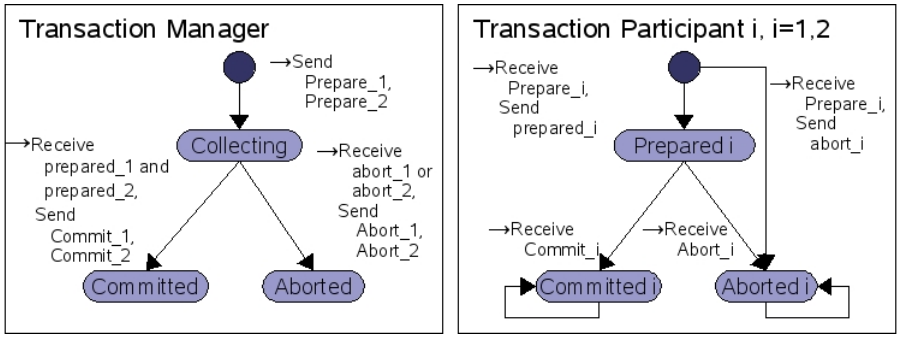


Fig. 4: State-charts for a 2-Phase-Commit Protocol with 2 Participants and 1 Transaction Manager

optimistically in the read phase. Thus the commit protocol will decide on abort if other transactions were committed in between.

A 2-Phase-Commit protocol is blocking if the TM fails in the state collecting and the TPs are not able to retrieve the outcome of the transaction. Therefore we use a non-blocking atomic commit protocol that is based on Paxos [10] in our framework. Gray introduced Paxos Atomic Commit in [7]. It uses replicated transaction managers which all collect votes from the data managers. If the leading transaction manager fails, these replicated transaction managers take over and distribute the decision of the atomic commit protocol to all participants. The Paxos Protocol and the Paxos Atomic Commit protocol are described later in this section.

Concurrency Control. Once a data manger successfully validates an operation on an item it has to ensure that no other transaction gets validated on the same item with a conflicting operation until the atomic commit protocol decides on the outcome. Therefore read and write locks are used during the commit phase that prevent concurrent conflicting validations.

Locks are only held during the atomic commit phase. Instead of letting transactions wait for a lock a TP will vote to abort in the atomic commit phase if it cannot acquire the lock for an operation. In that case the transaction has to be re-executed. This avoids distributed deadlock detection. We assume that read and write operations are less frequent than in traditional database systems. Thus the ratio of aborted transactions should be small.

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 [9]. 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 [10] 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.

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 [7] 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 decision of all proposers and send the acknowledgment for the vote of a TP's Paxos instance to the leader. Whenever the leader has collected enough acknowledgments for each participant's Paxos instance, it decides on commit if all instances have decided on prepared or it decides on abort if there is at least one Paxos instance of a participant that decides on abort. Thereafter the final abort/commit is sent to the initial proposers. If the leader is suspected 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 4. 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.

3.3 Replication

To provide higher reliability items are replicated. Each item has a fixed number of replicas. The replication scheme used here is key based. A key based replication essentially means that an item, which is a key-value pair, is stored under r replica keys. Thus, to store an item under key k, the value will be stored in the DHT under keys $Kr = \{k_1, k_2, k_3 \dots k_r\}$. We say that the replication degree is r and for key k, the set Kr to be the set of keys under which k is replicated. Each replica can be accessed symmetrically as the function to determine the replica keys is system-wide known [4].

As SONs usually are highly dynamic systems, operations on an item should make progress despite the unavailability of a number of replicas. Reads and writes thus require that a majority of replicas is accessible. A minority of replicas might be temporarily unavailable without hindering progress. Operations in our SON use majority-based algorithms. Majority-based algorithms are a special case of quorum algorithms. Quorum algorithms were introduced by Gifford [6] in order to maintain replicated data. Each replica is assigned a certain amount of votes. Read operations have to collect rv votes and write operations have to collect wv votes, where $rv + wv$ exceeds the total number of votes assigned to all replicas of an item. This ensures that read operations include at least one replica that was included by the latest write operation. In majority-based algorithms each replica is assigned exactly one vote and read and write operations have to include a majority of $m = \lfloor \frac{r}{2} \rfloor + 1$ votes. Thus they intersect in at least one replica.

3.4 Serializability in Presence of Responsibility Inconsistency

In order to ensure that $rv + wv$ always exceeds the total number of votes assigned to all replicas, the number of replicas in the system has to be constant for our majority-based algorithms. Each operation on an item has to ensure that it includes at least a majority of replicas, while the majority is based on the system's replication factor r. An additional replica that is added to the system would violate the above mentioned condition. However a responsibility inconsistency is equal to adding an additional replica. In that case two conflicting operations might end up with working on two disjoint sets with a majority of replicas, which we call *majority set*. This happens if two operations work on majority sets that both include distinct nodes that are involved in a responsibility inconsistency for one replica, but have no other replica in common. In that case it is not possible to detect a conflict between these operations, which can violate serializability. As it is not possible to ensure responsibility consistency, it is not possible to ensure serializability. However the existence of a responsibility inconsistency does not necessarily implicate disjoint majority sets for two conflicting operations.

In [12] we calculated the probability for two operations in one configuration to work on non-disjoint majority sets. If the probability for a failure detector to make false positives is 2% and therefore the probability to have a consistent responsibility is 99.999%, the probability for non-disjoint majority sets is 99.9999% if $r = 3$.

4 Transaction Algorithms

In this section we present the algorithms for the transactional DHT. We use an event based notation similar to the one used in [9].

4.1 System Architecture

Nodes in the system can take different roles in a transaction. For each transaction there exists the role of a leading Transaction Manager (TM), called *Leader*, which is the node the client is connected to. Additionally, a number of replicated Transaction Managers (rTM) are created according to the set of acceptors in Paxos commit. Nodes that are responsible for a replica of an item that is involved in the transaction have the role of a Transaction Participants (TP) in the protocol. Each node of the SON can have any number of TMs and TPs that are involved in different transactions. If there are multiple TPs on a node, they must share the database that contains the items with information about read and write locks. Each TP maintains a set of records for ongoing transactions, that have not yet been committed. Each record has a transaction ID, the new proposed value for the items the TP maintains and the new proposed version.

4.2 Transaction ID and Transaction Item

The leader of each transaction creates an unique transaction ID (TID). This ID is part of the SON's key space and can be treated like a item key. The leader creates the TID in a way such that it has a replica key of TID in its own key space. According to the replication scheme there are $r-1$ additional replica keys for the TID. The set of rTMs is determined by the nodes that are responsible for these associated replica keys. Thus the number of all TMs (Leader + rTMs) is equal to the replication factor r. At the end of a transaction each TM will store a replica of a so called *transaction item* with $\{TID, Decision\}$ as the {key, value} pair. We assume that a majority of rTMs does not change its responsibility such that the replica of the TID would not be part of its key space any more. Therefore a node that did not recieve the decision of the transaction can retrieve the decision by doing a quorum read on an item with the TID as key. The transaction item is maintained in the same way as normal items are. However it has to be garbage collected after a certain time.

4.3 System Assumptions

We identify the assumptions related to liveness, no nodes are blocked, and safety, no data is corrupted and 1-copy serializability is not violated. For liveness, it is assumed that direct communication between nodes as well as the bulk procedure is reliable. In addition, a majority of TMs must be alive and keep the TID within their range of responsibility until all alive TPs receive the transaction decision. This assumption is an extension of Paxos where all acceptors (TMs) must be alive during the protocol. For safety, we assume that a majority of replicas for an item are alive and that a majority of lookups targeting these replicase are consistent. A violation of the safety requirements may lead to inconsistent state. The probability of this happening is directly related to the replication factor.

4.4 Identifiers, Modules and Operations

Identifiers. Figure 5 lists all identifiers and variables used in the algorithms. The first part contains general identifiers that are used at transaction managers and transaction participants. The second part contains variables that are maintained by a transaction manager. Additionally, we introduce structures for votes that are received by transaction managers and for acknowledgments of votes. The last part contains variables that store information kept by a transaction participant.

External Modules Used in the Algorithms. The following modules are used by the algorithms

- EventuallyPerfectFailureDetector ($\Diamond P$) [9]
- EventualLeaderDetector (Ω) [9]

A leader uses a failure detector on every replica of the involved items. If it does not get a vote for a replica within a certain time threshold, it will start a failure handling procedure. A failure detector raises the event SUSPECT(tp) when it suspects the transaction participant tp to have failed. A leader election mechanism is used to guarantee progress of the atomic commit protocol. The set of replicated transaction manager will elect a new leader if they suspect the leader to have failed. The leader detector module raises the event TRUST($newleader$) to install a new leader.

Bulk Operation. The algorithms make use of a so called bulk operation [5]. This operation sends events to all nodes that are responsible for a key in a specified set of identifiers. E.g. a read operation on an item can be done with a bulk operation on the set of replica keys for that item.

Identifiers	
item	record
item.key	key
item.val	value
item.ts	timestamp/version number
item.op	kind of operation: write or read
tm	transaction manager
tp	transaction participant
r	replication degree of the system
Information maintained by a TM	
tid	ID of the transaction
TPs	set of Transaction Participants
TMs	set of replicated Transaction Managers
I	set of items involved in the transaction
Votes	Votes of the participants
AcksTMs	Acknowledgments sent by the TMs to the Leader
outcome	Overall outcome of the transaction
state	Either collectingNodes/collectingVotes/locallyDecided/decided
Suspected	set of nodes which are suspected to have failed
leader	the address of the leader
client	the address of the client issuing the transaction
vts	timestamp of a vote
rvts	timestamp of a vote acknowledged in a read phase
wvts	timestamp of a vote acknowledged in a write phase
ItemsInTrans	set of items that are currently involved in a transaction
Information contained in a vote	
i.key	key of the item the vote refers to
rkey	the key of the replica
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal - number of the proposal
Votes[i.key][rkey] = (vote, rts, wvts)	
vote	PREPARED/ABORT decision of a tp
rvts	timestamp of the vote that was accepted during the read phase
wvts	timestamp of the vote that was accepted (write phase)
AcksTMs[i.key][rkey]: {(vote, vts)*}	
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal that was accepted (write phase)
Information kept by a TP	
tid	ID of the transaction
TMs	transaction managers
i	the item
decision_of_tp	the decision it made

Fig. 5: Identifiers used in the algorithms

4.5 Algorithms

In the following we present the algorithms for the transaction processing. We first show the algorithm for the fault-free scenario. The algorithms for failure handling are shown separately in Section 5. The whole transaction processing algorithms refers to the execution of exactly one transaction. Thus we commit information that identifies a particular transaction for better readability.

The algorithms can be structured into different phases. Figure 6 identifies two main phases of a transaction. One is the *Read Phase* where the client determines the operations that are part of the transaction. The second one is the *Commit phase* which we further divide into *Initialization*, *Validation* and *Consensus*.

In the initialization phase the leader determines all nodes that act as replicated transaction managers (TMs). It determines the nodes by a key based search (lookup). After initialization these nodes communicate directly with each other without using a key based search. In the validation phase all TPs are sent a prepare request by a key based search. They are asked to validate the operations on the items they are responsible for. After validation the consensus on the outcome of the transaction is started, based on the validation results. The outcome is sent to the TPs directly with out doing a lookup.

4.6 Read Phase

During the read phase the client determines the operations that are part of the transaction. It can be any sequence of read and write operations. A client is connected to a certain node in the system. This node becomes the initial transaction manager which will act as the leader during the protocol. Figure 7 shows the particular communication steps in the read phase. The client instructs the leader to start a transaction (Algorithm 1) and to do operations until the client tells the leader to commit the transaction. The leader keeps track of the operations and keeps updates on items private to its local workspace. For read operations it will retrieve the value and version number of the item the client wants to read, while for write operation it has to retrieve the version number only (Algorithm 2). When the client signals the end of the transaction the leader will start a commit phase. Instead of instructing the leader to commit the transaction the client can also instruct it to abort the transaction before the commit phase. E.g. if the client reads a value that does not meet a certain condition. In that case the leader can simply throw away logged information on that transaction as the TPs have not yet made changes to their state, such that they do not have to be notified about this user triggered abort. Once a client tells the leader to commit a transaction the client cannot abort it any more on its own behalf.

Algorithm 2 includes a function *latest(Items)* which extracts the item with the highest version number from a set of items. It uses a *DB(key, rkey)* function that reads a replica from the local database. The replica is identified by the key of the item and a replica key or replica number. The function *replicakeys(key)* return all keys of replicas for a certain key.

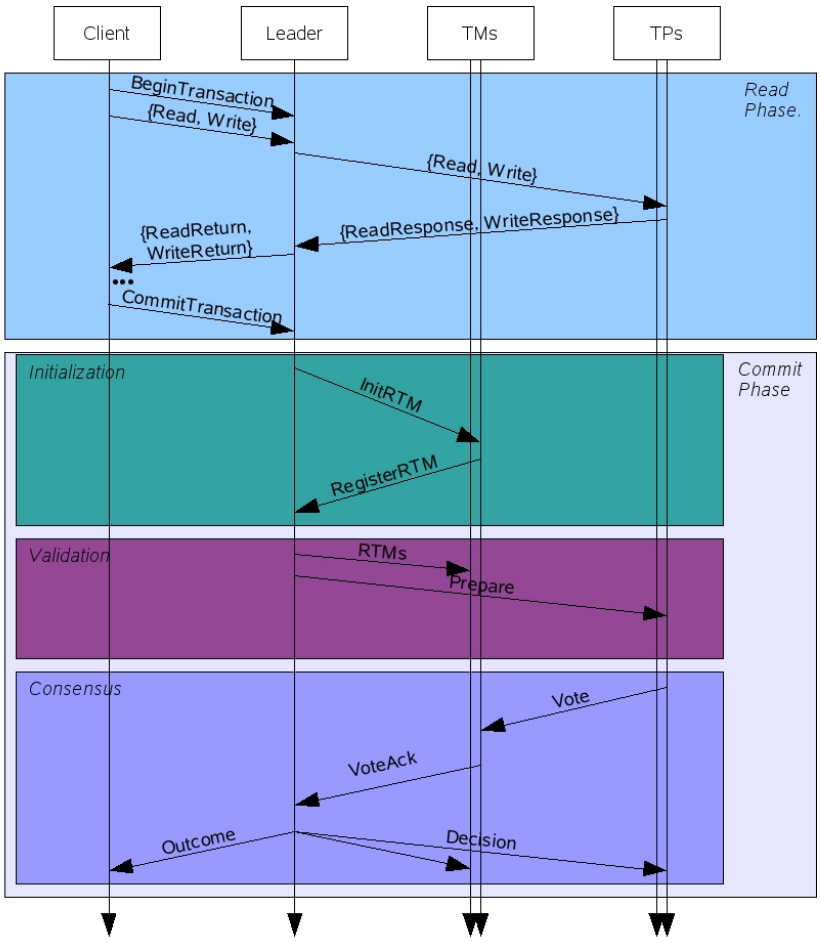


Fig. 6: The figure shows the different phases for a transaction together with the messages sent between the participating nodes

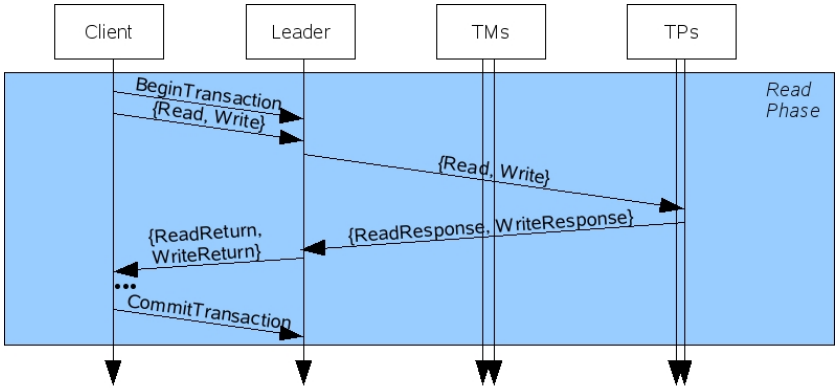


Fig. 7: The figure shows the messages which are part of the read phase. To start a transaction a client issues a BeginTransaction and signals the end of a transaction by a request to commit the transaction. In between it will add several *read* and *write* operations.

Algorithm 1 Interface to the Transaction Manager: Client signals Begin and End of a Transaction

```
1: upon event BEGINTRANSACTION() from client at tm
2:     client := client
3:     I := ∅
4:     readID:= writeID:= ⊥
5:     tid := generateTID()
6: end event

7: upon event COMMITTRANSACTION() from client at tm
8:     trigger STARTCOMMIT()
9: end event

10: upon event ABORTTRANSACTION() from client at tm
11:     delete information on transaction
12: end event
```

Algorithm 2 Processing of a Read Operation due to a Client's Read Request

```
1: function latest(Items) returns item is
2:     tmp_item := item{key:= ⊥, val:=⊥, ts:=−1, op:=⊥}
3:     foreach i in Items do
4:         if i.ts >tmp_item.ts then
5:             tmp_item := i
6:         end if
7:     end foreach
8:     return tmp_item

                                  ▷ A client requests a read operation at the Transaction Manager
9: upon event READ(key) from client at tm
10:    readID := createRID()
11:    Reads := ∅
12:    trigger BULK(replicakeys(key), {READ, key, readID})
13: end event

                                                          ▷ At the Transaction Participant
14: upon event BULK(rkey,{READ, key, readID}) from tm at tp
15:    i := DB(key, rkey)
16:    sendto tm : READRESPONSE(key, i.val, i.ts, readID)
17: end event

                                                          ▷ At the Transaction Manager
18: upon event READRESPONSE(key, val, ts, id) from tp at tm
19:    if readID=id then
20:        Reads := Reads ∪{(key, val, ts)}
21:    end if
22: end event

23: upon |Reads| ≥ (⌊r/2⌋ + 1) ∧ readID ≠⊥ do
24:    (k, val, v):= latest(Reads)
25:    I:= I ∪ item{key:= k, val:=val, ts:=v, op:=r}
26:    sendto client : READRETURN(value)
27:    readID := ⊥
28: end event
```

Algorithm 3 Processing of a Write Operation due to a Client's Write Request

▷ A client requests a write operation at the Transaction Manager

upon event WRITE($key, value$) **from** *client* **at** tm
    writeID := createWID()
    Writes := $\emptyset$
    trigger BULK(ReplicaKeys(key), {WRITE, key, writeID})
end event

▷ At the Transaction Participant

upon event BULK($rkey$,{WRITE, key, $writeID$}) **from** tm **at** tp
    i := DB(key, $rkey$)
    sendto tm : WRITERESPONSE($key, i.ts, writeID$)
end event

▷ At the Transaction Manager

upon event WRITERESPONSE(key, ts, id) **from** tp **at** tm
    if writeID=id **then**
        Writes := Writes $\cup\{(key, ts)\}$
    end if
end event

upon |Writes| $\geq (\lfloor r/2 \rfloor + 1) \wedge$ writeID $\neq \perp$ **do**
    (k, v):= latest(Writes)
    I:= I $\cup$ $item${key:= k, val:=$value$, ts:=v+1, op:=w}
    sendto client : WRITERETURN(success)
    writeID := $\perp$
end event

4.7 Commit Phase

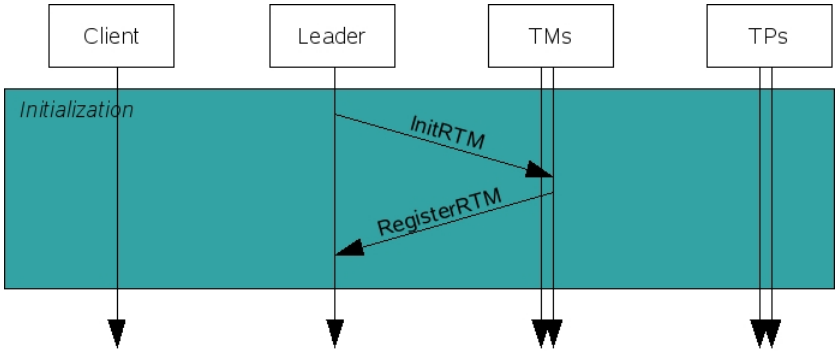


Fig. 8: Before starting the commit protocol the Leader determines the nodes that act as replicated TMs.

Initialization Figure 8 shows the course of events of the initialization phase. Initially nodes that have to act as rTMs are determined by a key based search based on the replica keys of the transaction ID. They have to be known before the commit protocol is started in order to enable leader election among the TMs and make the protocol fault-tolerant and non-blocking. In the next phase the leader will tell the rTMs the addresses of all other rTMs. As long as a majority of rTMs is reachable the protocol can decide on an outcome of the transaction.

Algorithm 4 contains the events and event handlers of the initialization phase. A node that gets a request to initialize a rTM has to initialize its data structures to collect the votes and the acknowledgments. For each item key and its replica keys the vote is initialized with a *rts* (read timestamp) with a value 1. The reason is that a TP will immediately start a write phase in its Paxos instance as it can be sure to be the first one that votes in that particular instance [7]. A TP's proposal number thus is 1.

Once the leader has collected enough TMs it can start the next phase. The leader always tries to collect all TMs. However if some of these nodes do not respond it can start the next phase after a timeout if it has collected at least a majority of TMs. A majority of TMs including the leader is necessary to guarantee progress for Paxos atomic commit.

Algorithm 4 Initialize the involved processes

```
1: upon event startCommit() at leader
                                        ▷ client is the process issuing the transaction
2:     trigger bulk(replicas(tid), {InitRTM, leader, tid, I, client})
3: end event

4: upon event bulk(rtid, {InitRTM, l, id, Items, cl}) from s at rtm
5:                                      ▷ A new transaction manager instance is created
6:     leader := l
7:     tid:= id
8:     I := Items
9:     client := cl
10:    foreach i in I do
11:        foreach rkey in replicaKeys(i.key) do
12:            Votes [i.key][rkey] := (⊥, 1, 0)
13:            AcksTMs [i.key][rkey] := ∅          ▷ Necessary if it becomes a leader
14:        end foreach
15:    end foreach
16:    sendto leader : registerRTM(rtm)
17:    state:= collectingVotes
18: end event

19: upon event registerRTM() from rtm at tm
20:    TMs := TMs ∪ rtm
21: end event

22: upon (|TMs| = r) do
23:    trigger startValidation
24: end event
```

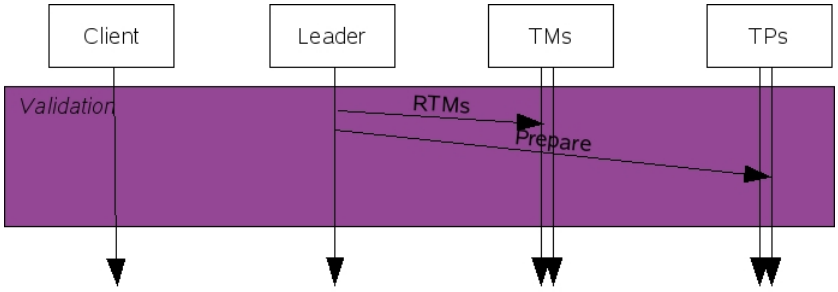


Fig. 9: The Leader sends a Prepare messages to the TPs which will start the validation and tells the TMs about all rTMs.

Validation After the initialization phase the leader tells each rTM the addresses of the other rTMs such that these are able to run a leader election mechanism among them. Additionally the leader sends the prepare request to the TPs together with the addresses of the nodes that act as rTMs. The prepare request is sent with a lookup operation on all replicas of the involved items. Figure 9 shows the course of events and Algorithm 5 the event handlers at the TMs.

The TPs check whether they can validate the operations sent by the prepare request. Each TP therefore locally applies the concurrency control mechanism. This is based on timestamps and locks. A node uses two dictionaries *readLock* and *writeLock* that contain locks on items. While write locks are exclusive, several read locks can be set at the same time. The locks are globally stored in a node. The $storeToLOG(Params)$ is a function that stores any information on a transaction in a TP's LOG. The $getFromLOG()$ function accordingly gets the information from the LOG. Algorithm 6 contains the procedure for validation.

For read operations a TP checks whether there is no lock for a concurrent write and whether the timestamp of the read request is valid, i.e. larger than or equal to the local item. If both checks are successful it will add a read lock and return prepared. For write operations the TP first ensures that there are no read or write locks set for concurrent conflicting operations. Then it compares the timestamp of the proposed item and the local item. This timestamp must to be equal to the currently stored timestamp + 1. If this is not the case, it means that a write operation has changed the item since it was accessed during the read phase. If both of the checks were successful the procedure will return prepared, otherwise abort. After the validation procedure, the TP starts to propose in a Paxos instance for this particular validation result.

Algorithm 5 Transaction Manager: Sending of a Prepare request

```
1: upon event startValidation at tm
2:     sendtoall TMs : RTMs(TMs, I)
3:     foreach i in I do
4:         trigger bulk(replicas(i), {Prepare, leader, tid, i, rkey, TMs})
5:     end foreach
6:     state:= collectingVotes
7: end event

8: upon event  RTMs(rtms, I) from tm at rtm
9:     TMs:= rtms
10:    (Ω).init(rtms)
11:    foreach i in I do
12:        ◇P.init({replica : (i.key, rkey) ∈ i})
13:    end foreach
14: end event
```

Algorithm 6 Validation Procedure at a Transaction Participant/Concurrency Control

```
1: upon event PREPARE(item, rkey, tid, TMs) from tm at tp
2:     ItemsInTrans:= ItemsInTrans ∪ (rkey, tid)
3:     vote:= validate(item, rkey)
4:     trigger PROPOSE(item.key, rkey, TMs, vote, 1)
5: end event

6: procedure validate(item rkey, tid) returns PREPARED/ABORT is
7:     i:= DB(item.key, rkey)
8:     result := ⊥
9:     if item.op = read then
10:        if writeLock[(i.key, rkey)] = ⊥ & (item.ts ≥i.ts) then
11:            readLock[(i.key, rkey)] := readLock[(i.key, rkey)] + 1
12:            storeToLOG(tid, item.key, item.ts,r, rkey,PREPARED)
13:            result:= PREPARED
14:        else
15:            storeToLOG(tid, item.key, item.ts,r, rkey,ABORT)
16:            result := ABORT
17:        end if
18:    else
19:        if writeLock[(i.key, rkey)] = ⊥ & readLock[(i.key, rkey)] = ⊥ & (item.ts =
       i.ts+1) then
20:            writeLock[(key, rkey)] := 1
21:            storeToLOG(tid, item.key, item.ts, item.val,w, rkey, PREPARED)
22:            result:= PREPARED
23:        else
24:            store (tid, item.key, item.ts, item.val,w, rkey, ABORT)
25:            result := ABORT
26:        end if
27:    end if
28:    return result
```

Consensus This phase uses the Paxos atomic commit protocol [7]. At the end of it each TP has to receive the same decision on the transaction to ensure the uniform consensus properties. The decision will be commit if the decision for all items is prepared, it will be abort if the decision for at least one item is abort.

A Paxos instance is started for each replica. The TP that is responsible for the replica uses this Paxos instance to distribute its vote on the set of replicated transaction managers that act as the acceptors. The replicated transaction managers accept the vote of a TP if they did not get a read request with a higher timestamp for the particular Paxos instance. Instead of sending the acknowledgment to the TP they send it to the leader (Algorithm 8). The reason is that the decision on the outcome of the atomic commit protocol is based on the decisions for all items. Therefore, the leader collects all acknowledgments from where it can derive the outcome of the atomic commit protocol. As soon as the outcome is known the leader sends it to all involved nodes and notifies the client.

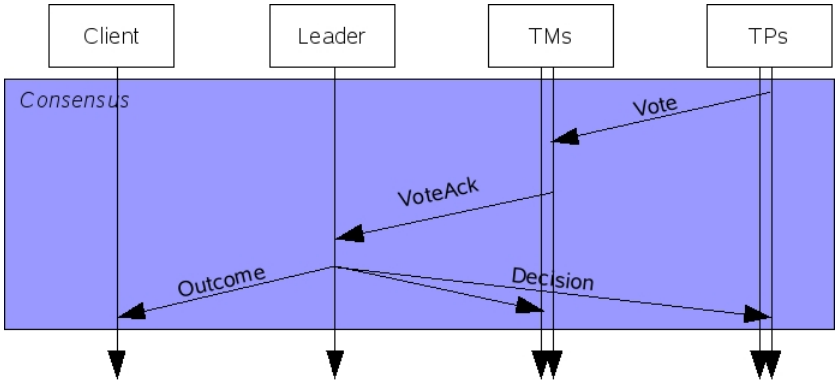


Fig. 10: After validation the TPs start a consensus to distribute their validation result. The leader will collect the decision for each particular consensus instance and conclude on the overall outcome of the transactions based on these instances.

As items in our DHT are replicated and operation on items require at least a majority of replicas to be accessible, the decision for an item also has to be based on a majority of replicas. Therefore the TM collects the votes of the TPs per item. A decision for an item is prepared if a majority of TPs that are responsible for a replica of that item vote to be prepared. The decision for an item is abort if there cannot be a majority of TPs that are responsible for a replica that vote to be prepared (Algorithm 9).

A TP that retrieves the decision for a transaction reads the information about the item it voted for from its LOG (Algorithms 10 and 11). It has to reset locks and write the item to the database if necessary. The learners are related to replica maintenance, which is explained in Section 6.

A TM that retrieves the decision for a transaction stores the decision in a special item, called *transaction item*, by calling $storeTransactionItem(decision)$. This item is replicated like all other items. The key for this transaction item can be deduced from the transaction ID. All TMs are nodes that are responsible for a replica key of the transaction item. If a node does not receive the decision by the normal execution of the protocol it can retrieve the decision by reading the transaction item like every normal item. We assume that a majority of replicated transaction managers may not fail and may not change their responsibility of the key range where the replica key of the TID is a member of until the outcome of the transaction is stored in the DHT. This can be achieved by choosing a proper replication factor.

Algorithm 7 Start Paxos Atomic Commit: Propose in a Paxos Instance

```
1: upon event PROPOSE(key, rkey, TMs, vote, ts) at p
2:     if ts=1 then
3:         sendtoall TMs : VOTE(key, rkey, vote, ts)
4:     else
5:         sendtoall TMs : READVOTE(key, rkey, ts)      ▷ See failure handling in 13
6:     end if
7: end event
```

Algorithm 8 Paxos Atomic Commit - Write Phase

```
1: upon event VOTE(key, rkey, vote, vts) from n at tm  ▷ n is either a tp or a tm
2:     if vts = 1 then
3:         TPs[key]:= TPs[key] ∪ (p, rkey)
4:     end if
5:     (currVote, rvts, wvts) := Votes[key][rkey]
6:     if vts ≥rvts and vts ≥ wvts then
7:         Votes[x.key][rkey] := (vote, rvts, vts)
8:         sendto Leader : VOTEACK(key, rkey, vote, vts)
9:     end if
10: end event

11: upon event VOTEACK(key, rkey, vote, vts) from tm at leader
12:     AcksTMs[key][rkey] := AcksTMs[key][rkey] ∪ {(vote, vts)}
13: end event
```

Algorithm 9 Paxos Atomic Commit - Making the Decision

▷ Ensure that a majority of TMs has stored the decision for a particular replica
function isPreparedReplica($i, rkey$) **returns** boolean **is**
    acks := AcksTMs[$i.key$][$rkey$]
    return $\exists vts : |\{(\text{PREPARED}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

function isAbortReplica(i, n) **returns** boolean **is**
    acks := AcksTMs[$i.key$][$rkey$]
    return $\exists vts : |\{(\text{ABORT}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

▷ Check whether the votes for a majority of replicas for an item is PREPARED
function isPrepared(i) **returns** boolean **is**
    return $|\{rkey : \text{isPreparedReplica}(i, rkey)\}| \geq \lfloor r/2 \rfloor + 1$

function isAbort(i) **returns** boolean **is**
    return $|\{rkey : \text{isAbortReplica}(i, rkey)\}| \geq \lceil r/2 \rceil$

upon $\forall i \in$ I: isPrepared(i) **at** *leader* **do**
    sendtoall TPs : DECISION(COMMIT)
    sendtoall TMs : DECISION(COMMIT)
    state:= DECIDED
    sendto client : OUTCOME(COMMIT)
end event

upon $\exists i \in$ I: isAbort(i) **at** *leader* **do**
    sendtoall TPs : DECISION(ABORT)
    sendtoall TMs : DECISION(ABORT)
    state:= DECIDED
    sendto client : OUTCOME(ABORT)
end event

upon event DECISION(*decision*) **from** tm **at** tm
    storeTransactionItem(*decision*)
    state:= DECIDED
end event

upon event DECISION(*decision*) **from** tm **at** tp
    trigger DECIDE(*decision*)
end event

Algorithm 10 Paxos Atomic Commit - Decide COMMIT at a TP

```
1: upon event DECIDE(COMMIT) at tp
2:     if not stored(COMMIT) then
3:         item := getFromLOG(tid)
4:         if Item = ⊥ then
5:             sendafterdelay(time, DECISION(COMMIT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                    ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)] := 0
15:                    DB(key, rkey) := (val, ts)
16:                else
17:                    DB(key, rkey) := (val, ts)
18:                end if
19:            end if
20:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
21:            foreach l in learners do
22:                remove(learners, Learners[rkey])
23:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
24:                if ltidrest = ∅ then
25:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
26:                end if
27:            end foreach
28:            ItemsInTrans := ItemsInTrans \rkey
29:            storeToLOG(COMMIT)
30:        end if
31:    end if
32: end event
```

Algorithm 11 Paxos Atomic Commit - Decide ABORT at a TP

```
1: upon event DECIDE(ABORT) at tp
2:     if not stored(ABORT) then
3:         item := getFromLOG()
4:         if item = ⊥ then
5:             sendafterdelay(time, DECISION(tid, ABORT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                    ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)]) := 0
15:                end if
16:            end if
17:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
18:            foreach l in learners do
19:                remove(learners, Learners[rkey])
20:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
21:                if ltidrest = ∅ then
22:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
23:                end if
24:            end foreach
25:        end if
26:        storeToLOG(ABORT)
27:    end if
28: end event
```

5 Transaction Algorithms: Failure Handling

This section covers failure handling of the atomic commit protocol during the consensus phase. Critical failures are those that block the transaction participants leaving a replica in a locked state. These can occur after the leader has sent the prepare request to the TPs. If the leader fails before that or if the leader cannot contact enough TPs or rTMs the transaction will simply be aborted. Failures have to be handled in a way that guarantees progress of the transaction processing while guaranteeing the properties of uniform consensus and atomic commit. In the following we distinguish between the failure of a TP and the failure of a leader. The failure handling reflects the one described in the Paxos atomic commit paper[7].

5.1 Failure of the Leader

When the leader fails the leader election mechanism will elect a new leader among all TMs. This new leader has to retrieve enough acknowledgments from the TPs' Paxos instances in order to be able to decide on the outcome of the transaction. Therefore it starts with a read phase in each single Paxos instance of the TPs (Algorithm13). In the write phase it will adopt the votes that have been proposed so far or vote to abort if there hadn't been a vote in the Paxos instance. As soon as the new leader got enough acknowledgments it can distribute the outcome of the commit phase. Figure 11 shows the course of events when a leader fails.

Algorithm 12 Replicated Transaction Manager: Trusting a New Leader

```
 1: upon event TRUST(leader) at tm
 2:     Leader := leader
 3:     newts := nextVts()
 4:     if state ≠ DECIDED ∧Leader = self then
 5:         foreach i in I do
 6:             foreach rkey in replicas(i.key) do
 7:                 trigger PROPOSE(i.key, rkey, TMs, ⊥, newts)
 8:             end foreach
 9:         end foreach
10:     end if
11: end event
```

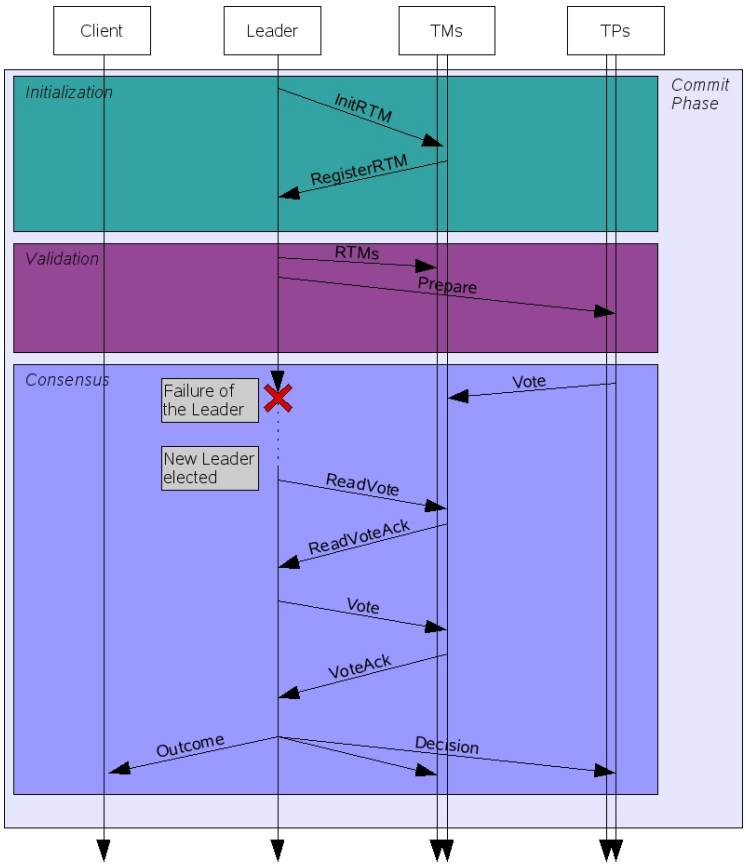


Fig. 11: If a Leader fails, another node will become the new leader and start a read phase in each TP's Paxos instance.

Algorithm 13 Paxos Atomic Commit - Read phase

```
 1: upon event readVote(i.key, rkey, vts) from leader at tm
 2:     (currVote, rvts, wvts) := Votes[i.key][rkey]
 3:     if vts >rvts and vts >wvts then
 4:         Votes[i.key][rkey] := (currVote, vts, wvts)
 5:         sendto Leader : readVoteAck(i.key, rkey, vts, (currVote, wvts))
 6:     end if
 7: end event

 8: upon event readVoteAck(i.key, rkey, vts, (vote, wvts)) from tm at leader
 9:     ReadVotes[i.key][rkey][vts] := ReadVotes[i.key][rkey][vts] ∪ {(vote, wvts)}
10: end event

11: upon ∃i.key, rkey, vts : | ReadVotes[i.key][rkey][vts]| ≥ ⌊r/2⌋ + 1 at Leader do
12:     vote := highest(ReadVotes[i.key][rkey][vts])
13:     if vote = ⊥ then
14:         vote := ABORT
15:     end if
16:     sendtoall TMs : vote(i.key, rkey, vote, vts)
17: end event
```

5.2 Failure of a TP

It must be noted that the protocol makes progress as long as foreach item a majority of TPs that are responsible for a replica is alive. In that case it is not necessary to start a failure handling for a TP. If there exists one item for which a majority of TPs cannot be reached, a leader would have to start voting in as many Paxos instances it needs to get a majority of votes for the item. However this situation occurs only if the system is broken, as we assume that for each item a majority of nodes being responsible for a replica is always available.

When the leader does not get a vote for a replica it suspects the corresponding TP to have failed. It will start to propose in the Paxos instance for that replica. As it does not know whether its suspicion is correct or the TP is alive and has already started to vote, the leader starts with a read phase. It will learn from the TMs whether there has been a vote made by the TP. If this is not the case the leader is free in its decision and will start a write phase with the value abort. Thus the leader decides to abort on behalf of the suspected TP.

Algorithm 14 Leader: Suspecting a TP during Consensus - Start a Read Phase

```
 1: function nextVts() returns vts is
 2:     select the next vts depending on the nodes key
 3:                                   ▷ TM_r will come with a proposal numbered r+1
 4:                                   ▷ Set of TMs: TM_1, TM_2, .., TM_r
 5:

 6: upon event SUSPECT((key, rkey)) at leader
 7:     foreach i in I do
 8:         trigger PROPOSE(key, rkey, TMs, nextVts())
 9:     end foreach
10: end event
```

6 Transactional Replica Maintenance

When a node joins the structured overlay network it will take over the responsibility for a certain key range from an existing node in the system. Similarly, when a node fails, its successor in the structured overlay network has to take over responsibility for the failed node's key range. A node knows that its responsibility has changed whenever it has to update its predecessor pointer.

The framework has to handle joins and failures of a node on the data level in a way that maintains data consistency. Handling these events on the data level is done after they are handled on the overlay level. E.g. a joining node n first sets its successor and predecessor pointers and notifies the corresponding nodes about itself. Thereafter the new node n has to retrieve the data it is responsible for. Data retrieval mechanisms may not decrease the number of up-to-date copies for an item and they may not violate serializability of transactions.

Whenever the predecessor of a node is changed the event is handled on the data-level. There are two possible situations. Either the range the node is responsible for was increased or it was decreased. In the first case the node has to fetch the data it has not stored yet, in the second case the node has to drop data it is no longer responsible for.

A new copy of an item can be initialized by any transaction that writes that item or it is initialized explicitly[2]. As the first possibility may potentially take a long time and other copies of the item might fail during that time, it is better to do an explicit initialization of the copy to decrease the probability of a system failure. The system fails if an operation that is performed on a majority of nodes holding a replica cannot return an up-to-date item. Explicit initializing of a new copy can be done by a copy operation that reads the existing copies of the item and stores the current value in the new copy. Additionally all transactions that update the item must know of the new copy. Otherwise a non-serial history could arise by the following three events:

- Transaction $T1$ updates x and y
- Node $N1$ joins and will become responsible for replicas x_1 and y_1

– Transaction $T3$ reads x and y

The non-serial sequential history of events and transaction phases that can occur:

1. $T1$ - Initialization phase: Get addresses of involved nodes
2. $T1$ - Validation phase: Send prepare request with lookup
3. $N1$ joins the ring (updates its successor, predecessor and the others)
4. $N1$ copies x_1 from an existing copy: Gets the old value
5. $T1$ - Consensus phase: Outcome is received by all TPs
6. $N1$ copies y_1 from an existing copy: Gets the new value
7. x_1 at $N1$ is out of date

In this situation the number of replicas that are out of date would be increased. This happens if the initialization of an item does not take into account ongoing transactions on the item. The copy operation to initialize a replica has to be done either with a transaction or by adding the new node as learner to ongoing transactions, as we will explain in the following. A node that has to initialize the replicas in its range has to know which items exist that have replicas in that range and it has to copy the data for these replicas. If the initialization would have been done by one transaction this transaction would fail if there is even one single write operation on one of the items. Another possibility is that all ongoing transactions have the new node as a so called *learner* for the outcome. This concept is used in our framework and introduced in the following.

6.1 Copy Operation

The operation that initializes copies in a certain key range will be called *copy operation* in the following. Within this operation a node asks the nodes that are responsible for the corresponding remaining replicas of the items in its new range to send their replicas to it. It has to read from at least a majority of replicas. Once the nodes being responsible for the remaining replicas get a copy request, they have to remember the new node as a learner for all ongoing transactions. They send to the new node all replicas that are not involved in a transaction. Items that are currently involved in the transaction are sent as soon as there is no transaction run on them any more for which the new node is registered as learner. This is a way to let all ongoing transactions know the new copy. The new node will not be able to answer requests for items in its range until it has retrieved the data for it.

6.2 Join and Leave

When a node joins or leaves the system the successor of the joining or leaving node changes its predecessor. This means that the key range of the successor node is changed. In the following we assume that the routing layer triggers an event called NewPredecessor. Upon such an event a node checks whether the

range it is responsible for has increased or has decreased. In the first case it has to copy data, in the second case it has to drop data. Similarly, a new node will fetch data once it knows the range it is responsible for.

Algorithm 15 Data Level: New Predecessor Event from the Routing Level

```
        ▷ The NewPredecessor event is triggered on the routing level, after a new
    predecessor is set due to join or periodic stabilization
 1: upon event NewPredecessor(prevpred, newpred) at n
 2:     if newpred ≠ nil then
 3:         if newpred ∈ (prevpred, n) then
 4:             trigger dropData(prevpred, newpred)
 5:         else
 6:             trigger fetchData(newpred, prevpred)
 7:         end if
 8:     else
 9:         oldpred=prevpred
10:     end if
11:     if prevpred = nil then
12:         trigger dropData(n, newpred)
13:         if oldpred ∈ (newpred, n) then
14:             trigger fetchData(newpred, oldpred)
15:         end if
16:     end if
17: end event
```

Algorithm 16 shows the copy operation. A node that has to fetch data first calculates all the other replica keys that correspond to its key range. It starts a bulk operation with these replica keys to read the data it has to store. Each node that gets a copydata request will send the values or add the requesting node as a learner to the transactions on items that are in the requested range.

Algorithm 16 Data level: Drop and fetch data

```
                                  ▷ After setting successor and predecessor and notifying the others
 1: upon event FETCHDATA(start, end) at n
 2:     ReplicaKeys:= getCorrespondingReplicaKeys(start, end),
 3:     trigger BULK(ReplicaKeys, {COPYDATA})
 4: end event

                                  ▷ After a new predecessor is set on the overlay level
 5: upon event DROPDATA(start, end) at n
 6:     DB:= DB \ DB ([start, end])
 7: end event

 8: upon event BULK([x, y], {COPYDATA})) from n' at n
 9:     foreach (r.key, tid) in ItemsInTrans do
10:         if r.key ∈ [x, y] then
11:             Learners([r.key]) = Learners([r.key]) ∪ (n', tid)
12:         end if
13:     end foreach
14:     Items := DB([x, y])
15:     Items := Items \ ItemsInTrans
16:     sendto n' : COPYDATARESPONSE(Items)
17: end event

18: upon event COPYDATARESPONSE(Replicas) from n at newnode
19:     foreach i in Replicas do
20:         myrkey := getOwnReplicaKey(i.key)
21:         MyData[(myrkey, i.key)]:= MyData([(myrkey, i.key)]) ∪ (i.val, item.ts)
22:         if | MyData [(myrkey, i.key)]| ≥ ⌊r/2⌋ + 1 then
23:             if latest(MyData[(myrkey, i.key)]) > DB(i.key, myrkey) then
24:                 DB(i.key, myrkey):=(i.val, i.ts)
25:             end if
26:         end if
27:     end foreach
28: end event
```

7 Evaluation

7.1 Analytical Evaluation of the Commit Protocol

Number of messages Figure 12 illustrates the different communication steps that are necessary for a failure free execution of the protocol. Including the initialization phase six steps are required to make a decision on the outcome of the transaction. Note that in our algorithms the leader is at the same time a transaction manager. The number of messages depends on the replication factor r in the system and the number n of items involved in the transaction. This are the number of messages sent in each step:

1. r Lookup message for TMs

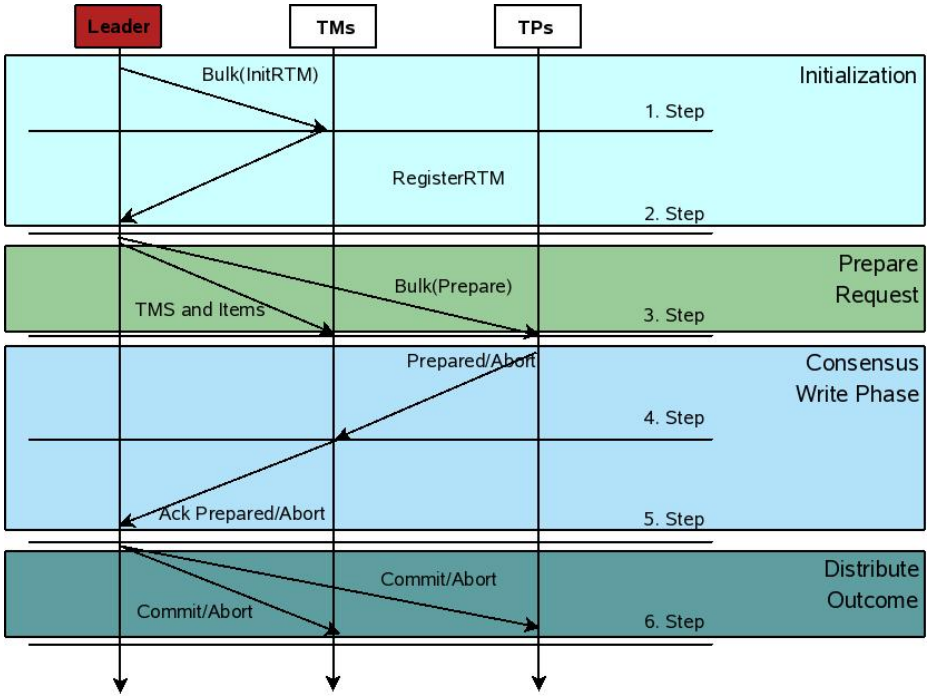


Fig. 12: The figure shows the communication steps required for the whole atomic commit protocol

2. r Registration messages from TMs
3. $n * r + r$ Prepare request to TPs and information message to TMs
4. $n * r * r$ Vote of TPs to all TMs
5. $n * r * r$ Acknowledgment for each vote
6. $n * r + r$ Decision sent to the TPs and TMs

Overall we need $(1+r)2nr+4r$ messages. The r messages from the first step use a bulk operation that is based on lookups. Similarly the $n * r$ prepare requests in the third step also use lookups. Note that the number of messages can be optimized. A TM can send all the acknowledgments for the votes it got within one message instead of sending a separate message for each vote.

Upper Timebounds

1. $O(logN)$ Lookup request to TMs
2. $O(1)$ Direct communication: Latency of slowest TM
3. $O(logN)$ Lookup request to TPs
4. $O(1)$ Direct communication: Latency of slowest conncection from a TP to a TM in a majority for an item

5. $O(1)$ Direct communication: Latency of slowest TM to leader connection
6. $O(1)$ Direct communication: Latency of slowest leader to TMs and TPs connection

7.2 Experimental Evaluation

The transaction algorithm is implemented as part of Scalaris [11], a P2P-based key/value store. We tested the performance of Scalaris and the transaction algorithm on an Intel cluster up to 16 nodes. Each node has two Quad-Core E5420s (8 cores in total) running at 2.5 GHz and 16 GB of main memory. The nodes are connected via GigE and Infiniband; we used the GigE network for our evaluation.

On each physical node we were running one multi-core Erlang virtual machine. Each virtual machine hosted 16 Scalaris nodes. We used a replication degree of four, that is, there exist four copies of each key-value pair.

We tested two operations: a read and a modify operation. The read operation reads a key-value pair. The modify operation reads a key-value pair, increments the value and writes the result back to the distributed Scalaris store. To guarantee consistency, the read-increment-write is executed within a transaction. The read operation, in contrast, simply reads from a majority of the keys. The benchmarks involved the following steps:

- Start watch.
- Start n Erlang client processes in each VM.
- Execute the read or modify operation i times in each client.
- Wait for all clients to finish.
- Stop watch.

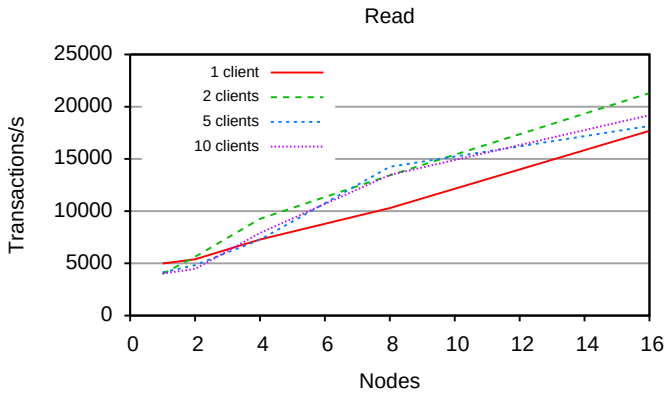


Fig. 13: Read performance of the transaction algorithm. The read operation is executed with increasing number of local threads and cluster sizes.

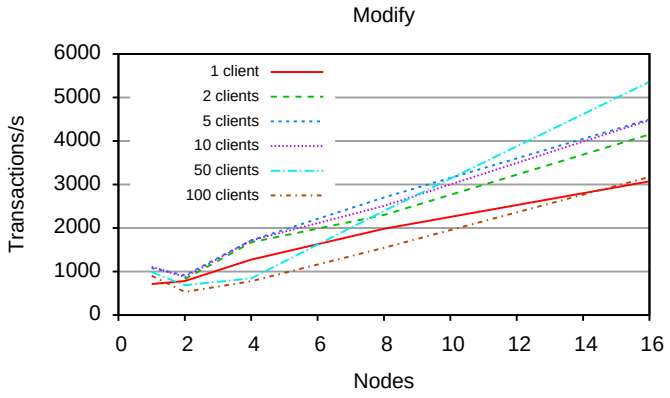


Fig. 14: Write performance of the transaction algorithm. The write operation is executed with increasing number of local threads and cluster sizes.

Figure 13 and 14 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 13, each thread reads a key 2000 times while the modify benchmarks in Fig. 14 modify each key 100 time in each thread.

As can be seen, the system scales about linearly over a wide range of system sizes. In the read benchmarks (Fig. 13), two clients per VM produce an optimal load for the system, resulting in more than 20,000 read operations per second on a 16 node (=128 core) cluster. Using only one client (red graph) does not produce enough operations to saturate the system, while five clients (blue graph) cause too much contention. Note that each read operation involves accessing a majority (3 out of 4) replicas.

The performance of the modify operation (Fig. 14) is of course lower, but still scales nicely with increasing system sizes. Here, the best performance of 5,500 transactions per second is reached with fifty load generators per VM, each of them generating approximately seven transactions per second. This results in 344 transactions per second on each server.

Note that each modify transaction requires Scalaris to execute the adapted Paxos algorithm, which involves finding a majority (i.e. 3 out of 4) of transaction participants and transaction managers, plus the communication between them. The performance graphs illustrate that a single client per VM does not produce enough transaction load, while fifty clients are optimal to hide the communication latency between the transaction rounds. Increasing the concurrency further to 100 clients does not improve the performance, because this causes too much contention. Note that for the 100-clients-case, there are actually 16*100 clients issuing increment transactions. Overall, both graphs illustrate the linear scalability of Scalaris.

8 Discussions

The algorithms do not include garbage collection issues. A transaction manager has to keep the information for a transaction long enough to be sure that each transaction participant knows the outcome. Transaction participants could acknowledge that they got the decision of the atomic commit protocol.

References

1. L. O. Alima, A. Ghodsi, and S. Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *Post-proceedings of Global Computing*, Lecture Notes in Computer Science (LNCS), pages 223–250. Springer Verlag, 2004.
2. Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
3. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.
4. A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems . In *Proceedings of the 3rd International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'05)*, volume 4125 of *Lecture Notes in Computer Science (LNCS)*, pages 74–85. Springer-Verlag, 2005.
5. Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.
6. David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.
7. Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.
8. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.
9. Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
10. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
11. Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable transactional p2p key/value store. In *ACM SIGPLAN Erlang Workshop*, September 2008.
12. Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, and Alexander Reinefeld. On consistency of data in structured overlay networks. In *CoreGRID Integration Workshop 2008*, 2008.
13. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.