FLORIAN SCHINTKE, ALEXANDER REINEFELD, SEIF HARIDI, THORSTEN SCHÜTT

Enhanced Paxos Commit for Transactions on DHTs

ZIB-Report 09-28 (September 2009)

Enhanced Paxos Commit for Transactions on DHTs

Florian Schintke*,Alexander Reinefeld*, Seif Haridi† and Thorsten Schütt*
* Zuse Institute Berlin
† Royal Institute of Technology, Sweden

***Abstract*—Key/value stores which are built on structured overlay networks often lack support for atomic transactions and strong data consistency among replicas. This is unfortunate, because consistency guarantees and transactions would allow a wide range of additional application domains to benefit from the inherent scalability and fault-tolerance of DHTs.**

The Scalaris key/value store supports strong data consistency and atomic transactions. It uses an enhanced Paxos Commit protocol with only four communication steps rather than six. This improvement was possible by exploiting information from the replica distribution in the DHT. Scalaris enables implementation of more reliable and scalable infrastructure for collaborative Web services that require strong consistency and atomic changes across multiple items.

I. Introduction

Distributed hash tables (DHTs) and other structured overlay networks (SONs) were developed to provide an efficient key based location of nodes and associated data in the presence of node joins, leaves and crashes (churn). Due to churn, two challenges arise in such systems: (1) When a node crashes, all data stored on this node is lost. (2) When a node is suspected to be crashed, lookup inconsistencies and responsibility inconsistencies may occur, which may lead to wrong query results or loss of update requests. Responsibility inconsistency occurs when multiple nodes believe they are responsible for an overlapping range of items.

The first issue can be addressed by data replication. The second issue can only be relieved but not overcome: It was shown that in an asynchronous network atomic overlay maintenance is impossible [7] and thus responsibility inconsistency is unavoidable. Clearly, data consistency cannot be achieved if responsibility consistency is violated. But as shown in [19], the probability of inconsistent data accesses can be reduced by increasing the replication degree, and performing reads on a majority of replicas. In typical Internet scenarios, for example, only three replicas give a consistency probability of five nines. It can be further improved by adding more replicas or by increasing the share of nodes required for a quorum, but it can never be made 100%[1].

Scalaris [16] is a transactional key/value store which uses symmetric key replication [8] to ensure data availability in the face of churn. Data consistency is enforced by performing all data operations on a majority of replicas.

In this paper, we present improved algorithms for concurrency control and transaction processing, that are based on approaches presented in [14], [16]:

[1]Inconsistencies might still happen if multiple nodes join between two existing nodes [19].

- We show how Paxos Commit can be efficiently embedded into a DHT to perform a low latency non-blocking atomic commit on replicated items. Our commit protocol including the commit phase and the validation phase requires just four message delays in the failure-free case (Sect. V-B).
- We discuss failure scenarios and explain how they are dealt with (Sect. V-B).
- We illustrate how transactions are executed and validated in Scalaris and how concurrency control is performed using readers-writer locks (Sect. V-C and Sect. V-D).
- We evaluate the latency-critical path of our commit protocol by checking each step for its earliest start time (Sect. VI).

Before going into the details in Sect. V and VI, we discuss related work in the following, describe our general overlay structure and replication scheme in Sect. III and provide the fundamentals of Paxos Consensus and Paxos Commit in Sect. IV.

II. Related Work

There are several production systems that use Paxos Consensus [12], like Google's distributed lock service Chubby [3]. The closest to our work is Etna [13] which provides replicated atomic registers. Etna uses consensus to agree on the replica membership set. It does not provide transactional semantics on multiple data items.

Dynamo [5] is a large-scale key/value store. In contrast to Scalaris [16], Dynamo favours availability instead of strong consistency. It provides eventual consistency and no transactions.

We describe an improved transaction commit protocol which reduces the number of message delays in the failure-free case by two compared to our previous protocol [14].

III. Scalaris: Replicated Data on Structured Overlays

Scalaris [16] is a distributed, transactional key/value store with replicated items. It uses symmetric data replication [8] on top of a structured overlay like Chord [20] or Chord$^{\#}$ [17]. In contrast to many other key/value stores, Scalaris provides strong data consistency. It uses the same transaction mechanism for providing replica synchronization as well as transactional semantics on multiple data items.

In the following, we describe the DHT layer and replication layer.

A. Structured Overlay Networks

Distributed hash tables (DHTs) provide a scalable means for storing and retrieving data items in decentralized systems. They are usually implemented on top of structured overlay networks which provide robustness in dynamic environments with unreliable hosts. A DHT has a simple API for storing, retrieving and deleting key/value pairs: *put(key,value), get(key)*, and *delete(key)*.

We use the structured overlay protocol Chord$^{\#}$ [17] for storing and retrieving key/value pairs in nodes that are arranged in a virtual ring. This ring defines a key space where all values can be stored according to the associated key. Nodes can be placed at arbitrary places on the ring and are responsible for all data between their predecessor and themselves. The placement policy ensures even distribution of load over the nodes.

In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to other DHTs like Chord [20], Kademlia and Pastry, Chord$^{\#}$ stores the keys in lexicographical order. This enables range queries and it gives control over the placement of data on the ring structure, which is necessary when deploying a Chord$^{\#}$ ring over datacenters to have better control over latencies. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way [17] that successive fingers in the routing table jump over an exponentially increasing number of nodes in the ring.

To access the node responsible for a given key k, a *DHT lookup* with an average of $0.5\log_b N$ routing hops is performed. The base b can be chosen according to the application requirements, e.g. faster lookup versus lower space requirements [1].

Due to churn, nodes can join and leave at any time, and the ring must be repaired. Stabilization routines run periodically, check the ring healthiness and repair the routing tables according to the finger placement algorithm. If the ring becomes partitioned, a bad pointer list keeps information on nodes on the other part of the ring and a merge algorithm [18], [11] can be used to rejoin them again.

B. Data Replication

To prevent loss of data in the case of failing nodes, the key/value pairs are replicated over r nodes. Several schemes like successor list replication or symmetric replication [8] exist. Symmetric replication stores each item under r keys. A globally known function places the keys $\{k_1,\ldots,k_r\}$ symmetrically in the key space. Read and write operations are performed on a majority of replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes. This scheme is shown to ensure key consistency for data lookups under realistic networking conditions [19].

IV. Paxos Consensus and Paxos Commit

To provide strong consistency over all replicas, transactions are implemented on top of our structured overlay where symmetric replication is employed. We use optimistic concurrency control with a backward validation scheme. Our Scalaris system uses an adapted Paxos Commit for non-blocking atomic commit, which in turn uses Paxos Consensus for each individual data replica to fault-tolerantly agree on prepared or abort for each replica.

Algorithm 1 Paxos Consensus: Proposer

```
 1: initialize
 2:   $r$ = any round number greater than all $r$ seen before
 3:   multicast prepare(r) to all acceptors
 4:   ack_received = $\emptyset$
 5: on receipt of ack($r, v_i, rlast_i$) from acceptor $acc_i$
 6:   ack_received = ack_received $\cup$ $(r, v_i, rlast_i)$
 7:   if $|ack\_received| > \frac{n}{2}$                    ▷ get index of newest round
 8:     $j = \max(rlast_k$: for all $k$ such that $\{r, v_k, rlast_k\} \in$ ack_received)
 9:         ▷ end of information gathering phase
10:   if $v_j = \bot$                                      ▷ no value agreed yet?
11:     $v_j$ = any_value                                  ▷ we propose a value
12:   multicast accept($r$, $v_j$) to all acceptors
```

We first describe the Paxos Consensus protocol and then discuss the non-blocking atomic commit protocol.

A. Paxos Consensus

In a distributed consensus protocol, all correct (i.e. non-failing) processes eventually choose a single value from a set of proposed values. A process may perform many communication operations during the protocol execution, but it must eventually decide a value by passing it to the client process that invoked the consensus protocol.

Throughout this paper, we assume a fail-stop model where failing processes do not recover. To simulate this behaviour, returning nodes will rejoin with a new identity and empty state.

Lamport's *Paxos Consensus* [12], [15] is a non-blocking consensus protocol for asynchronous distributed systems. Alternative algorithms were proposed by Chandra and Toueg [4] and by Dwork [6]. Paxos implements a *uniform consensus* which achieves agreement even when a minority of processes should fail. Uniform consensus has the following properties [10]:

- *Termination:* Every correct process eventually decides some value.
- *Validity:* If a process decides v, then v was proposed by some process.
- *Integrity:* No process decides twice.
- *Agreement:* No two processes decide differently.

1) Outline of the algorithm: Each process may take the role of a *proposer*, an *acceptor*, or a *learner*, or any combination thereof. A proposer attempts to get a consensus on a value. This value is either its own proposal or the resulting value of a previously achieved consensus. The acceptors altogether act as a collective memory on the consensus status achieved so far. The number of acceptors must be known in advance and must not increase during runtime, as it defines the size of the majority set m required to be able to achieve consensus. The decision, whether a consensus is reached, is announced by a learner.

Algorithm 2 Paxos Consensus: Acceptor

```
1:  initialize
2:    $r_{ack} = 0, r_{accepted} = 0, v = \perp$    ▷ no round acknowledged or accepted yet, no value
3:  on receipt of prepare(r) from proposer
4:    if $r > r_{ack} \wedge r > r_{accepted}$    ▷ new round?
5:      $r_{ack} = r$    ▷ memorize that we saw round $r$
6:      send ack(r, v, $r_{accepted}$) to proposer
7:  on receipt of accept(r, w) from proposer
8:    if $r \geq r_{ack} \wedge r > r_{accepted}$    ▷ latest round?
9:      $r_{accepted} = r$    ▷ memorize that we accepted in round $r$
10:     $v = w$
11:     send accepted($r_{accepted}$, v) to learners
12: on receipt of decided(v) from learner
13:   cleanup()
```

Algorithm 3 Paxos Consensus: Learner

```
1:  on receipt of accepted(r,v) from a majority of acceptors
2:    multicast decided(v)    ▷ v is consensus
```

Proposers trigger the protocol by initiating a new *round*. Acceptors react on requests from proposers. By holding the current state of accepted proposals, the acceptors collectively provide a distributed, fault-tolerant memory for the consensus. In essence, a majority of acceptors together 'know' whether an agreement is already achieved, while the proposers are necessary to trigger the consensus process and to 'read' the distributed memory.

Each round is marked by a distinct round number r. Round numbers are used as a mean of decentralized tokens. The protocol does not limit the number of concurrent proposers: There may be multiple proposers at the same time with different round numbers r. The proposer with the highest r holds the token for achieving consensus. Only messages with the highest round number ever seen by each acceptor, will be processed by that acceptor. All others will be ignored. If at any round, a majority of the acceptors accepted a proposal with value v, it will again be chosen by all subsequent rounds. This ensures the *validity* and *integrity* properties.

Alg. 1, 2, and 3 depict the protocols of the proposer, acceptor, and learner, respectively. The algorithm can be split into two phases: (1) an information gathering phase to check whether there was already an agreement in previous rounds, and (2) a consolidation phase to distribute the consensus to a majority of acceptors and thereby to agree on the decision. In the best case, consensus may be achieved in a single round. In the worst case, the decision may be arbitrarily long delayed by interleaving proposers with successively increasing round numbers (token stealing by each other).

2) Information gathering phase: A proposer starts a new round (lines 1–3 of Alg. 1) by selecting a round number r greater than any round number seen before. At start time, an arbitrary round number is chosen. The only restriction on round numbers is that they must be unique across all possible proposers. This can be achieved, for example, by appending the proposer's identifier. If any new round number happens to be smaller than an earlier one, the round will be detected as outdated and will be ignored.

The proposer sends its round number with a *prepare(r)* message to the acceptors and starts a timeout (timeouts are not shown in the algorithms). If it does not get an *ack* message from a majority of the acceptors within the timeout, it starts from the beginning with a higher round number and retries with a slightly increased timeout. The timeout implements an eventually perfect failure detector $\diamond\mathcal{P}$ on an arbitrary majority of acceptors.

When an acceptor receives a *prepare(r)* message (lines 3–6 of Alg. 2), it checks whether the given round r is newer than any previously seen round. If the received r is greater, the acceptor memorizes the round and acknowledges with *ack(r, v, $r_{accepted}$)* where v is the value accepted previously in round $r_{accepted}$.

Note that a proposed value v may be accepted several times by an acceptor in different rounds. If the round number r is outdated, the acceptor does nothing. Alternatively, the acceptor may send *nack(r, $r_{accepted}$)* to help the proposer to quickly find a higher number for a new round (this improvement is not shown in the algorithms).

3) Consolidation phase: After collecting a majority of *ack* messages, the proposer checks for the latest value that was accepted by an acceptor (lines 4–9 of Alg. 1). If it is still the initial $\perp$, the proposer chooses a value by itself, otherwise it takes the latest accepted value v_j. The proposer then sends an *accept(r, v_j)* request to the acceptors.

An acceptor receiving an *accept(r, v_j)* request checks the round. If it is the latest one, it updates its local state and confirms the accept request with *accepted(r, v)* to the learners (lines 7–11 of Alg. 2). Otherwise the acceptor does nothing or sends *naccepted()* to the proposer.

When a learner receives *accepted(r, v)* messages from a majority of the acceptors, the consensus is finished with value v.

4) Discussion: When a proposer crashes, any other process (or even multiple processes) may take the role of a proposer. The new proposer(s) may retrieve the so far achieved consensus (if any) from the acceptors by triggering a new round.

Since the acceptors have no indication on whether a consensus has been achieved already, they must run forever, always being prepared to take new *accept(r,w)* messages from other proposers. When a new *accept(r,w)* with a higher round number r comes in, they are obliged to accept and store the new value w. As an improvement, the application may decide that a consensus was achieved and consumed and hence the acceptors may be terminated.

B. Paxos Commit

Gray and Lamport [9] describe a commit protocol based on Paxos Consensus. Instead of using a simple version with a single Paxos Consensus as a stable storage, they propose a variant that needs more messages but one less message delay.

It performs a Paxos Consensus for each item (TP) involved in the transaction.

In the simple variant, the transaction manager (TM) is responsible to make the decision. It works as follows: The TM asks all TPs whether they are prepared to commit the requested transaction and TPs answer with either prepared or abort. If all TPs are prepared, the TM initiates a Paxos Consensus and takes the role of a proposer by sending *accept(prepared)* to the acceptors, otherwise by sending *accept(abort)*. The acceptors answer *accepted* and on a majority of such answers the TM sends the final decision (commit or abort) to all TPs for execution. This procedure involves 5 message delays.

The Paxos Commit proposed in [9] needs one fewer message delay. It does so with a separate Paxos Consensus instance for each TP. As before, the TM asks all TPs whether they are prepared to commit the requested transaction. This time, however, the TPs do not reply to the TM directly, but initiate a Paxos Consensus for their decision by taking the role of a proposer and sending their proposal *accept(prepared)* or *accept(abort)* to the acceptors for stable storage. After consensus is achieved, they reply with the outcome to the TM in its role as a learner, which then combines the results and sends the final decision to all TPs for execution. This requires 4 message delays and $N(2F+3)-1$ messages for N TPs, and $2F+1$ acceptors.

If the TM or a TP fails in the decision process, any replicated transaction manager (RTM) may read the decision from the acceptors, or propose to abort if there was no consensus yet.

V. Transactions in Scalaris

Scalaris supports transactional semantics. A client connected to the system can issue a sequence of operations including reads and writes within a transactional context, i.e. *begin trans* ... *end trans*. This sequence of operations is executed by a local transaction manager TM associated with the overlay node to which the client is connected. The transaction will appear to be executed atomically if successful, or not executed at all if the transaction aborts.

A. System Architecture

Transactions in Scalaris are executed optimistically. This implies that each transaction is executed completely locally at the client in a read-phase. If the read phase is successful the TM tries to commit the transaction permanently in a commit phase, and permanently stores the modified data at the responsible overlay nodes. Concurrency control is performed as part of this latter phase. A transaction t will abort only if: (1) other transactions hold the majority of locks of some overlapping data items (simultaneous validation); or (2) other successful transactions have already modified data that is accessed in transaction t (version conflict).

Each item is assigned a version number. Read/write operations work on a majority of replicas to obtain the highest version number and thereby the latest value. A read operation selects the data value with highest version number, and a write operation increments the highest version number of the item.

The commit phase employs an adapted version of the Paxos atomic commit protocol [9], which is non-blocking. In contrast to the 3-Phase-Commit protocol used in distributed database systems, the Paxos Commit protocol still works in the majority part of a network that became partitioned due to some network failure. It employs a group of replicated transaction managers (RTMs) rather than a single transaction manager. Together they form a set of acceptors with the TM acting as the leader.

B. Transaction Validation with Paxos Commit

Scalaris executes the following four steps in the failure-free case (Fig. 1).

1) Prerequisites: For a fast transaction validation, each node in the overlay permanently maintains a list of $r-1$ other nodes, that can be used as *Replicated Transaction Managers (RTMs)*. The location of these nodes could be according to the scheme of symmetric replication. Once these nodes are located, they are maintained through the use of failure detection.

Step 1. The client contacts an arbitrary node in the Scalaris ring with a transaction log (*translog*) of read and write operations for the validation phase. This node becomes the *Transaction Manager (TM)*. The TM chooses a *transaction identifier (Tid)* and a *Paxos Consensus identifier (P_i)* for each replica of each item. It sends an *init_RTM* message with the translog, the Tid, all P_i, and the addresses of all RTMs to each RTM. Additionally, the TM sends to all *Transaction Participants (TP)* an *init_TP* message with the translog, Tid, RTMs, and the individual P_i for each TP.

Step 2. Each TP initiates a Fast Paxos Consensus with the received P_i. Each TP proposes either *prepared* or *abort* with an *accept* message to the acceptors according to its local validation strategy (see later).
As the TP is the only initial proposer, it uses the lowest round number by default and thereby skips the information gathering phase ('Fast Paxos Consensus'). The proposal is sent to the TM and RTMs. If the TP decided *prepared* it locks its replica.
When a TM or RTM receives an accept message from a TP, it also learns the address of the TP to be used later in the protocol.

Step 3. The TM will take the role of a learner in each consensus instance. To allow the TM to calculate each consensus instance, each RTM sends a list of *accepted* messages to the TM. As soon as the TM received a majority of *accepted* messages for a given consensus instance P_i it decides on i.

Step 4. The TM will decide the transaction to *commit* if for each item a majority of the consensus instances have decided prepared, otherwise it will decide *abort*.
After having received the decision from the TM, the TPs execute the changes, release the locks and finish.

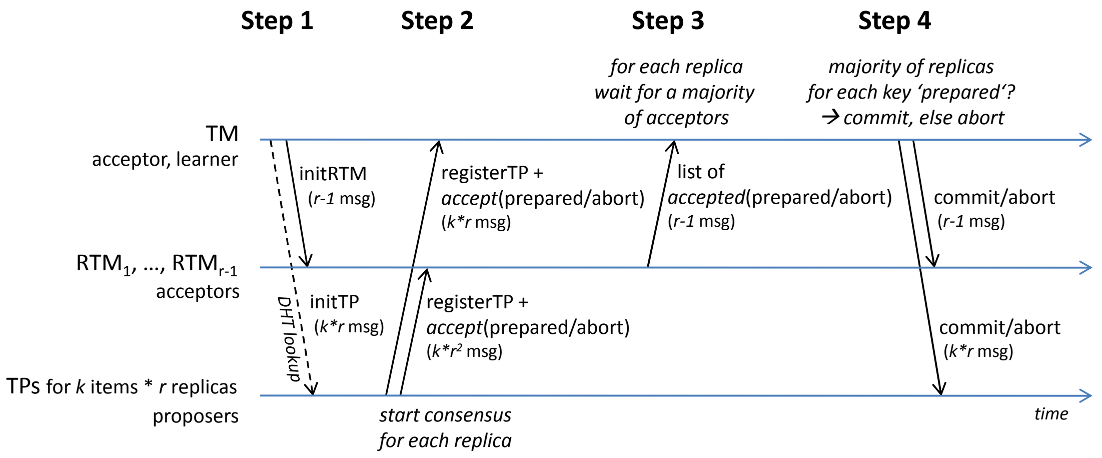

Step 1
Step 2
Step 3
Step 4
for each replica wait for a majority of acceptors
majority of replicas for each key 'prepared'? → commit, else abort
TM
acceptor, learner
initRTM
(r-1 msg)
registerTP + accept(prepared/abort)
(k*r msg)
list of accepted(prepared/abort)
(r-1 msg)
commit/abort
(r-1 msg)
RTM1, ..., RTMr-1
acceptors
DHT lookup
initTP
(k*r msg)
registerTP + accept(prepared/abort)
(k*r² msg)
commit/abort
(k*r msg)
TPs for k items * r replicas
proposers
start consensus for each replica
time

Fig. 1. Timeline diagram of a Scalaris commit.

2) Discussion: As a precondition, we assume that a majority of RTMs plus TM and a majority of replicas for each item are correct. The following failures may happen:

When the TM fails, any RTM may take its role by initiating a new round for every Paxos Consensus involved. In Scalaris, the RTMs' failure detectors have different timeouts, so that multiple RTMs will never compete for leadership and no explicit leader election algorithm is necessary. The new TM is able to continue with the protocol, because the current status on the consensus is safely stored at the RTMs (acceptors).

When an RTM fails, the protocol continues with the rest of the RTMs.

When a TP fails in step 2, the TM or some RTM does not receive an *accept* message from the TP within the specified timeout. The TM or RTM then takes the role of a proposer and proposes *abort* for the corresponding consensus instance with a round number > 1, if no consensus was already achieved before the TP crashed. Until only a minority of the Paxos Consensus for the replicas of a given item votes *abort* and a majority of them votes *prepared* the transaction still can be committed. This can be safely done, as in contrast to Paxos Commit, we operate on replicated items.

C. Working Phase: Building a Translog in Scalaris

We now describe the working phase in which Scalaris builds a translog with all items that are to be updated in an atomic operation. Alg. 4 shows an example of a client code for a money transfer from bank account A to account B. The money transfer should be executed atomically—if the balance in account A allows to. In the example, each account is replicated over three keys $key_{A_1}, \dots, key_{A_3}$ and $key_{B_1}, \dots, key_{B_3}$. Fig. 2 shows the corresponding Scalaris ring with the replicas.

The client code shown in Alg. 4 is formulated in the functional programming language Erlang [2]. It works as follows. First, it defines a function F, that will perform the working phase of the transaction (lines 2-12). It then executes this function to retrieve a transaction log (line 13) and thereafter attempts to validate it by calling scalaris:commit() on the outcome of the working phase (line 14).

Algorithm 4 Example of a Scalaris transaction in Erlang.

```
 1: my_transaction() ->
 2:   F = fun (TransLog) ->
 3:     {X, TL1} = scalaris:read(TransLog, "Acc A"),
 4:     {Y, TL2} = scalaris:read(TL1, "Acc B"),
 5:     if X > 100 ->
 6:       TL3 = scalaris:write(TL2, "Acc A", X - 100),
 7:       TL4 = scalaris:write(TL3, "Acc B", Y + 100),
 8:       {ok, TL4};
 9:     true ->
10:       {ok, TL2};
11:     end
12:   end,
13:   MyTransLog = F(EmptyTransLog),
14:   Result = scalaris:commit(MyTransLog) .
```

The working phase is 'read only' and does not modify any values or locks. It stores only the relevant data for each accessed key in the transaction log *translog*. Each translog entry is a 5-tuple consisting of: (1) the performed operation, (2) the key involved, (3) a status flag indicating success or failure, (4) the corresponding value, and (5) the corresponding version.

A *read* request for a key k triggers a quorum read on the replicas, if k is not yet included in the translog. It returns the read value and the accordingly updated translog as a tuple.

A *write* request for a key k first triggers a quorum read on the replicas, if k is not yet included in the transaction log. Then a new translog entry with the incremented version number and the new value is created or updated accordingly.

The quorum reads for read and write operations require DHT lookups with $O(\log n)$ hops. If a quorum read fails, this is recorded in the corresponding status flag in the translog. If any status flag in the translog is *failed*, the whole transaction will be aborted.

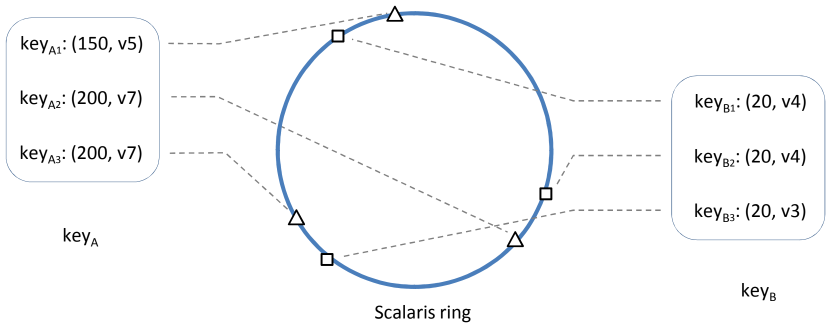


Fig. 2. Scalaris ring with two items key$_A$ and key$_B$.

D. Using the Translog in the Validation Phase

Based on the commit protocol presented in Sec. V-B we now describe the validation strategy in more detail. We show how Scalaris places locks and decides according to the translog.

A TP receives in step 1 of Fig. 1 the corresponding translog entry. To choose between the proposals *prepared* and *abort* it checks the following constraints:

- Is the version number still valid?
  For reads: Is the local version number in the data store the same as the one listed in the translog entry?
  For writes: Is the local version number in the data store one less than the version number stored in the translog?
- Is the lock of the key available?
  For reads: Is no write lock set?
  For writes: Is neither a read lock, nor a write lock set?

If both checks are successful, the TP proposes *prepared* and increments for reads the read lock counter and for writes it sets the write lock. Otherwise it proposes *abort*.

When a TP receives a write commit in step 4 of Fig. 1, it writes the value and version number from the translog into the key.

For read and write operations, independent of commit or abort, the TP releases the locks.

VI. Evaluation

For globally distributed structured overlay systems, latency is an important issue. To reduce the latency in our majority based system, we may assign a majority of the replicas of an item to nodes near the main popularity of that item. This is possible using Chord# [17] as an overlay, as it allows to arbitrarily assign nodes to ranges of keys and as it does not use hashing but keeps the keys in lexicographical order in the ring.

1) The latency-critical path: In step 1 of our commit protocol, an initialization message is send to each RTM and TP (see Fig. 1). Each TP immediately responds with its *accept* message to the TM and RTMs. So, some *accept* message may arrive at an RTM earlier than the corresponding initialization message. This is not a problem, as the RTM will record it and assign it later via the given transaction and consensus identifiers. Similarly in the case of *accepted* messages from RTMs (step 3) that may arrive earlier at the TM than the *accept* messages from the TPs sent in step 2.

In step 3, each RTM collects an *accept* message for each consensus (each TP) and sends a list of *accepted* in a single message to the TP. While this protocol is optimal with respect to the number of messages sent, the overall latency can be reduced by sending each *accepted* message immediately after receipt of the corresponding *accept*. Then the TM must await a consensus for a majority of the P_i for each item, independent from which RTMs it came. Progress between step 2 and 4 depends on the m lowest latency paths from TPs (via RTMs) to the TM for each item, where $m = r/2+1$ is the size of the majority set.

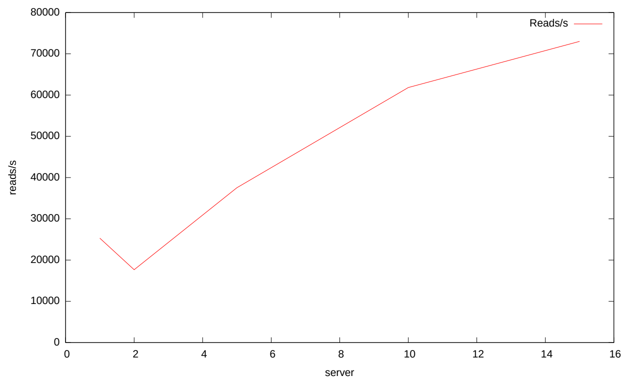
2) Empirical Results: We compared the performance of simple quorums reads with full transactions on an Intel cluster with 16 nodes. Each node has two Dual-Core Intel Xeons (4 cores in total) running at 2.66 GHz and 8 GB of main memory. The nodes are connected via GigE. On each server we ran s Scalaris nodes distributed over v Erlang virtual machine. We used a replication degree of four, i.e. there are four copies of each key-value pair. For generating load, we started c clients in each Erlang VM and each client performed the function under test i times. We ran the tests with various combinations for (s,v,c,i). The graphs in Fig. 3 show the aggregated performance over all clients and the number of clients per VM of the best parameter combinations. The best parameter settings usually used 1 VM per server with 16 or 32 Scalaris nodes.

The left graph in Fig. 3 shows the throughput for quorum reads. The maximum of 73,000 lookups is achieved with 15 servers. As the quorum reads are dominated by the lookup, which scales with $\log N$, the curve does not scale linearly. Two servers achieve a lower read performance than one because of the additional TCP overhead.

The right graph in Fig. 3 shows the performance of read-modify-write transactions with Paxos. 15 servers are capable of handling almost 14,000 transactions per second. More importantly, the curve scales almost linearly with an increasing number of servers.

VII. Conclusion

We presented an atomic transaction protocol that has been efficiently embedded into a DHT and uses four communication



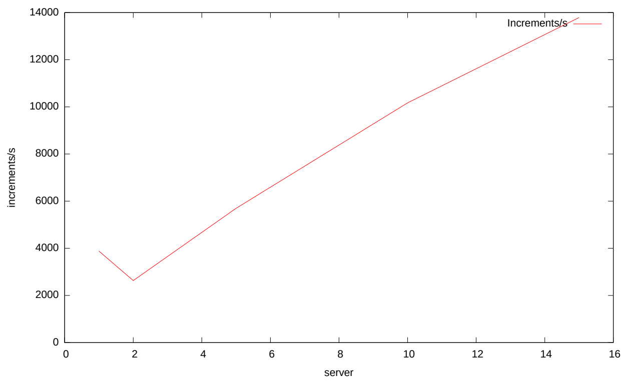


Fig. 3. Performance of quorum reads (left) and transactions with Paxos (right).

steps only. It makes progress as long as a majority of TPs for each item and a majority of RTMs (including the TM) are correct (non-failing).

The transaction protocol was used to implement Scalaris [16], a fault-tolerant key/value store with replicated items on a DHT. The DHT ensures scalability while the enhanced Paxos commit protocol provides data consistency. The implementation comprises a total of 9,700 lines of Erlang code: 7,000 for the P2P layer with replication and basic system infrastructure and 2,700 lines for the transaction layer.

References

[1] L. Alima, S. El-Ansary, P. Brand and S. Haridi. DKS(N,k,f): A family of low-communication, scalable and fault-tolerant infrastructures for P2P applications. *Workshop on Global and P2P Computing*, CCGRID 2003, May 2003.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006.

[4] T.D. Chandra, S. Toueg. Unreliable failure detector for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: Amazon's highly available key-value store. *SOSP*, Oct. 2007.

[6] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[7] A. Ghodsi. Distributed k-ary system: Algorithms for distributed hash tables. *PhD Thesis*, Royal Institute of Technology, 2006.

[8] A. Ghodsi, L. Alima, S. Haridi. Symmetric replication for structured Peer-to-Peer systems. *DBISP2P*, Aug. 2005.

[9] J. Gray, L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[10] R. Guerraoui, L. Rodrigues. Introduction to reliable distributed programming. Springer-Verlag, 2006.

[11] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. ESOA, 2005.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[13] A. Muthitacharoen, S. Gilbert, R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, 2005.

[14] M. Moser, S. Haridi. Atomic commitment in transactional DHTs. 1st CoreGRID Symposium, Aug. 2007.

[15] R. D. Prisco, B. W. Lampson, N. A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[16] T. Schütt, F. Schintke, A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. *ACM SIGPLAN Erlang Workshop*. 2008.

[17] T. Schütt, F. Schintke, A. Reinefeld. Structured overlay without consistent hashing: Empirical results. *GP2PC'06*, May. 2006.

[18] T. M. Shafaat, A. Ghodsi, S. Haridi. Handling Network Partitions and Mergers in Structured Overlay Networks. P2P 2007.

[19] T.M. Shafaat, M. Moser, T. Schütt, A. Reinefeld, A. Ghodsi, S. Haridi. Key-based consistency and availability in structured overlay networks. Infoscale, June 2008.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A scalable Peer-to-Peer lookup service for Internet applications. ACM SIGCOMM 2001.