Towards the Future Internet
G. Tselentis et al. (Eds.)
IOS Press, 2009

doi:10.3233/978-1-60750-007-0-148

A Scalable, Transactional Data Store for Future Internet Services*

Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Seif Haridi

{reinefeld,schintke,schuett}@zib.de
Zuse Institute Berlin and onScale solutions
and
haridi@kth.se
Royal Institute of Technology, Sweden

Abstract. Future Internet services require access to large volumes of dynamically changing data records that are spread across different locations. With thousands or millions of distributed nodes storing the data, node crashes or temporary network failures are normal rather than exceptions and it is therefore important to hide failures from the application.

We suggest to use peer-to-peer (P2P) protocols to provide self-management among peers. However, today's P2P protocols are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key-value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol. As a proof of concept, we implemented a simple Wikipedia clone with Scalaris which outperforms the public Wikipedia with just a few servers.

1 Introduction

Web 2.0, that is, the Internet as an information society platform supporting business, recreation and knowledge exchange, initiated a business revolution. Service providers offer Internet services for shopping (Amazon, eBay), online banking, information (Google, Flickr, Wikipedia), social networking (MySpace, Facebook), and recreation (Second Life, online games). In our information society, Web 2.0 services are no longer just nice to have, but customers depend on their continuous availability, regardless of time and space. A typical trend is illustrated by Wikipedia where users are also providers of information. This implies that its underlying data store is updated continously from multiple sources.

* This work was partly funded by the EU projects SELFMAN under grant IST-34084 and the EU project XtreemOS under grant IST-33576.

How to cope with such strong demands, especially in case of interactive community services that cannot be simply replicated? All users access the same Wikipedia, meet in the same Second Life environment and want to discuss with others via Twitter. Even the shortest interruption, caused by system downtime or network partitioning may cause huge losses in reputation and revenue. Web 2.0 services are not just an added value, but they must be dependable. Apart from 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Indeed, scalability is a key challenge. In addition to scalability and availability any global service to be affordable, somehow requires the system to be self managing (see sidebar).

Availability is the proportion of time a system is in a functioning condition. More formally, availability is the ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and down time. Availability is often specified in a logarithmic unit called "nines" which corresponds roughly to a number of nines following the decimal point. "Six nines", for example, denote an availability of 0.999999, allowing a maximum downtime of 31 seconds per year.

Scalability refers to the capability of a system to increase the total throughput under an increased load when resources are added. A scalable database management system is one that can be upgraded to process more transactions by adding new processors, devices and storage, and which can be upgraded easily and transparently without service interrupt.

Self Management refers to the ability of a system to adjust to changing operating conditions and requirements without human intervention at runtime. Self Management includes self configuration, self healing and self tuning.

Our Scalaris system, described below, provides a comprehensive solution for self managing and scalable data management. Scalaris is a transactional key-value store that runs over multiple data centers as well as on peer-to-peer nodes. We expect Scalaris and similar systems to become an important core service of future Cloud Computing environments.

As a common key aspect, all Web 2.0 services have to deal with concurrent data updates. Typical examples are checking the availability of products and their prices, purchasing items and putting them into virtual shopping carts, and updating the state in multi-player online games. Clearly, many of these data operations have to be atomic, consistent, isolated and durable (so-called ACID properties). Traditional centralized database systems are ill-suited for this task, sooner or later they become a bottleneck for business workflow. Rather, a scalable, transactional data store like Scalaris is what is needed.

In this paper, we present the overall system architecture of Scalaris. We have implemented the core data service of Wikipedia using Scalaris. Its scalability and self-* capabilities were demonstrated in the IEEE Scalable Computing Challenge

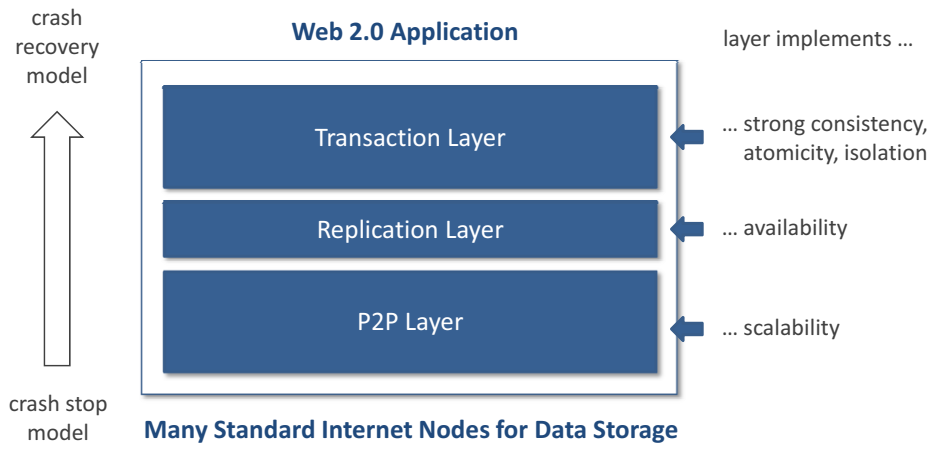

Fig. 1: Scalaris system architecture.

2008, where Scalaris won the 1^{st} price (www.ieeetcsc.org/scale2008). Talks on Scalaris were given at the the Google Scalability Conference 2008 [19] and the Erlang eXchange 2008.

The paper is organized as follows. The following Section provides an overview on Scalaris' system architecture, Section 3 describes its self-management features and Section 4 gives further details on the implementation. In Section 5 we demonstrate how Scalaris can be used for implementing Web 2.0 services. As a proof-of-concept, we have chosen a simple Wikipedia clone; performance results are given in Section 6.

2 Scalaris

As part of the EU funded SELFMAN project we set out to build a distributed key/value store capable of serving thousands or even millions of concurrent data accesses per second. Providing strong data consistency in the face of node crashes and hefty concurrent data updates was one of our major goals.

With Scalaris, we do not attempt to replace current database management systems with their general, full-fledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multi-player online games. Our system consists of three layers:

– At the bottom, an enhanced structured overlay network, with logarithmic routing performance, provides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographic ordering instead of random hashing enables control of data placement which is necessary for low latency access in multi-datacenter environments.

- The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures.
- The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses an optimistic concurrency control strategy and a fast non-blocking commit protocol with low communication overhead. This protocol has been optimally embedded in the overlay network.

As illustrated in Fig. 1, these three layers together provide a scalable and highly available distributed key/value store which serves as a core building block for many Web 2.0 applications as well as other global services. The following sections describe the layers in more detail.

2.1 P2P Overlay

At the bottom layer, we use the structured overlay protocol Chord$^{\#}$ [17,18] for storing and retrieving key-value pairs in nodes (peers) that are arranged in a virtual ring. This ring defines a key space where all data items can be stored according to the associated key. In our case we assume that any key is an arbitrarily long string of characters, therefore the key space is infinite. Nodes are 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 traditional Distributed Hash Tables (DHTs) like Chord [21], Kademlia [12] and Pastry [15], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries, and control over the placement of data on the ring structure. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table jump over an exponentially increasing number of nodes in the ring. This finger placement will yield uniform in-/out-degree of the overlay network and thus avoids hotspots.

Chord$^{\#}$ uses the following algorithm for computing the fingers in the routing table (the infix operator $x\,.\,y$ retrieves y from the routing table of a node x):

$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1}\,.\,\mathit{finger}_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node, listed in its $(i-1)^{th}$ finger, for the node at which its $(i-1)^{th}$ finger refers to. In general, at any node, the fingers at level i are set to the neighbor's finger at the preceding level $i-1$. At the lowest level, the fingers point to the direct successor. The resulting structure is similar to a skiplist, but the fingers are computed deterministically without any probabilistic component and each node has its individual exponentially spaced fingers. The fingers are maintained by a periodic stabilization algorithm according to the above formula.

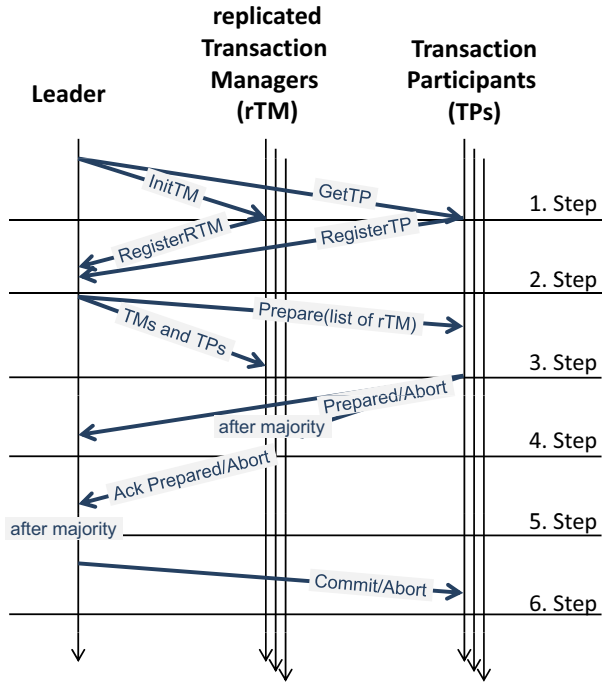

Fig. 2: Adapted Paxos used in Scalaris.

Compared to Chord [21], Chord$^{\#}$ does the routing in the *node space* rather than in the *key space*. This finger placement has three advantages over that of Chord: First, it naturally works with any type of keys as long as a total order over the keys is defined, and second, finger maintenance is cheaper [17], requiring just one hop instead of a full logarithmic search (as in Chord). To support logarithmic routing performance in skewed key distributions while nodes are arbitrarily placed in the key space—which we have to in our scenario—the third and probably most important difference becomes our trump card: the incoming routing links (fingers) will still be evenly distributed across all nodes. This prevents nodes from becoming hot spots and ensures continuous progress when routing.

2.2 Replication and Transaction Layer

The scheme described so far provides scalable access to distributed key/value pairs. To additionally tolerate node failures, we replicate all key/value pairs over r nodes using symmetric replication [5]. Basically each key is mapped by a globally known function to a set of keys $\{k_1, \ldots, k_r\}$ and the item is replicated according to those keys. Read and write operations are performed on a majority

of the replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes. This scheme is shown to provide key consistency for data lookups under realistic networking conditions [20]. For repairing the replication degree of items, nodes have to read the missing data from a majority of replicas. This is necessary to guarantee strong data consistency.

The system 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 are 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.

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 try to commit changes on some overlapping data items simultaneously; or (2) other successful transactions have already modified data that is accessed in transaction t.

Each item is assigned a version number. Read/write operation works on a majority of replicas to obtain the highest version number. 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 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 (rTM) rather than a single transaction manager. Together they form a set of acceptors with the TM acting as the leader.

The commit is basically divided into two phases, the validation phase and the consensus phase. During the validation phase the replicated transaction managers rTM are initialized, and the updated data items together with references to the rTM are sent to the nodes responsible for the data items in a Prepare message. These latter nodes are called transaction participants TPs.

Each TP proposes 'prepared' or 'abort' in a fast Paxos consensus round with the acceptor set. As each acceptor collects votes from a majority of replicas for each item, it will be able to decide on a commit/abort for the whole transaction. For details see [13,20]. This scheme favors atomicity over availability. It always requires a majority of nodes to be available for the read and commit phase. This policy distinguishes Scalaris from other distributed key-value stores, like e.g. Dynamo [3].

3 Self-Management

For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. Scalaris greatly reduces the operation cost with its built-in self* properties:

- *Self healing:* Scalaris continuously monitors the hosts it is running on. When it detects a node crash, it immediately repairs the overlay network and the database. Management tasks such as adding or removing hosts require minimal human intervention.
- *Self tuning:* Scalaris monitors the nodes' workload and autonomously moves items to distribute the load evenly over the system in order to improve the response time of the system. When deploying Scalaris over multiple datacenters, these algorithms are used to place frequently accessed items nearby the users.

These protection schemes do not only help in stress situations, but they also monitor and pro-actively repair the system before any service interruption might occur. With traditional database systems these operations require human interference which is error prone and costly. When using Scalaris, fewer system administrators can operate much larger installations compared to legacy databases.

4 Implementation

Implementing distributed algorithms correctly is a difficult and tedious task, especially when using imperative programming languages and multi-threading with a shared state concurrency model. The resulting code is often lengthy and error-prone, because large parts of the code deal with shared objects [22] and with exception handing such as node or network failures.

For this reason, message passing as in the *actor model* [7] is becoming the accepted paradigm for describing and reasoning about distributed algorithms [6]. Scalaris was also developed according to this model. The basic primitives in this model are actors and messages. Every actor has a state, can send messages, act upon messages and spawn new actors.

These primitives are easily mapped to Erlang processes and messages [1]. The close relationship between the specification and the programming language allows a smooth transition from the theoretical model to prototypes and eventually to a complete system.

Our Erlang implementation of Scalaris comprises eight major components with a total of 11,000 lines of code: 7,000 for the P2P layer with replication and basic system infrastructure, 2,700 lines for the transaction layer, and 1,300 lines for the Wikipedia infrastructure. Each Scalaris node is organized into the following components:

- The *Failure Detector* supervises other peers and notifies subscribers of remote node failures.

- The *Configuration Store* provides access to the current configuration and allows modifications of various system parameters.
- The *Key Holder* stores the identifier of the node in the overlay.
- The *Statistics Collector* collects statistics and forwards them to central statistic servers.
- The *Chord#* *Node* component is composed of subcomponents for overlay maintenance and overlay routing. It maintains, among other things, the successor list and the routing table. It provides the functionality of the structured overlay layer.
- The *Database* stores the key-value pairs of this node. The current implementation uses an in-memory dictionary, but disk store based on DETS or Mnesia could also be used.
- The *Transaction Manager* runs the transaction protocols.
- The *Replica Repair* maintains the replication degree of items.

The processes are organized in an Erlang OTP supervisor tree. When any of the slaves crashes, it is restarted by the Erlang supervisor. When either of the Chord# Node or the Database component fails, the other is explicitly killed and both are restarted to ensure consistency. This is equivalent to a new node joining the system.

5 Deployment: Wikipedia on Scalaris

As a challenging benchmark for Scalaris, we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". Wikipedia runs on three sites. The main one in Tampa is organized in three layers, the proxy server layer, the web server layer, and the MySQL database layer. The proxy layer serves as a cache for recent requests, and the web server layer runs the application logic and issues requests to the database layer. Wikipedia handles about 50,000 requests per second, from which 48,000 are cache hits in the proxy server layer and 2,000 are processed by the database layer. The proxy and the web server layers are embarrassingly parallel and therefore trivial to scale. From a scalability point of view, only the database layer is challenging.

Our implementation uses Scalaris to replace the database layer. This enables us to run Wikipedia on geographically distributed sites and to scale to almost any number of hosts, as shown in the evaluation section. Our Wikipedia implementation inherits all the favorable properties of Scalaris, such as scalability and self management.

Instead of using a relational database, we map the Wikipedia content to our Scalaris key/value store [14]. We use the following mappings, using prefixes in the keys to avoid name clashes.

	key	**value**
page content	title	list of Wikitext for all versions
backlinks	title	list of titles
categories	category name	list of titles

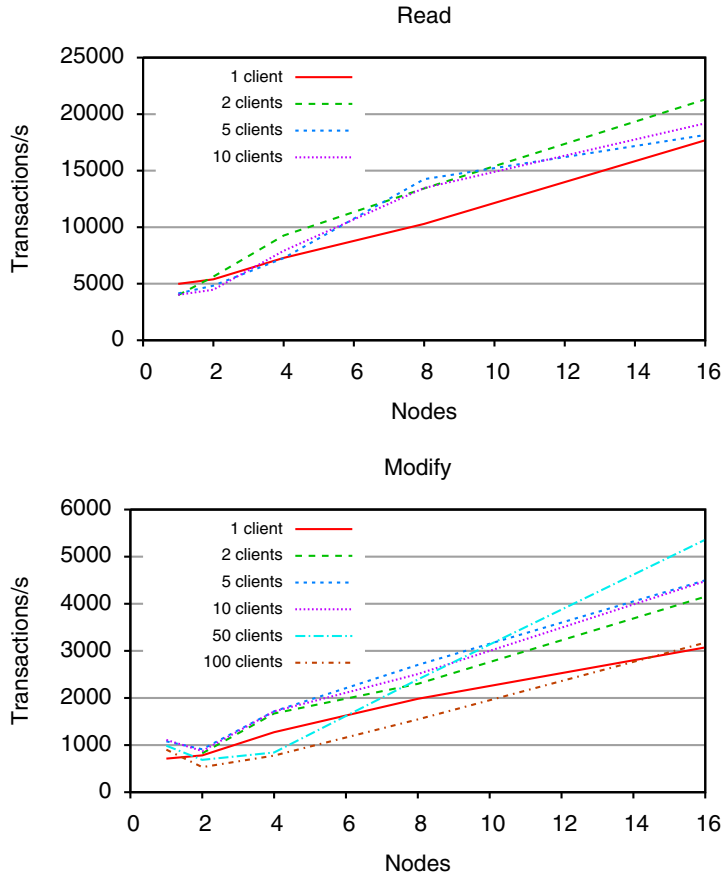

Fig. 3: Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes.

On a page update a transaction across all affected keys (content, backlinks, and categories) and their replicas is triggered.

6 Evaluation

We tested the performance of Scalaris 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.

Figure 3 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 3.a, each thread reads a key 2000 times while the modify benchmarks in Fig. 3.b 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. 3.a), 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. 3.b) 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.

7 Summary

Scalaris provides a scalable and self managing transactional key-value store. We have implemented Wikipedia using Scalaris. Its scalability and self* capabilities were demonstrated in the IEEE Scalable Computing Challenge 2008, where Scalaris won the 1st prize.

Compared to other data services, Scalaris has significantly lower operating costs and is self-managing. Scalaris and similar systems will be an important building block for Web 2.0 services and future Cloud Computing environments.

While Wikipedia served here as a first demonstrator to show the potential of Scalaris, we envisage a large variety of commercial Web 2.0 applications ranging from e-commerce and social networks to infrastructure services for maintaining server farms. The Scalaris code is open source (scalaris.googlecode.com).

Acknowledgements

Many thanks to Nico Kruber, Monika Moser, and Stefan Plantikow who implemented parts of Scalaris. Also thanks to Ali Ghodsi, Thallat Shafaat, and Joe Armstrong for their support and many discussions.

References

1. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007
2. R. Baldoni, L. Querzoni, A. Virgillito, R. Jiménez-Peris, and M. Patiño-Martínez. *Dynamic Quorums for DHT-based P2P Networks.* NCA, pp. 91–100, 2005.
3. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.
4. JJ Furman, J. S. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *SIGMOD 2008*, Jun. 2008.
5. A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *3rd Intl. Workshop on Databases, Information Systems and P2P Computing*, 2005.
6. R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag 2006.
7. C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.
8. A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. *SIGMOD 2008*, Jun. 2008.
9. L. Lamport. The Part-Time Parliament *ACM Transactions on Computer Systems* 16(2): 133–169, 1998.
10. L. Lamport. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.
11. M. M. Masud and I. Kiringa. *Maintaining consistency in a failure-prone P2P database network during transaction processing.* Proceedings of the 2008 International Workshop on Data management in peer-to-peer systems, pp. 27–34, 2008.
12. P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. *IPTPS 2002*, Mar. 2002.
13. M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

14. S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Oct. 2007.
15. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware 2001*, Nov. 2001.
16. Scalaris code: http://code.google.com/p/scalaris/.
17. T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.
18. T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.
19. T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.
20. T.M. Shafaat, M. Moser, A. Ghodsi, S. Haridi, T. Schütt, and A. Reinefeld. Key-Based Consistency and Availability in Structured Overlay Networks. Third Intl. ICST Conference on Scalable Information Systems, June 2008.
21. I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001. Concepts, Techniques, and Models of Computer Programming
22. P. Van Roy and S. Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, March 2004.