Scalaris: Reliable Transactional P2P Key/Value Store

Web 2.0 Hosting with Erlang and Java

Thorsten Schütt Florian Schintke Alexander Reinefeld

Zuse Institute Berlin and onScale solutions

schuett@zib.de, schintke@zib.de, reinefeld@zib.de

Abstract

We present *Scalaris*, an Erlang implementation of a distributed key/value store. It uses, on top of a structured overlay network, replication for data availability and majority based distributed transactions for data consistency. In combination, this implements the ACID properties on a scalable structured overlay.

By directly mapping the keys to the overlay without hashing, arbitrary key-ranges can be assigned to nodes, thereby allowing a better load-balancing than would be possible with traditional DHTs. Consequently, Scalaris can be tuned for fast data access by taking, e.g. the nodes' geographic location or the regional popularity of certain keys into account. This improves Scalaris' lookup speed in datacenter or cloud computing environments.

Scalaris is implemented in Erlang. We describe the Erlang software architecture, including the transactional Java interface to access Scalaris.

Additionally, we present a generic design pattern to implement a responsive server in Erlang that serializes update operations on a common state, while concurrently performing fast asynchronous read requests on the same state.

As a proof-of-concept we implemented a simplified Wikipedia frontend and attached it to the Scalaris data store backend. Wikipedia is a challenging application. It requires—besides thousands of concurrent read requests per seconds—serialized, consistent write operations. For Wikipedia's category and backlink pages, keys must be consistently changed within transactions. We discuss how these features are implemented in Scalaris and show its performance.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed databases; C.2.4 [*Distributed Systems*]: Distributed applications; D.2.11 [*Software architectures*]: Patterns; E.1 [*Data structures*]: Distributed data structures

General Terms Algorithms, Design, Languages, Management, Reliability

Keywords Wikipedia, Peer-to-Peer, transactions, key/value store

1. Introduction

Global e-commerce platforms require highly concurrent access to distributed data. Millions of read operations must be served within milliseconds even though there are concurrent write accesses. Enterprises like Amazon, eBay, Myspace, YouTube, or Google solve this problems by operating tens or hundreds of thousands of servers in distributed datacenters. At this scale, components fail continuously and it is difficult to maintain a consistent state while hiding failures from the application.

Peer-to-peer protocols provide self-management among peers, but they 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, both of them implemented in Erlang with an application interface to Java. To prove our concept, we implemented a simple Wikipedia clone on Scalaris which performs several thousand transactions per second on just a few servers.

In this paper, we give details on the design and implementation of Scalaris. We highlight Erlang specific topics and illustrate algorithm details with code samples. Talks on Scalaris were given at the IEEE International Scalable Computing Challenge 2008[1], the Google Scalability Conference 2008 [15] and the Erlang eXchange 2008.

The paper is organized as follows. After a brief review of related work we describe the overall system architecture and then discuss implementation aspects in Section 4. In Section 5, we present a generic design pattern of a responsive, stateful server, which is used in Scalaris. We then present our example application, a distributed Wikipedia clone in Section 6 and we end with a conclusion.

2. Related Work

Scalable, transactional data stores are of key interest to the community and hence there exists a wide variety of related work. Amazon's key/value store Dynamo [3] and its commercial counterpart SimpleDB which is used in the S3 service, are similar to our work, because they are also based on a scalable P2P substrate. But in contrast to Scalaris, they implement only eventual consistency rather than strong consistency. Moreover, Dynamo does not support transactions over multiple items.

The work of Baldoni *et al.* [2] focuses on algorithms for the creation of dynamic quorums in P2P overlays—an issue that is of particular relevance for the transaction layer in Scalaris. They show that in P2P systems the quorum acquisition time and the message latency are more important than the quorum size, which has been

[1] Scalaris won the 1^{st} price at SCALE 2008, www.ieeetcsc.org/scale2008

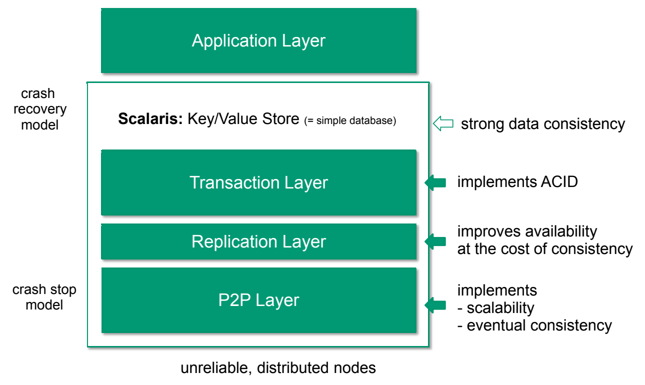

Application Layer
crash recovery model
Scalaris: Key/Value Store (= simple database)
strong data consistency
Transaction Layer
implements ACID
Replication Layer
improves availability at the cost of consistency
crash stop model
P2P Layer
implements - scalability - eventual consistency
unreliable, distributed nodes

Figure 1. Scalaris system architecture.

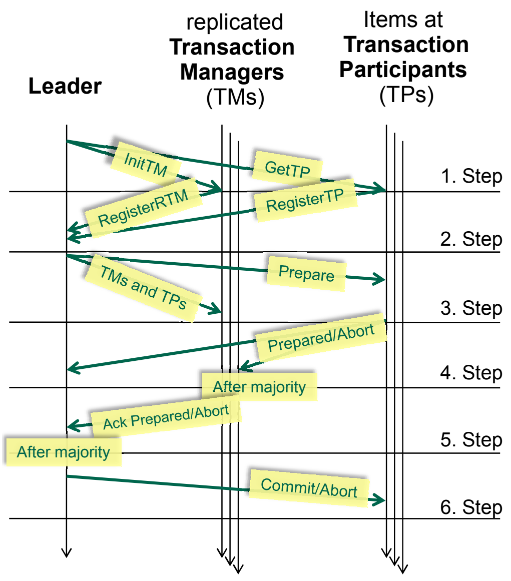


Figure 2. Adapted Paxos used in Scalaris.

traditionally used as a performance metric in distributed systems. This is in line with our results showing that an increasing replication degree r only marginally affects the access time, because the replicas residing in the $\lceil (r+1)/2 \rceil$ fastest nodes take part in the consensus process.

Masud *et al.* [10] also discuss database transactions on structured overlays, but with a focus on the consistent execution of transactions in the presence of failing nodes. They argue that executing transactions over the acquaintances of peers speeds up the transaction time and success rate. Scalaris has a similar concept, but here the peer 'acquaintances' are realized by the load balancer.

With Cassandra [8] and Megastore [4], Facebook and Google recently presented two databases based on the P2P paradigm. Megastore extends Bigtable with support of transactions and multiple indices. Cassandra is more similar to Dynamo as it also provides eventual consistency.

3. System Architecture

Scalaris is a distributed key/value store based on a structured P2P overlay that supports consistent writes. The system comprises three layers (Fig. 1):

- At the bottom, a structured overlay network with logarithmic routing performance builds the basis for the key/value store. In contrast to many other DHTs, our overlay stores the keys in lexicographical order, hence efficient range queries are possible.
- The middle layer implements replication and ACID properties (atomicity, concurrency, isolation, durability) for concurrent write operations. It uses a Paxos consensus protocol [9] which is integrated into the overlay protocol to ensure low communication overhead.
- The top layer hosts the application, a distributed key/value store. This layer can be used as a scalable, fault-tolerant backend for online services for shopping, banking, data sharing, online gaming, or social networks.

Fig. 1 illustrates the three layers. The following sections describe them in more detail.

3.1 P2P Overlay

At the bottom layer, the structured overlay protocol Chord$^{\#}$ [13, 14] is used for storing and retrieving key/value pairs in nodes (peers) that are arranged in a virtual ring. In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to Chord [17], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table cross an exponentially increasing number of nodes in the ring.

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):

$$finger_i = \begin{cases} successor & : i = 0 \\ finger_{i-1} \,.\, finger_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node listed in its $(i-1)^{th}$ finger to which node his $(i-1)^{th}$ finger refers to. In general, the fingers in level i are set to the fingers' neighbors in the next lower level $i-1$. At the lowest level, the fingers point to the direct successors. The resulting structure is similar to a skiplist, but the fingers are computed deterministically without any probabilistic component.

Compared to Chord, Chord$^{\#}$ does the routing in the *node space* rather than the *key space*. This finger placement has two advantages over that of Chord: First, it works with any type of keys as long as a total order over the keys is defined, and second, finger updates are cheaper, because they require just one hop instead of a full search (as in Chord). A proof of Chord$^{\#}$'s logarithmic routing performance can be found in [13].

3.2 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]. 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.

Each item is assigned a version number. Read operations select the item with the highest version number from a majority of the replicas. Thus a single read operation accesses $\lceil (r+1)/2 \rceil$ nodes, which is done in parallel.

Write operations are done with an adapted Paxos atomic commit protocol [11]. In contrast to the 3-Phase-Commit protocol (3PC) used in distributed database systems, the adapted Paxos is non-blocking, because it employs a group of *acceptors* rather than a

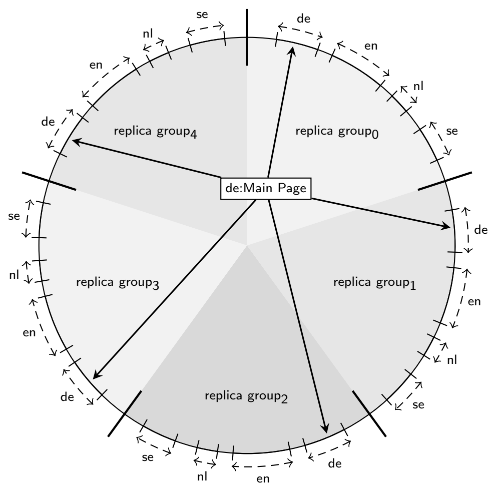


Figure 3. Symmetric replication and multi-datacenter scenario. By assigning the majority of the 'de'-, 'nl'-, and 'se'-replicas to nodes in Europe, latencies can be reduced.

single transaction manager. We select those nodes as acceptors that are responsible for symmetric replication of the transaction manager. The group of acceptors is determined by the transaction manager just before the prepare request is sent to the transaction participants (Fig. 2). This gives a pseudo static group of transaction participants at validation time, which is contacted in parallel.

Write operations and transactions need three phases, including the phase to determine the nodes that participate in the atomic commit. For details see [11, 16].

In Scalaris, the adapted Paxos protocol serves two purposes: First it ensures that all replicas of a *single* key are updated consistently, and second it is used for implementing transactions over *multiple* keys, thereby realizing the ACID properties (atomicity, concurrency, isolation, durability).

3.3 Deployment in Global Datacenters

While we also tested Scalaris on globally distributed servers using PlanetLab[2], its deployment in globally distributed datacenters is more relevant for international service providers. In such scenarios, the latency between the peers is roughly the same and the peers are in general more reliable.

When deploying Scalaris in multi-datacenter environments, a single structured overlay will span over all datacenters. The location of replicas will influence the access latency and thereby the response time perceived by the user. As Chord$^{\#}$ supports explicit load-balancing, it can—besides adapting to e.g. heterogeneous hardware and item popularity—place the replicas in specific centers. A majority of replicas of German Wiki pages, for example, should be placed in European datacenters to reduce the access latency for German users.

Scalaris uses symmetric replication [5]. Here, a key 'de:Main Page' is stored in five different locations in the ring (see Fig. 3). The locations are determined by prefixing the key with '0', '1', ..., '5'. So the key of the third replica is '2de:Main Page' and the third replicas of all German articles will populate a consecutive part of the ring. By influencing the load-balancing strategy we can guarantee this segment to be always hosted in a particular datacenter.

[2] http://www.planet-lab.org

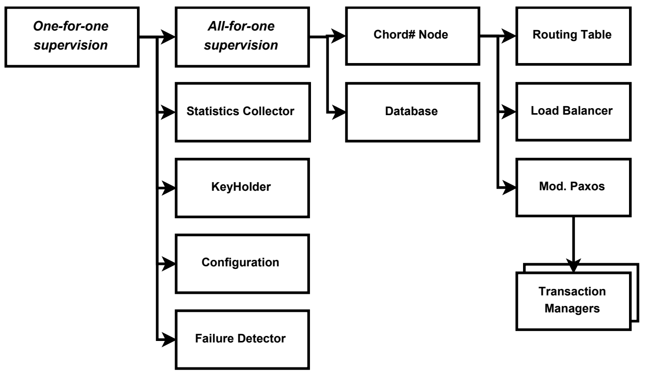


Figure 4. Supervisor tree of a Scalaris node. Each box represents one process.

4. Erlang Implementation

The *actor model* [7] is a popular model for designing and implementing parallel or distributed algorithms. It is often used in the literature [6] to describe and to reason about distributed algorithms. Chord$^{\#}$ and the transaction algorithms described above were 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 can be easily mapped to Erlang processes and messages. The close relationship between the theoretical model 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 many components. It has 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.

4.1 Components and Supervisor Tree

Scalaris is a distributed algorithm. Each peer runs a number of processes as shown in Fig. 4:

Failure Detector supervises other peers and sends a crash message when a node failure is detected.

Configuration provides access to the configuration file and maintains parameter changes made at runtime.

Key Holder stores the identifier of the node in the overlay.

Statistics Collector collects statistics and forwards them to central statistic servers.

Chord$^{\#}$ Node performs all important functions of the node. It maintains, among other things, the successor list and the routing table.

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 processes are organized in a supervisor tree as illustrated in Fig. 4. The first four processes are supervised by a *one-for-one supervisor* [1]: When a slave crashes, it is restarted by the supervisor. The right-most processes (Chord$^{\#}$ Node and Database) are supervised by an *all-for-one supervisor* which restarts *all* slaves when a single slave crashed. In Scalaris, when either of the Chord$^{\#}$ Node or the Database process fails, the other is explicitly killed and both are restarted to ensure consistency.

4.2 Naming Processes

In Erlang, there are two ways of sending messages to processes: by process id or by addressing the name registered as an atom. This scheme provides a flat name space. We implemented a hierarchical name space for processes.

As described in Sec. 4.1, each Chord$^{\#}$ node comprises a group of processes. Within this group, we address processes by name. For example, the failure detector can be addressed as failure_detector.

Running several Chord$^{\#}$ nodes within one Erlang Virtual Machine (VM) would lead to name clashes. Hence, we implemented a hierarchical process name space where each Chord$^{\#}$ node forms a 'process group'. As a side-effect, we can traverse the naming hierarchy to provide monitoring information grouped by Chord$^{\#}$ nodes.

For this naming scheme, every process stores its group id in its own process dictionary. At startup time, processes announce their name and process identifier to a dictionary inside the VM, which is handled by a separate process in the VM. It can be queried to find processes by name or by traversing the process hierarchy. Additionally, most Chord$^{\#}$ processes support the {'$gen_cast', {debug_info, Requestor}} message, which allows processes to provide custom monitoring information to the web interface.

4.3 WAN Deployment

Erlang provides the 'distributed mode' for small and medium deployments with limited security requirements. This makes it easy to port the application from an Erlang VM to a cluster. In large deployments, however, the network traffic caused by the management tasks within the VM dominates the overall traffic.

In our code, we replaced the '!' operator and the self() function by cs_send:send() resp. cs_send:this(). At compile time we can configure the cs_send module to use the Erlang distributed mode or our own transport layer using TCP/IP, which will be based on the Erlang SSL library in the future.

This approach also allows us to separate the application logic from the transport layer. Hence, NAT traversal schemes and firewall-aware communication can be implemented without the need to change Chord$^{\#}$ code.

4.4 Transaction Interface

Transactions are executed in two phases, the read phase and the commit phase. The read phase goes through all operations of the transaction and keeps the result of each operation in the transaction log. During this phase, the state of the system remains unchanged. In the commit phase, the recorded effects are applied to the database when the ACID properties are not violated.

Read phase. For the read phase, we use a lambda expression which describes the individual operations to be performed in the transaction (see Alg. 4.1). The mentioned transaction log is passed through all calls to the transaction API and updated accordingly. Passing a function to the transaction framework allows us to easily re-execute a transaction after a failure due to concurrency.

Commit phase. The commit phase is started by calling do_transaction (see last line in Alg. 4.1). The transaction is executed asynchronously. The function spawns a new process and returns immediately. The ProcessId which is passed will be notified of the outcome of the transaction. The SuccessFun resp. FailureFun are applied to the result of the transaction before the result is sent back. For the Scalaris implementation, we use the two functions to include transaction numbers into the status messages when a process has several outstanding transactions.

We use the Jinterface package to enable Java programs to perform transactions. The transaction log is managed by the Java program. On a commit the complete log is passed to Erlang and the do_transaction function. Note that transaction descriptions in Java are usually more compact because error handling is done using exceptions (see Alg. 4.2) while in Erlang, the error handling is done in the actual code.

Algorithm 4.1 Incrementing the key *Increment* inside a transaction

```
run_test_increment(State, Source_PID)->
    % the transaction
    TFun = fun(TransLog) ->
        Key = "Increment",
        {Result, TransLog1} = transaction_api:read(Key, TransLog),
        {Result2, TransLog2} =
            if Result == fail ->
                    Value = 1,                  % new key
                    transaction_api:write(Key, Value, TransLog);
                true ->
                    {value, Val} = Result,      % existing key
                    Value = Val + 1,
                    transaction_api:write(Key, Value, TransLog1)
            end,
        % error handling
        if Result2 == ok ->
            {{ok, Value}, TransLog2};
            true -> {{fail, abort}, TransLog2}
        end
    end,
    SuccessFun = fun(X) -> {success, X} end,
    FailureFun =
      fun(Reason)-> {failure, "test increment failed", Reason} end,

    % trigger transaction
    transaction:do_transaction(State, TFun, SuccessFun,
                               FailureFun, Source_PID).
```

Algorithm 4.2 Java Transactions

```
// new Transaction object
Transaction transaction = new Transaction();
// start new transaction
transaction.start();

//read account A
int accountA =
    new Integer(transaction.read("accountA")).intValue();
//read account B
int accountB =
    new Integer(transaction.read("accountB")).intValue();

//remove 100$ from accountA
transaction.write("accountA",
    new Integer(accountA - 100).toString());
//add 100$ to account B
transaction.write("accountB",
    new Integer(accountB + 100).toString());

transaction.commit();
```

5. Responsive, Stateful Server in Erlang

In distributed server software, slow write operations often block faster reads. Alg. 5.1 shows a generic server architecture (design pattern) that manages reads and writes on a shared state separately. This is done in such a way that read requests can be immediately answered even though a concurrent write operation still blocks the process. Two processes manage the shared state: a public asyn-

Algorithm 5.1 Responsive, stateful server

```
-module(account).
-export([start/0,syncloop/2,slowbalance/2]).

newAccount() -> 0.
start() -> spawn(fun() ->
        Account = newAccount(),
        SyncLoopPid = spawn(account, syncloop, [self(), Account]),
        asyncloop(SyncLoopPid, Account)
       end).

% all requests have to be send to the asyncloop
% read from State via spawns, if its a slow read
% forward writes to the syncloop
asyncloop(SyncLoopPid, State) ->
    receive
    {updatestate, StateNew} ->
        % for better consistency make a join for all spawned
        %   slow reads here
        % for better security, only allow the syncloop
        %   process to update the state
        asyncloop(SyncLoopPid, StateNew);
    {balance, Pid} ->
        Pid ! State,
        asyncloop(SyncLoopPid, State);
    {slowbalance, Pid} ->
        spawn(account, slowbalance, [State, Pid]),
        asyncloop(SyncLoopPid, State);
    % all other messages go to the synchronous loop
    Message ->
            SyncLoopPid ! Message,
            asyncloop(SyncLoopPid, State)
    end.

% internally use a syncloop to serialize all State changes
syncloop(AsyncLoopPid, State) ->
    receive
    {credit, Amount} ->
        NewState = State + Amount,
        AsyncLoopPid ! {updatestate, NewState},
        syncloop(AsyncLoopPid, NewState);
    {draw, Amount} ->
          NewState = State - draw(Amount),
            AsyncLoopPid ! {updatestate, NewState},
          syncloop(AsyncLoopPid, NewState);
        _ ->
            syncloop(AsyncLoopPid, State)
    end.

% functions, that take some time to be executed
slowbalance(State, Pid) ->
    receive
    after 60000 ->
      Pid ! State
    end.

draw(Amount) ->
    receive
    % the bank still works with your money for 10 seconds
    after 10000 ->
      Amount
    end.
```

chronous receive loop asyncloop that performs the reads and forwards the write requests to a private synchronous receive loop syncloop. By this means, write requests are serialized and there is a local atomic point in time when the state changes.

Slow reads may still deliver outdated state. This can be overcome by waiting for all outstanding reads to be completed before changing the state in the asyncloop (not depicted in the algorithm).

Example. Alg. 5.1 shows the processing of states for a bank account. The server provides two read requests (balance and slowbalance) and two write requests (credit and draw) for managing an account. Clients send all their requests to the asyncloop. The server is started by calling account:start(). This spawns a process, which first initializes the account with zero, spawns the syncloop with a reference to itself, and finally executes the asyncloop.

On a balance or slowbalance request to the asyncloop, the account balance is returned to the requesting process from the current state. In case of slowbalance the state is given to a spawned process, which is then executed concurrently in the background. In practice, this spawning should be used when some calculations or other time consuming tasks must be executed on the state before the request can be answered. This way, other requests can be performed by the server concurrently. Here, the corresponding function slowbalance just waits 60 seconds before delivering the result.

In addition, the asyncloop handles updatestate requests as discussed below. All other messages are forwarded to the syncloop.

The syncloop handles the write requests credit and draw. All other messages are ignored and dropped. The syncloop must not spawn processes to calculate state changes, as all state manipulation must be serial to ensure consistency. Here, the draw takes 10 seconds to be performed (the bank uses this time to work with your money). This time has to be consumed synchronously. In practice this could be a time consuming calculation which is necessary to determine the new state. After having calculated the new state, syncloop sends the state with an updatestate request to the asyncloop and works on the new state by itself.

When the asyncloop receives an updatestate message from the syncloop it takes over the new state from the message. This is the atomic point in time when the write request becomes active, as all future requests will operate on this new state.

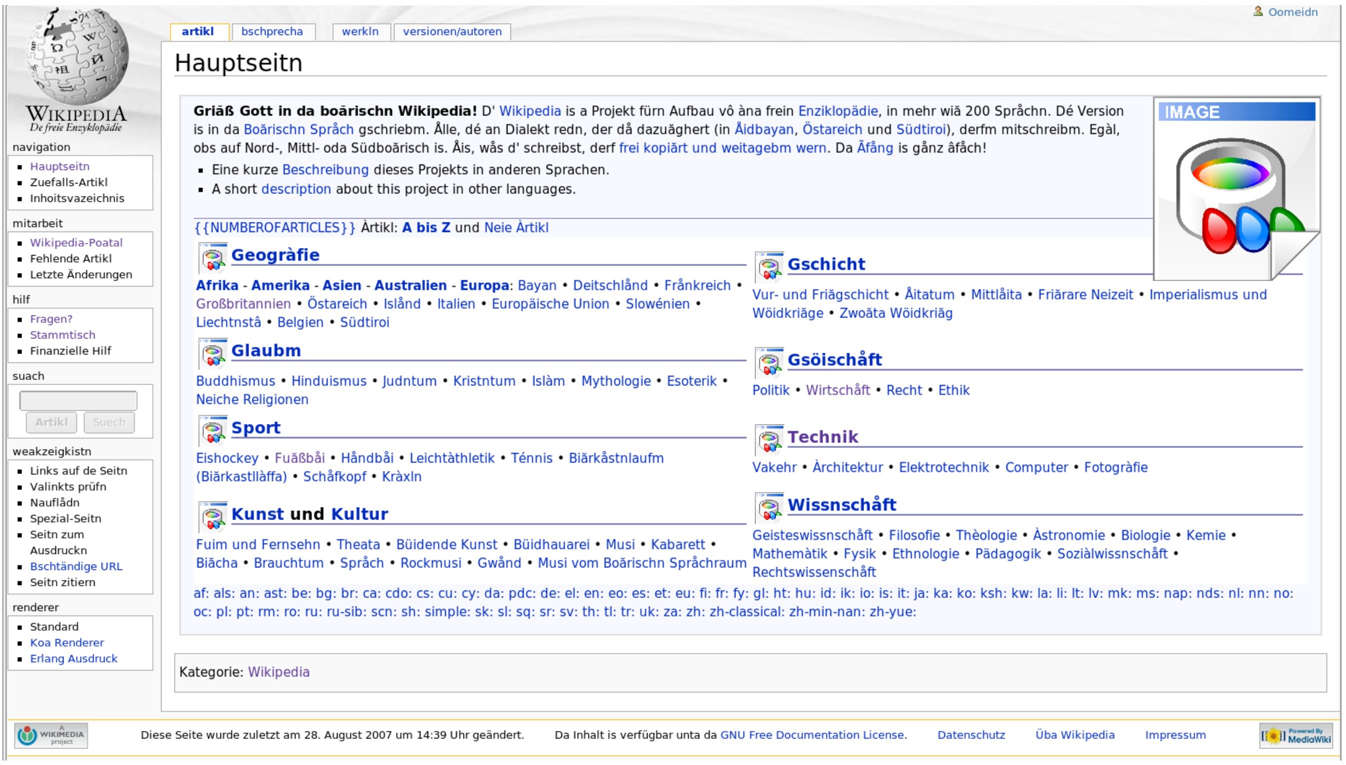
This leads to a relaxed consistency in the server that is sufficient for updating the routing tables and successor lists. Here, relaxed consistency does not harm, because these tables are subject to churn and will be periodically updated with unreliable link information anyway. If a stronger consistency model is needed, the transaction mechanism of the Erlang Mnesia database package could be used.

6. Use Case: Wikipedia

To demonstrate Scalaris' performance, we chose Wikipedia, the 'free encyclopedia, that anyone can edit', as a challenging test application. In contrast to the public Wikipedia, which is operated on three clusters in Tampa, Amsterdam, and Seoul, our Erlang implementation can be deployed on worldwide distributed servers. We ran it in two installations, one on PlanetLab and one on a local cluster.

The public Wikipedia uses PHP to render the Wikitext to HTML and stores the content and page history in MySQL databases. Instead of using a relational database, we map the Wikipedia content to our Scalaris key/value store [12]. 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



Oomeidn

artikl | bschprecha | werkln | versionen/autoren

Hauptseitn

Griäß Gott in da boärischn Wikipedia! D' Wikipedia is a Projekt fürn Aufbau vô àna frein Enziklopädie, in mehr wiă 200 Språchn. Dé Version is in da Boărischn Språch gschriebm. Ålle, dé an Dialekt redn, der då dazuăghert (in Åidbayan, Östareich und Südtiroi), derfm mitschreibm. Egàl, obs auf Nord-, Mittl- oda Südboărisch is. Åis, wås d' schreibst, derf frei kopiärt und weitagebm wern. Da Åfång is gånz âfåch!

- Eine kurze Beschreibung dieses Projekts in anderen Sprachen.
- A short description about this project in other languages.

IMAGE

{{NUMBEROFARTICLES}} Àrtikl: **A bis Z** und Neie Àrtikl

Geogràfie

Afrika - **Amerika** - **Asien** - **Australien** - **Europa**: Bayan • Deitschlånd • Frånkreich • Großbritannien • Östareich • Islånd • Italien • Europäische Union • Slowénien • Liechtnstâ • Belgien • Südtiroi

Gschicht

Vur- und Friägschicht • Åitatum • Mittlåita • Friărare Neizeit • Imperialismus und Wöidkriäge • Zwoăta Wöidkriäg

Glaubm

Buddhismus • Hinduismus • Judntum • Kristntum • Islàm • Mythologie • Esoterik • Neiche Religionen

Gsöischåft

Politik • Wirtschåft • Recht • Ethik

Sport

Eishockey • Fuäßbåi • Håndbåi • Leichtàthletik • Ténnis • Biărkåstnlaufm (Biărkastllåffa) • Schåfkopf • Kràxln

Technik

Vakehr • Àrchitektur • Elektrotechnik • Computer • Fotogràfie

Kunst und Kultur

Fuim und Fernsehn • Theata • Büidende Kunst • Büidhauarei • Musi • Kabarett • Biăcha • Brauchtum • Språch • Rockmusi • Gwånd • Musi vom Boărischn Språchraum

Wissnschåft

Geisteswissnschåft • Filosofie • Thèologie • Àstronomie • Biologie • Kemie • Mathemàtik • Fysik • Ethnologie • Pädagogik • Soziàlwissnschåft • Rechtswissenschåft

af: als: an: ast: be: bg: br: ca: cdo: cs: cu: cy: da: pdc: de: el: en: eo: es: et: eu: fi: fr: fy: gl: ht: hu: id: ik: io: is: it: ja: ka: ko: ksh: kw: la: li: lt: lv: mk: ms: nap: nds: nl: nn: no: oc: pl: pt: rm: ro: ru: ru-sib: scn: sh: simple: sk: sl: sq: sr: sv: th: tl: tr: uk: za: zh: zh-classical: zh-min-nan: zh-yue:

Kategorie: Wikipedia

WIKIPEDIA
De freie Enzyklopädie

navigation
- Hauptseitn
- Zuefalls-Artikl
- Inhoitsvazeichnis

mitarbeit
- Wikipedia-Poatal
- Fehlende Artikl
- Letzte Änderungen

hilf
- Fragen?
- Stammtisch
- Finanzielle Hilf

suach

Artikl | Suech

weakzeigkistn
- Links auf de Seitn
- Valinkts prüfn
- Nauflådn
- Spezial-Seitn
- Seitn zum Ausdruckn
- Bschtändige URL
- Seitn zitiern

renderer
- Standard
- Koa Renderer
- Erlang Ausdruck

Diese Seite wurde zuletzt am 28. August 2007 um 14:39 Uhr geändert. Da Inhalt is verfügbar unta da GNU Free Documentation License. Datenschutz Üba Wikipedia Impressum

Figure 6. Screenshot of the Bavarian Wikipedia on Scalaris. Images are not included in the dump.

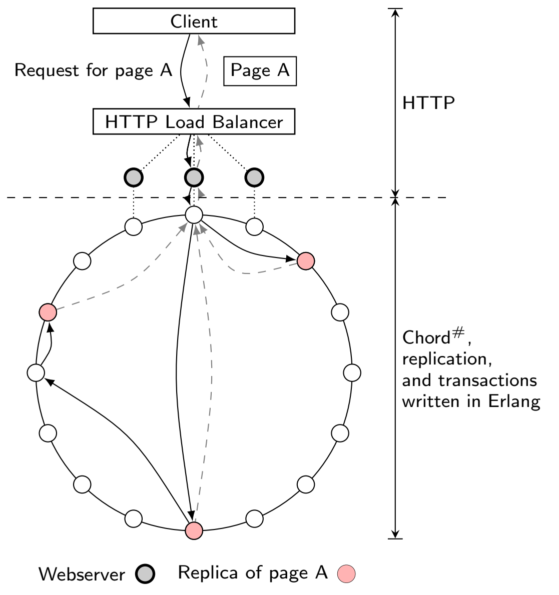


Figure 5. Wikipedia on Scalaris.

The page rendering of the Wikitext is done in Java in the web servers (see Fig. 5) running jetty. Here, we modified the Wikitext renderer of the *plog4u* project for our purposes.

Using this data layout, users may view pages by typing the URL, they can navigate to other pages via hyperlinks, they can edit pages and view the history of changes, and create new pages (see the screenshot in Fig. 7). Since the Wikipedia dumps do not include images, we render a proxy image at the corresponding positions instead. Moreover, we do not maintain a full text index and therefore full text search is not supported by our implementation. This could easily be performed by external crawling and search indexing mechanisms.

When modifying a page, a transaction over all replicas of the responsible keys is created and executed. The transaction includes the page itself, all backlink pages for inserted and deleted links, and all category pages for inserted and deleted categories.

Performance. Our Erlang implementation serves 2,500 transactions per second with just 16 servers. This is better than the public Wikipedia, which serves a total of 45,000 requests per second, of which only 2,000 hit the backend of approx. 200 servers. For the experiments, we used a HTTP load balancer (haproxy) to distribute the requests over all participating servers. The load generator (siege) requested randomly selected pages from the load balancer.

7. Conclusion

We presented *Scalaris*, a distributed key/value store based on the Chord$^{\#}$ structured overlay with symmetric data replication and a transaction layer implementing ACID properties. With Wikipedia as a demonstrator application we showed that Scalaris provides the desired scalability and efficiency.

Our implementation greatly benefited from the use of Erlang/OTP. It provides a set of useful libraries and operating procedures for building reliable distributed applications. As a result, the code is more concise than C or Java code.

Additionally, we presented an Erlang pattern that implements responsive, stateful services by overlapping fast reads with concurrent synchronous (slower) write operations. This framework did not only prove useful in our key/value store, but it can be used in many other Erlang implementations.

We believe that Scalaris could be of great value for suppliers of online services such as Amazon, eBay, Myspace, YouTube, or Google. Today, global service providers face the challenge of ensuring consistent data access for millions of customers in a 24/7 mode. In such environments, system crashes, software faults and heavy load imbalances are the norm rather than exceptions. Here, it is a challenging task to maintain a consistent view on data and services while hiding failures from the application.

Our P2P approach with replication and ACID provides a dependable and scalable alternative to standard database technology, albeit with a reduced data model. Each additional peer contributes additional main memory to the system, hence the combined memory capacity resembles that of current (large) SAN storage systems. If this is not sufficient, Scalaris can be easily modified to write its data onto disk. For backup purposes, our ACID implementation allows to take consistent snapshots of all data items during runtime.

Apart from distributed transactional data management, Scalaris can also be used for building scalable, hierarchical pub/sub services, reliable resource selection in dynamic systems, or internet chat services.

Acknowledgments

Many thanks to Joe Armstrong for commenting on our responsive server code and to Nico Kruber for implementing the Java transaction interface and adapting the Wiki renderer. This work was partly funded by the EU project Selfman under grant IST-34084 and the EU project XtreemOS under grant IST-33576.

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. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.

[10] 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.

[11] M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

[12] 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.

[13] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.

[14] T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.

[16] 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.

[17] 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.