Scalaris

Users and Developers Guide

Version 0.1

Florian Schintke, Thorsten Schütt

November 16, 2009

Contents

I Users Guide **5**

1 Introduction **7**
- 1.1 Brewer's CAP Theorem . . . 7

2 Download and Installation **9**
- 2.1 Requirements . . . 9
- 2.2 Download . . . 9
  - 2.2.1 Development Branch . . . 9
  - 2.2.2 Releases . . . 9
- 2.3 Configuration . . . 9
- 2.4 Build . . . 10
  - 2.4.1 Linux . . . 10
  - 2.4.2 Windows . . . 10
  - 2.4.3 Java-API . . . 10
- 2.5 Running Scalaris . . . 11
  - 2.5.1 Running on a local machine . . . 11
  - 2.5.2 Running distributed . . . 11
- 2.6 Installation . . . 12
- 2.7 Logging . . . 12

3 Using the system **13**
- 3.1 JSON API . . . 13
  - 3.1.1 Deleting a key . . . 15
- 3.2 Java command line interface . . . 16
- 3.3 Java API . . . 16

4 Testing the system **17**
- 4.1 Running the unit tests . . . 17

5 Troubleshooting **19**
- 5.1 Network . . . 19

II Developers Guide **21**

6 How a node joins the system **23**
- 6.1 General Erlang server loop . . . 23
- 6.2 Starting additional local nodes after boot . . . 23
  - 6.2.1 Supervisor-tree of a Scalaris node . . . 24
  - 6.2.2 Starting the or-supervisor and general processes of a node . . . 24
  - 6.2.3 Starting the and-supervisor with a peer and its local database . . . 26
  - 6.2.4 Initializing a `cs_node`-process . . . 27

6.2.5 Actually joining the ring 28
6.2.6 Beginning to serve requests 30

7 Routing and routing tables in the Overlay **31**
7.1 Simple routing table 33
7.1.1 Data types 33
7.1.2 A simple routingtable behaviour 33
7.2 Chord routing table 34
7.2.1 Data types 34
7.2.2 The routingtable behaviour for Chord 35

8 Directory Structure of the Source Code **39**

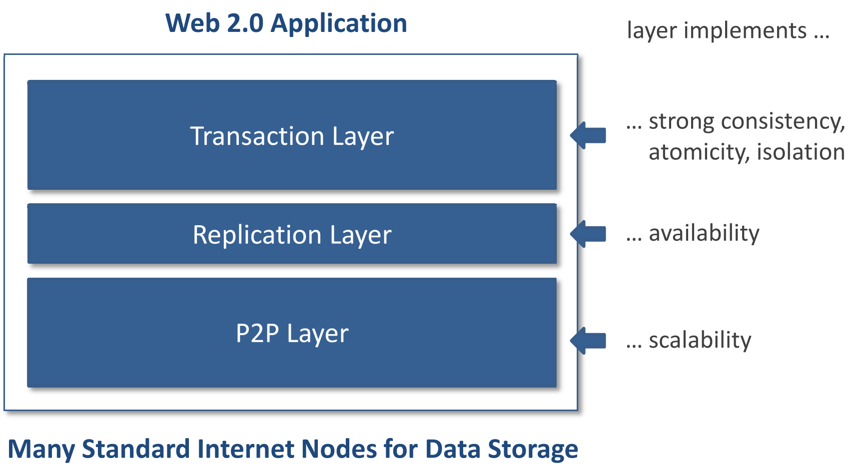
9 Java API **41**

Part I

Users Guide

1 Introduction

Scalaris is a scalable, transactional, distributed key-value store based on the peer-to-peer principle. It can be used to build scalable Web 2.0 services. The concept of Scalaris is quite simple: Its architecture consists of three layers.
It provides self-management and scalability by replicating services and data among peers. Without system interruption it scales from a few PCs to thousands of servers. Servers can be added or removed on the fly without any service downtime.



Scalaris takes care of:

- Fail-over
- Data distribution
- Replication
- Strong consistency
- Transactions

The Scalaris project was initiated by Zuse Institute Berlin and onScale solutions and is partly funded by the EU projects Selfman and XtreemOS. Additional information (papers, videos) can be found at http://www.zib.de/CSR/Projects/scalaris and http://www.onscale.de/scalaris.html.

1.1 Brewer's CAP Theorem

In distributed computing there exist the so called CAP theorem. It basically says that in distributed systems there are three desirable properties for such systems but can have only any two of them.

Strict Consistency. Any read operation has to return the result of the latest write operation on the same data item.

Availability. Items can be read and modified at any time.

Partition Tolerance. The network on which the service is running may split into several partitions which cannot communicate with each other. Lateron the may rejoin again.

For example, a service is hosted on one machine in Seattle and one machine in Berlin. This service is partition tolerant if it can tolerate that all Internet connections over the Atlantic (and Pacific) are interrupted for a few hours and then get repaired afterwards.

The goal of Scalarisis to provide strict consistency and partition tolerance. We are willing to sacrifice availability to make sure that the stored data is always consistent. I.e. when you are running Scalariswith a replication degree of 4 and the network splits into two partitions, one partition with three replicas and one partition with one replica, you will be able to continue to use the service only in the larger partition. All requests in the smaller partition will time out until the two networks merge again. Note, most other key-value stores tend to sacrifice consistency.

2 Download and Installation

2.1 Requirements

For building and running Scalaris, some third-party modules are required which are not included in the Scalaris sources:

- Erlang R12 or newer
- GNU-like Make

Note, the Version 12 of Erlang is required. Scalaris will not work with older versions.
To build the Java API (and the command-line client) the following modules are required additionally:

- Java Development Kit 1.6
- Apache Ant

Before building the Java API, make sure that **`JAVA_HOME`** and **`ANT_HOME`** are set. **`JAVA_HOME`** has to point to a JDK 1.6 installation, and **`ANT_HOME`** has to point to an Ant installation.

2.2 Download

The sources can be obtained from http://code.google.com/p/scalaris. RPMs are available from http://download.opensuse.org/repositories/home:/tschuett/.

2.2.1 Development Branch

You find the latest development version in the svn repository:

```
# Non-members may check out a read-only working copy anonymously over HTTP.
svn checkout http://scalaris.googlecode.com/svn/trunk/ scalaris-read-only
```

2.2.2 Releases

Releases can be found under the 'Download' tab on the web-page.

2.3 Configuration

Scalaris reads two configuration files from the working directory: **`bin/scalaris.cfg`** (mandatory) and **`bin/scalaris.local.cfg`** (optional). The former defines default settings and is included in the release. The latter can be created by the user to alter settings. A sample file is **`bin/scalaris.local.cfg.example`**. To run Scalarisdistributed over several nodes, each node requires a **`bin/scalaris.local.cfg`**:

File **scalaris.local.cfg**:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for distributed Erlang
% (see cs_send.erl to switch)

% {boot_host, {boot,'boot@foo.bar.com'}}.
% {log_host, {boot_logger, 'boot@foo.bar.com'}}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for TCP mode.
% (see cs_send.erl to switch)

% Insert the appropriate IP-addresses for your setup
% as comma separated integers:
% IP Address, Port, and label of the boot server
{boot_host, {{127,0,0,1},14195,boot}}.

% IP Address, Port, and label of the log server
{log_host, {{127,0,0,1},14195,boot_logger}}.
```

Scalarisdistinguishes currently two different kinds of nodes: (a) the boot-server and (b) regular nodes. For the moment, we limit the number of boot-servers to exactly one. The remaining nodes are regular nodes. The boot-server is contacted to join the system. On all servers, the **boot_host** option defines the server where the boot server is running. In the example, it is an IP address plus a TCP port.

2.4 Build

2.4.1 Linux

Scalaris uses autoconf for configuring the build environment and GNU Make for building the code.

```
%> ./configure
%> make
%> make docs
```

For more details read **README** in the main Scalaris checkout directory.

2.4.2 Windows

We are currently not supporting Scalaris on Windows. However, we have two small `bat` files for building and running a boot server. It seems to work but we make no guarantees.
For the most recent description please see the FAQ at http://code.google.com/p/scalaris/wiki/FAQ.

2.4.3 Java-API

The following commands will build the Java API for Scalaris:

```
%> make java
```

This will build scalaris.jar, which is the library for accessing the overlay network. Optionally, the documentation can be build:

```
%> cd java-api
%> ant doc
```

2.5 Running Scalaris

As mentioned above, in Scalaris there are two kinds of nodes:

- boot servers
- regular nodes

In every Scalaris, at least one boot server is required. It will maintain a list of nodes taken part in the system and allows other nodes to join the ring. For redundancy, it is also possible to have several boot servers. In the future, we want to eliminate this distinction, so any node is also a boot-server.

2.5.1 Running on a local machine

Open at least two shells. In the first, go into the bin directory:

```
%> cd bin
%> ./boot.sh
```

This will start the boot server. On success http://localhost:8000 should point to the management interface page of the boot server. The main page will show you the number of nodes currently in the system. After a couple of seconds a first Scalaris should have started in the boot server and the number should increase to one. The main page will also allow you to store and retrieve key-value pairs.
In a second shell, you can now start a second Scalaris node. This will be a 'regular server'. Go in the bin directory:

```
%> cd bin
%> ./cs_local.sh
```

The second node will read the configuration file and use this information to contact the boot server and will join the ring. The number of nodes on the web page should have increased to two by now. Optionally, a third and fourth node can be started on the same machine. In a third shell:

```
%> cd bin
%> ./cs_local2.sh
```

In a fourth shell:

```
%> cd bin
%> ./cs_local3.sh
```

This will add 3 nodes to the network. The web pages at http://localhost:8000 should show the additional nodes.

2.5.2 Running distributed

Scalaris can be installed on other machines in the same way as described in Sect. 2.6. In the default configuration, nodes will look for the boot server on localhost on port 14195. You should create a **scalaris.local.cfg** pointing to the node running the boot server.

```
% Insert the appropriate IP-addresses for your setup
% as comma separated integers:
% IP Address, Port, and label of the boot server
{boot_host, {{127,0,0,1},14195,boot}}.
```

If you are using the default configuration on the boot server it will listen on port 14195 and you only have to change the IP address in the configuration file. Otherwise the other nodes will not find the boot server. On the remote nodes, you only need to call **./cs_local.sh** and they will automatically contact the configured boot server.

2.6 Installation

For simple tests, you do not need to install Scalaris. You can run it directly from the source directory. Note: **make install** will install scalaris into **/usr/local**. But is more convenient to build RPMs and install those.

```
svn checkout http://scalaris.googlecode.com/svn/trunk/ scalaris-0.0.1
tar -cvjf scalaris-0.0.1.tar.bz2 scalaris-0.0.1 --exclude-vcs
cp scalaris-0.0.1.tar.bz2 /usr/src/packages/SOURCES/
rpmbuild -ba scalaris-0.0.1/contrib/scalaris.spec
```

Your source and binary rpm will be generated in /usr/src/packages/SRPMS and RPMS. We also build rpms using checkouts from svn and provide them using the openSUSE BuildService at http://download.opensuse.org/repositories/home:/tschuett/. RPM packages are available for

- Fedora 9, 10,
- Mandriva 2008, 2009,
- openSUSE 11.0, 11.1,
- SLE 10, 11,
- CentOS 5 and
- RHEL 5.

Inside those repositories you will also find an erlang rpm - you don't need this if you already have a recent enough erlang version!

2.7 Logging

Scalaris uses the log4erl library (see **contrib/log4erl** for logging status information and error messages. The log level can be configured in **bin/scalaris.cfg**. The default value is `error`; only errors and severe problems are logged.

```
%% @doc Loglevel: debug < info < warn < error < fatal < none
{log_level, error}.
```

In some cases, it might be necessary to get more complete logging information, e.g. for debugging. In 6.2 on page 23, we are explaining the startup process of Scalarisnodes in more detail, here the `info` level provides more detailed information.

```
%% @doc Loglevel: debug < info < warn < error < fatal < none
{log_level, info}.
```

3 Using the system

3.1 JSON API

Scalaris supports a JSON API for transactions. To minimize the necessary round trips between a client and Scalaris, it uses request lists, which contain all requests that can be done in parallel. The request list is then send over to a Scalaris node with a POST message. The result is an opaque TransLog and a list containing the results of the requests. To add further requests to the transaction, the TransLog and another list of requests may be send to Scalaris. This process may be repeated as necessary. To finish the transaction, the request list can contain a 'commit' request as last element, which triggers the validation phase of the transaction processing.
The JSON-API can be accessed via the Scalaris-Web-Server running on port 8000 by default and the page **`jsonrpc.yaws`** (For example at: http://localhost:8000/jsonrpc.yaws). The following example illustrates the message flow:

Client | **Scalaris node**

Make a transaction, that sets two keys: →

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      [
        { "write":{"keyA":"valueA"} },
        { "write":{"keyB":"valueB"} },
        { "commit":"commit" }
      ]
    ],
  "id":0
}
```

← Scalaris sends results back

```
{ "result":
  { "results":
      [
        { "op":"commit",
          "value":"ok",
          "key":"ok" },
        { "op":"write",
          "value":"valueB",
          "key":"keyB" },
        { "op":"write",
          "value":"valueA",
          "key":"keyA" }
      ],
    "translog":
      [...]
  },
  "id" : 0
}
```

In a second transaction: Read the two keys →

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      [
        { "read":"keyA" },
        { "read":"keyB" }
      ]
    ]
  "id":0
}
```

← Scalaris sends results back

```
{ "result":
  {"results":
    [
      { "op":"read",
        "value":"valueB",
        "key":"keyB" },
      { "op":"read",
        "value":"valueA",
        "key":"keyA" }
    ],
   "translog":
    [...] // this list is the translog
          // for further operations!
          // We name it TLOG here.
  },
  "id" : 0
}
```

Calculate something with the read values and make further requests, here a write and the commit for the whole transaction. Include also the latest translog we got from Scalaris (named **TLOG** here). →

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      TLOG, // translog from prev. result.
      [
        { "write":{"keyA":"valueA2"} },
        { "commit":"commit" }
      ]
    ],
  "id" : 0
}
```

← Scalaris sends results back

```
{ "result":
  { "results":
    [ { "op":"commit",
        "value":"ok",
        "key":"ok" },
      { "op":"write",
        "value":"valueA2",
        "key":"keyA" }
    ],
    "translog":
    [...]
  },
  "id" : 0
}
```

A sample usage of the JSON API using Ruby can be found in **`contrib/jsonrpc.rb`**.
A single request list must not contain a key more than once!
The allowed requests are:

```
{ "read":"any_key" }

{ "write":{"any_key":"any_value"} }

{ "commit":"commit" }
```

The possible results are:

```
{ "op":"read", "key":"any_key", "value":"any_value" }
{ "op":"read", "key":"any_value", "fail":"reason" } // 'not_found' or 'timeout'

{ "op":"write", "key":"any_key", "value":"any_value" }
{ "op":"read", "key":"any_key", "fail":"reason" }

{ "op":"commit", "value":"ok", "key":"ok" }
{ "op":"commit", "value":"fail", "fail":"reason" }
```

3.1.1 Deleting a key

Outside transactions keys can also be deleted, but it has to be done with care, as explained in the following thread on the mailing list: http://groups.google.com/group/scalaris/browse_thread/thread/ff1d9237e218799.

```
{
  "method":"delete",
  "version":"1.1",
  "params":
    [
      { "key":"any_key" }
    ],
  "id" : 0
}
```

Two sample results

```
{ "result":
  { "ok":2, // how many replicas were deleted successsfully
    "results": [ "ok", "ok", "locks_set", "undef" ]
  }
}
```

```
{ "result":
  { "failure":"reason" }
}
```

3.2 Java command line interface

The jar file contains a small command line interface client. For convenience, we provide a wrapper script called **scalaris** which setups the Java environment:

```
%> cd java-api
%> ./scalaris -help
usage: scalaris
 -g,--getsubscribers <topic>   get subscribers of a topic
 -help                         print this message
 -minibench                    run mini benchmark
 -p,--publish <params>         publish a new message for a topic: <topic>
                               <message>
 -r,--read <key>               read an item
 -s,--subscribe <params>       subscribe to a topic: <topic> <url>
 -u,--unsubscribe <params>     unsubscribe from a topic: <topic> <url>
 -w,--write <params>           write an item: <key> <value>
```

Read and write can be used to read resp. write from/to the overlay. getsubscribers, publish, and subscribe are the PubSub functions.

```
%> ./scalaris -write foo bar
write(foo, bar)
%> ./scalaris -read foo
read(foo) == bar
```

The scalaris library requires that you are running a 'regular server' on the same node. Having a boot server running on the same node is not sufficient.

3.3 Java API

The **scalaris.jar** provides the command line client as well as a library for Java programs to access Scalaris. The library provides two classes:

- **Scalaris** provides a high-level API similar to the command line client.
- **Transaction** provides a low-level API to the transaction mechanism.

For details we refer the reader to the Javadoc:

```
%> cd java-api
%> ant doc
%> firefox doc/index.html
```

4 Testing the system

4.1 Running the unit tests

There are some unit tests in the **`test`** directory. You can call them by running **`make test`** in the main directory. The results are stored in a local **`index.html`** file.
The tests are implemented with the **`common-test`** package from the Erlang system. For running the tests we rely on **`run_test`**, which is part of the **`common-test`** package, but is not installed by default. **`configure`** will check whether **`run_test`** is available. If it is not installed, it will show a warning and a short description of how to install the missing file.
Note: for the unit tests, we are setting up and shutting down several overlay networks. During the shut down phase, the runtime environment will print extensive error messages. These error messages do not indicate that tests failed! Running the complete test suite takes about 5 minutes. Only when the complete suite finished, it will present statistics on failed and successful tests.

5 Troubleshooting

5.1 Network

Scalarisuses a couple of TCP ports for communication. It does not use UDP at the moment.

8000	HTTP Server on the boot node
8001	HTTP Server on the other nodes
14195	Port for inter-node communication (boot server)
14196	Port for inter-node communication (other nodes)

Please make sure that at least 14195 and 14196 are not blocked by firewalls.

Part II

Developers Guide

6 How a node joins the system

6.1 General Erlang server loop

Servers in Erlang often use the following structure to maintain a state while processing received messages:

```
receive
  Message ->
    State1 = f(State),
    loop(State1)
end.
```

The server runs an endless loop, that waits for a message, processes it and calls itself using tail-recursion in each branch. The loop works on a **State**, which can be modified when a message is handled.

6.2 Starting additional local nodes after boot

After booting a new Scalaris-System as described in Section 2.5.1 on page 11, ten additional local nodes can be started by typing `admin:add_nodes(10)` in the Erlang-Shell that the boot process opened [1].

```
scalaris/bin> ./boot.sh
[...]
=INFO REPORT==== 12-May-2009::16:24:18 ===
Yaws: Listening to 0.0.0.0:8000 for servers
 - http://localhost:8000 under ../docroot
[info] [ CC ] this() == {{127,0,0,1},14195}
[info] [ DNC <0.96.0> ] starting DeadNodeCache
[info] [ DNC <0.96.0> ] starting Dead Node Cache
[info] [ RM <0.97.0> ] starting ring maintainer

[info] [ RT <0.99.0> ] starting routingtable
[info] [ Node <0.101.0> ] joining 315238232250031455306327244779560426902
[info] [ Node <0.101.0> ] join as first 315238232250031455306327244779560426902
[info] [ FD <0.74.0> ] starting pinger for {{127,0,0,1},14195,<0.101.0>}
[info] [ Node <0.101.0> ] joined
[info] [ CY ] Cyclon spawn: {{127,0,0,1},14195,<0.102.0>}
(boot@csr-pc9)1> admin:add_nodes(10)
```

In the following we will trace, what this function does to join additional nodes to the system. The function `admin:add_nodes(int)` is defined as follows.

File **admin.erl**:

```
%%--------------------------------------------------------------------
%% Function: add_nodes(int()) -> ok
%% Description: add new Scalaris nodes
%%--------------------------------------------------------------------
% @doc add new Scalaris nodes on the local node
% @spec add_nodes(int()) -> ok

add_nodes(Count) ->
```

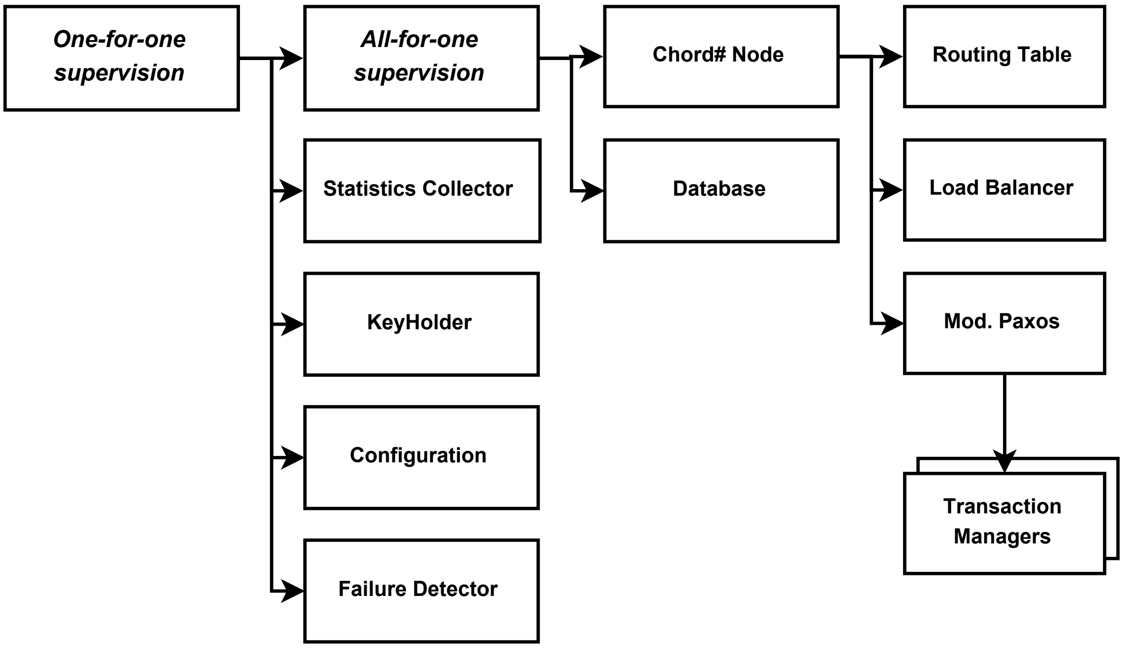
[1]Increase the log level to `info` to get the detailed startup logs. See Sect. 2.7 on page 12

```
    add_nodes(Count, 0).

% @spec add_nodes(int(), int()) -> ok
add_nodes(Count, Delay) ->
    add_nodes_loop(Count, Delay).

add_nodes_loop(0, _) ->
    ok;
add_nodes_loop(Count, Delay) ->
    supervisor:start_child(main_sup, {randoms:getRandomId(),
                                      {cs_sup_or, start_link, []},
                                      permanent,
                                      brutal_kill,
                                      worker,
                                      []}),
    %timer:sleep(Delay),
    add_nodes_loop(Count - 1, Delay).
```

It calls **add_nodes_loop(Count, Delay)** with a delay of 0. This function starts a new child for the main supervisor **main_sup**. As defined by the parameters, to actually perform the start, the function **cs_sup_or:start_link** is called by the Erlang supervisor mechanism. For more details on the OTP supervisor mechanism see Chapter 18 of the Erlang book [1] or the online documentation at http://www.erlang.org/doc/man/supervisor.html.

6.2.1 Supervisor-tree of a Scalaris node

When starting a new node in the system, the following supervisor tree is created:



6.2.2 Starting the or-supervisor and general processes of a node

Starting supervisors is a two step process: the supervisor mechanism first calls the **init()** function of the defined module (**cs_sup_or:init()** in this case) and then calls the start function (**start_link** here.
So, lets have a look at **cs_sup_or:init**, the 'Scalaris *or* supervisor'.

File **cs_sup_or.erl**:

```
init([Options]) ->
```

```
    InstanceId = string:concat("cs_node_", randoms:getRandomId()),
    boot_server:connect(),
    KeyHolder =
        {cs_keyholder,
         {cs_keyholder, start_link, [InstanceId]},
         permanent,
         brutal_kill,
         worker,
         []},
    _RSE =
        {rse_chord,
         {rse_chord, start_link, [InstanceId]},
         permanent,
         brutal_kill,
         worker,
         []},
    Supervisor_AND =
        {cs_supervisor_and,
         {cs_sup_and, start_link, [InstanceId, Options]},
         permanent,
         brutal_kill,
         supervisor,
         []},
    RingMaintenance =
        {?RM,

         {util, parameterized_start_link, [?RM:new(config:read(ringmaintenance_trigger)),
                                           [InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
    RoutingTable =
        {routingtable,
         {util, parameterized_start_link, [rt_loop:new(config:read(routingtable_trigger)),
                                           [InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
    DeadNodeCache =
        {deadnodecache,
          {util, parameterized_start_link, [dn_cache:new(config:read(dn_cache_trigger)),
                                           [InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
    Vivaldi =
        {vivaldi,
          {util, parameterized_start_link, [vivaldi:new(config:read(vivaldi_trigger)),
                                           [InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
     CS_Reregister =
        {cs_reregister,
         {util, parameterized_start_link,[cs_reregister:new(config:read(cs_reregister_trigger)),[InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
    DC_Clustering =
        {dc_clustering,
         {dc_clustering, start_link, [InstanceId]},
         permanent,
         brutal_kill,
         worker,
         []},
     Cyclon =
```

```
        {cyclon,
         {util, parameterized_start_link, [cyclon:new(config:read(cyclon_trigger)),
             [InstanceId]]},
         permanent,
         brutal_kill,
         worker,
         []},
    Self_Man =
        {self_man,
         {self_man, start_link, [InstanceId]},
         permanent,
         brutal_kill,
         worker,
         []},
    {ok, {{one_for_one, 10, 1},
          [
           Self_Man,
           CS_Reregister,
           KeyHolder,
           RoutingTable,
           Supervisor_AND,
           Cyclon,
           DeadNodeCache,
           RingMaintenance,
           Vivaldi

           %,DC_Clustering

           %_RSE
          ]}}.
```

The return value of the **init()** function specifies the child processes of the supervisor and how to start them. Here, we define a list of processes to be observed by a **one_for_one** supervisor. The processes are: **KeyHolder**, **DeadNodeCache**, **RingMaintenance**, **RoutingTable**, and a **Supervisor_AND** process.
The term **{one_for_one, 10, 1}** specifies that the supervisor should try 10 times to restart each process before giving up. **one_for_one** supervision means, that if a single process stops, only that process is restarted. The other processes run independently.
The **cs_sup_or:init()** is finished and the supervisor module, starts all the defined processes by calling the functions that were defined in the list of the **cs_sup_or:init()**.
For a join of a new node, we are only interested in the starting of the **Supervisor_AND** process here. At that point in time, all other defined processes are already started and running.

6.2.3 Starting the and-supervisor with a peer and its local database

Again, the OTP will first call the **init()** function of the corresponding module:

File **cs_sup_and.erl**:

```
init([InstanceId, Options]) ->
    Node =
        {cs_node,
         {cs_node, start_link, [InstanceId, Options]},
         permanent,
         brutal_kill,
         worker,
         []},
    DB =
        {?DB,
         {?DB, start_link, [InstanceId]},
         permanent,
         brutal_kill,
         worker,
         []},

```

```
    {ok, {{one_for_all, 10, 1},
          [
           DB,
           Node
           ]}}.
```

It defines three processes, that have to be observed using an **one_for_all**-supervisor, which means, that if one fails, all have to be restarted. Passed to the **init** function is the **InstanceId**, a random number to make nodes unique. It was calculated a bit earlier in the code. Exercise: Try to find where.
As you can see from the list, the **DB** is started before the **Node**. This is intended and important, because **cs_node** uses the database, but not vice versa. The supervisor first completely initializes the DB process and afterwards calls **cs_node:start_link**. We only go into details here, for the latter.

File **cs_node.erl**:

```
%% @doc spawns a scalaris node, called by the scalaris supervisor process
%% @spec start_link(term()) -> {ok, pid()}
start_link(InstanceId) ->
    start_link(InstanceId, []).

start_link(InstanceId, Options) ->
    gen_component:start_link(?MODULE, [InstanceId, Options], [{register, InstanceId, cs_node}]).
```

cs_node implements the **gen_component** behaviour. This component was developed by us to enable us to write code which is similar in syntax and semantics to the examples in [2]. Similar to the **supervisor** behaviour, the component has to provide an **init** function, but here it is used to initialize the state of the component. This function is described in the next section.
Note: ?**MODULE** is a predefined Erlang macro, which expands to the module name, the code belongs to (here: **cs_node**).

6.2.4 Initializing a `cs_node`-process

File **cs_node.erl**:

```
%% @doc joins this node in the ring and calls the main loop
-spec(init/1 :: ([any()]) -> cs_state:state()).
init([_InstanceId, _Options]) ->
    boot_server:be_the_first(),
    {join_state1}.
```

The **gen_component** behaviour registers the **cs_node** in the process dictionary. Formerly, the process had to do this himself, but we moved this code into the behaviour. If the **cs_node** is the first node, he will start immediately. Otherwise, the process sleeps for a random amount of time. If you would start 1000 processes with **admin:add_nodes**(1000), the boot-server would receive many join requests at the same time, which is not intended. It will also make the ring stabilization process more complicated. Adding 100s of nodes within a short period of time induces more churn into the system, than the ring maintenance can handle.
Then, the node retrieves its **Id** from the keyholder: **Id** = **cs_keyholder:get_key**(). In the first call, a random identifier is returned, otherwise the latest set value. If the **cs_node**-process failed and is restarted by its supervisor, this call to the keyholder ensures, that the node still keeps its **Id**, assuming that the keyholder process is not failing. This is important for the load-balancing and for consistent responsibility of nodes to ensure consistent lookup in the structured overlay. Note: the name **Key-holder** actually is an id-holder.
If a node changes its position in the ring for load-balancing, the key-holder will be informed and the **cs_node** finishes itself. This triggers a restart of the corresponding database process via the

and-supervisor. When the supervisor restarts both processes, they will retrieve the new position in the ring from the key-holder and join the ring there.
The supervisor was configured to restart a node at most 10 times. Does that mean, that a node can only change its position in the ring 10 times (caused by load-balancing)?

6.2.5 Actually joining the ring

After retrieving its identifier, the node starts the join process (**cs_join:join**).

File **cs_join.erl**:

The boot-server is contacted to retrieve the known number of nodes in the ring. If the ring is empty, **join_first** is called. Otherwise, **join_ring** is called.
If the ring is empty, the joining node is the only node in the ring and will be responsible for the whole key space. **join_first** just creates a new state for a Scalaris node consisting of an empty routing table, a successorlist containing itself, itself as its predecessor, a reference to itself, its responsibility area from **Id** to **Id** (the full ring), and a load balancing schema.

File **cs_join.erl**:

The macro ?**RT** maps to the configured routing algorithm and ?**RM** to the configured ring maintenance algorithm. It is defined in **include/scalaris.hrl**. For further details on the routing see Chapter 7 on page 31.
The state is defined in

File **cs_state.erl**:

```
new(RT, Successor, Predecessor, Me, MyRange, LB, DB) ->
    #state{
     routingtable = RT,
     successor = Successor,
     predecessor = Predecessor,
     me = Me,
     my_range = MyRange,
     lb=LB,
     join_time=now(),
     deadnodes = gb_sets:new(),
     trans_log = #translog{
       tid_tm_mapping = dict:new(),
       decided = gb_trees:empty(),
       undecided = gb_trees:empty()
      },
     db = DB
    }.
```

If a node joins an existing ring, **reliable_get_node** is called for the own **Id** in **cs_join:join()**. This lookup delivers the node who is currently responsible for the new node's identifier – the successor for the joining node. If this lookup fails for some reason, it is tried again, by recursivly calling the **join()**.
What, if the **Id** is exactly the same as that of the existing node? This could lead to lookup and responsibility inconsistency? Can this be triggered by the load-balancing? This is a bug, that should be fixed!!!
Then, **cs_join:join_ring** is called:

File **cs_join.erl**:

First the node is initialized. Then it sends a **join** message to the successor including a reference to itself and the chosen **Id**.
The message is received by the old node in **cs_node.erl**. There exists a **{join, X}** handler.

File **cs_node.erl**:

```
on({join, Source_PID, Id, UniqueId}, State) ->
    cs_join:join_request(State, Source_PID, Id, UniqueId);



on(_, _State) ->
    unknown_event.

%% userdevguide-begin cs_node:start
%% @doc joins this node in the ring and calls the main loop
-spec(init/1 :: ([any()]) -> cs_state:state()).
init([_InstanceId, _Options]) ->
    boot_server:be_the_first(),
    {join_state1}.

%% userdevguide-end cs_node:start

%% userdevguide-begin cs_node:start_link
%% @doc spawns a scalaris node, called by the scalaris supervisor process
%% @spec start_link(term()) -> {ok, pid()}
start_link(InstanceId) ->
    start_link(InstanceId, []).

start_link(InstanceId, Options) ->
    gen_component:start_link(?MODULE, [InstanceId, Options], [{register, InstanceId, cs_node}]).
%% userdevguide-end cs_node:start_link

get_local_cyclon_pid() ->
    InstanceId = erlang:get(instance_id),
    if
        InstanceId == undefined ->
           log:log(error,"[ Node ] ~p", [util:get_stacktrace()]);
        true ->
            ok
    end,
    process_dictionary:lookup_process(InstanceId, cyclon).

get_local_cs_reregister_pid() ->
    InstanceId = erlang:get(instance_id),
    if
        InstanceId == undefined ->
           log:log(error,"[ Node ] ~p", [util:get_stacktrace()]);
        true ->
            ok
    end,
    process_dictionary:lookup_process(InstanceId, cs_reregister).
```

This triggers a call to **join_request** on the old node.

File **cs_join.erl**:

```
join_request(State, Source_PID, Id, UniqueId) ->
    Pred = node:new(Source_PID, Id, UniqueId),
    {DB, HisData} = ?DB:split_data(cs_state:get_db(State), cs_state:id(State), Id),
    cs_send:send(Source_PID, {join_response, cs_state:pred(State), HisData}),
    ring_maintenance:update_pred(Pred),
    cs_state:set_db(State, DB).
```

The **cs_node** notifies the ring maintenance, that he has a new predecessor. Then he removes the key-value pairs from his database which are now in the responsibility of the joining node. Then it sends a **join_response** to the new node with its former predecessor, the data, it has to host, and its successorlist.

Back on the joining node: it waits for the **join_response** message in **cs_join:join_ring**(). The next steps after the message was received from the old node are to initialize the maintenance components for the ring and routing table, the database and the state of the **cs_node**.

6.2.6 Beginning to serve requests

cs_join:join() was called from **cs_node:start**(), which now continues

File **cs_node.erl**:

```
%% @doc joins this node in the ring and calls the main loop
-spec(init/1 :: ([any()]) -> cs_state:state()).
init([_InstanceId, _Options]) ->
    boot_server:be_the_first(),
    {join_state1}.
```

The **cs_replica_stabilization:recreate_replicas**() function is called, which is not yet implemented. It would recreated necessary replicas that were lost due to load-balancing and node failures.

Finally, the loop for request handling is started.

7 Routing and routing tables in the Overlay

Each node of the ring can perform searches in the overlay.
A search is done by a lookup in the overlay, but there are several other demands for communication between peers, so Scalaris provides a general interface to route a message to another peer, that is currently responsible for a given **key**.

File **cs_lookup.erl**:

```
[...]
unreliable_lookup(Key, Msg) ->
    get_pid(cs_node) ! {lookup_aux, Key, Msg}.

unreliable_get_key(Key) ->
    unreliable_lookup(Key, {get_key, cs_send:this(), Key}).
[...]
```

The message **Msg** could be a **get** which retrieves content from the responsible node or a **get_node** message, which returns a pointer to the node.
All currently supported messages are listed in the file **cs_node.erl**.
The message routing is implemented in lookup.erl

File **lookup.erl**:

```
[...]
lookup_fin(Msg) ->
    self() ! Msg.

lookup_aux(State, Key, Msg) ->
    Terminate = util:is_between(cs_state:id(State), Key, cs_state:succ_id(State)),
    P = ?RT:next_hop(State, Key),
    ?LOG("[ ~w | I | Node   | ~w ] lookup_aux ~w ~w ~s~n",
      [calendar:universal_time(), self(), Terminate, P, Key]),
    if
        Terminate ->
            cs_send:send(P, {lookup_fin, Msg});
        true ->
            cs_send:send(P, {lookup_aux, Key, Msg})
    end.
[...]
```

Each node is responsible for a certain key interval. The function **util:is_between** is used to decide, whether the key is between the current node and its successor. If that is the case, final step is done using **lookup_fin()**, which delivers the message to the local node. Otherwise, the message is forwarded to the next nearest known peer (listed in the routing table) determined by **?RT:next_hop**.
routingtable.erl is a generic interface for routing tables. It can be compared to interfaces in Java. In Erlang interfaces can be defined using a so called 'behaviour'. The files **rt_simple** and **rt_chord** implement the behaviour 'routingtable'.
The macro ?RT is used to select the current implementation of routing tables. It is defined in **scalaris.hrl**.

File **scalaris.hrl**:

```
%%This file determines which kind of routingtable is used. Uncomment the
%%one that is desired.
```

```

%%Standard Chord routingtable
-define(RT, rt_chord).

%%Simple routingtable
%-define(RT, rt_simple).
```

The functions, that have to be implemented for a routing mechanism are defined in the following file:

File **routingtable.erl**:

```
behaviour_info(callbacks) ->
    [
     % create a default routing table
     {empty, 1},
     % mapping: key space -> identifier space
     {hash_key, 1}, {getRandomNodeId, 0},
     % routing
     {next_hop, 2},
     % trigger for new stabilization round
     {init_stabilize, 3},
     % dead nodes filtering
     {filterDeadNode, 2},
     % statistics
     {to_pid_list, 1}, {get_size, 1},
     % for symmetric replication
     {get_keys_for_replicas, 1},
     % for debugging
     {dump, 1},
     % for bulkowner
     {to_dict, 1},
     % convert from internal representation to version for cs_node
     {export_rt_to_cs_node, 4},
     % update pred/succ in routing stored in cs_node
     {update_pred_succ_in_cs_node, 3}
    ];
```

empty/1 gets a successor passed and generates an empty routing table. The data structure of the routing table is undefined. It can be a list, a tree, a matrix ...

hash_key/1 gets a key and maps it into the overlay's identifier space.

getRandomNodeId/0 returns a random node id from the overlay's identifier space. This is used for example when a new node joins the system.

next_hop/2 gets a routing table and a key and returns the node, that should be contacted next (is nearest to the id).

init_stabilize/3 is called periodically to rebuild the routing table. The parameters are the identifier of the node, the successor and the old routing table state.

filterDeadNode/2 is called by the failuredetector and tells the routing table about dead nodes to be eliminated from the routing table. This function cleans the routing table.

to_pid_list/1 get all PIDs of the routing table entries.

get_size/1 get the routing table's size.

get_keys_for_replicas/1 Returns for a given **Key** the keys of its replicas. This used for implementing symmetric replication.

dump/1 dump the state. Not mandatory, may just return **ok**.

to_dict/1 returns the routing tables entries in an array-like structure. This is used by bulk-operations to create a broadcast tree.

7.1 Simple routing table

One implementation of a routing table is the **rt_simple**, which routes via the successor, which is inefficient, as it needs a linear number of hops to reach its goal. A more robust implementation, would use a successor list. This implementation is not very efficient on churn.

7.1.1 Data types

First, the data structure of the routing table is defined:

File **rt_simple.erl**:

```
% @type key(). Identifier.
-type(key()::pos_integer()).
% @type rt(). Routing Table.
-ifdef(types_are_builtin).
-type(rt()::{node:node_type(), gb_tree()}).
-type(external_rt()::{node:node_type(), gb_tree()}).
-else.
-type(rt()::{node:node_type(), gb_trees:gb_tree()}).
-type(external_rt()::{node:node_type(), gb_trees:gb_tree()}).
-endif.
```

A routing table is a pair of a node (the successor) and an (unused) **gb_tree**. Keys in the overlay are identified by integers.

7.1.2 A simple routingtable behaviour

File **rt_simple.erl**:

```
%% @doc creates an empty routing table.
%%      per default the empty routing should already include
%%      the successor
-spec(empty/1 :: (node:node_type()) -> rt()).
empty(Succ) ->
    {Succ, gb_trees:empty()}.
```

The empty routing table consists of the successor and an empty **gb_tree**.

File **rt_simple.erl**:

```
%% @doc hashes the key to the identifier space.
-spec(hash_key/1 :: (any()) -> key()).
hash_key(Key) ->
    BitString = binary_to_list(crypto:md5(Key)),
    % binary to integer
    lists:foldl(fun(El, Total) -> (Total bsl 8) bor El end, 0, BitString).
```

Keys are hashed using MD5 and have a length of 128 bits.

File **rt_simple.erl**:

```
%% @doc returns the next hop to contact for a lookup
-spec(next_hop/2 :: (cs_state:state(), key()) -> pid()).
next_hop(State, _Key) ->
    cs_state:succ_pid(State).
```

Next hop is always the successor.

File **rt_simple.erl**:

```
%% @doc triggered by a new stabilization round
-spec(init_stabilize/3 :: (key(), node:node_type(), rt()) -> rt()).
```

```
init_stabilize(_Id, Succ, _RT) ->
    % renew routing table
    empty(Succ).
```

init_stabilize/3 resets its routing table with the current successor.

File **rt_simple.erl**:

```
%% @doc removes dead nodes from the routing table
-spec(filterDeadNode/2 :: (rt(), cs_send:mypid()) -> rt()).
filterDeadNode(RT, _DeadPid) ->
    RT.
```

filterDeadNodes/2 does nothing, as only the successor is listed in the routing table and that is reset periodically in **init_stabilize**/3.

File **rt_simple.erl**:

```
%% @doc returns the pids of the routing table entries .
-spec(to_pid_list/1 :: (rt()) -> [cs_send:mypid()]).
to_pid_list({Succ, _RoutingTable} = _RT) ->
    [node:pidX(Succ)].
```

to_pid_list/1 returns the pids of the routing tables, as defined in **node.erl**.

File **rt_simple.erl**:

```
normalize(Key) ->
    Key band 16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.

n() ->
    16#100000000000000000000000000000000.

%% @doc returns the replicas of the given key
-spec(get_keys_for_replicas/1 :: (key() | string()) -> [key()]).
get_keys_for_replicas(Key) when is_integer(Key) ->
    [Key,
     normalize(Key + 16#40000000000000000000000000000000),
     normalize(Key + 16#80000000000000000000000000000000),
     normalize(Key + 16#C0000000000000000000000000000000)
    ];
get_keys_for_replicas(Key) when is_list(Key) ->
    get_keys_for_replicas(hash_key(Key)).
```

The **get_keys_for_replicas**/1 implements symmetric replication, here. The call to **normalize** implements the modulo by throwing high bits away.

File **rt_simple.erl**:

```
%% @doc
-spec(dump/1 :: (rt()) -> ok).
dump(_State) ->
    ok.
```

dump/1 is not implemented.

7.2 Chord routing table

The file **rt_chord.erl** implements Chord's routing.

7.2.1 Data types

File **rt_chord.erl**:

```
-type(key()::pos_integer()).
-ifdef(types_are_builtin).
-type(rt()::gb_tree()).
-type(external_rt()::gb_tree()).
-type(dict_type() :: dict()).
-else.
-type(rt()::gb_trees:gb_tree()).
-type(external_rt()::gb_trees:gb_tree()).
-type(dict_type() :: dict:dictionary()).
-endif.
-type(index() :: {pos_integer(), pos_integer()}).

-type(state()::rt()).
```

The routing table is a **gb_tree**. Identifiers in the ring are integers. Note, that in Erlang integer can be of arbitrary precision. For Chord, the identifiers are in $[0, 2^{128})$, i.e. 128-bit strings.

7.2.2 The routingtable behaviour for Chord

File **rt_chord.erl**:

```
%% @doc creates an empty routing table.
-spec(empty/1 :: (node:node_type()) -> rt()).
empty(_Succ) ->
    gb_trees:empty().
```

empty/1 returns an empty **gb_tree**.
hash_key(**Key**) and **getRandomNodeId** call their counterparts from **rt_simple.erl**

File **rt_chord.erl**:

```
%% @doc returns the next hop to contact for a lookup
%%      Note, that this code will be called from the cs_node process and
%%      it will have an external_rt!
-spec(next_hop/2 :: (cs_state:state(), key()) -> cs_send:mypid()).
next_hop(State, Id) ->
    case util:is_between(cs_state:id(State), Id, cs_state:succ_id(State)) of
        %succ is responsible for the key
        true ->
            cs_state:succ_pid(State);
        % check routing table
        false ->
            case util:gb_trees_largest_smaller_than(Id, cs_state:rt(State)) of
                nil ->
                    cs_state:succ_pid(State);
                {value, _Key, Value} ->
                    Value
            end
    end.
```

next_hop traverses the routing table beginning with the longest finger (2^{127}) by calling the helper function **next_hop**/5.

File **rt_chord.erl**:

```
```

If the entry exists, it is retrieved from the **gb_tree**. If the id of the routing table entry is between ourselves and the searched id, the finger is chosen. If anything fails, **Candidate** (the successor) is chosen.
Why could a routing table entry be **null**? **filterDeadNodes** changes entries to **null**.

BUG: Instead of directly returning **Candidate** one should further traverse the routing table for shorter appropriate fingers. If doing so, a check whether **Index** is zero, would become necessary.

If the finger is to long, recursively try the next shorter finger.

File **rt_chord.erl**:

```
%% @doc starts the stabilization routine
-spec(init_stabilize/3 :: (key(), node:node_type(), rt()) -> rt()).
init_stabilize(Id, _Succ, RT) ->
    % calculate the longest finger
    Key = calculateKey(Id, first_index()),
    % trigger a lookup for Key
    cs_lookup:unreliable_lookup(Key, {rt_get_node, cs_send:this(), first_index()}),
    RT.
```

The routing table stabilization is triggered with the index 127 and then runs asynchronously, as we do not want to block the **rt_loop** to perform other request while recalculating the routing table. We have to find the node responsible for the calculated finger and therefore perform a lookup for the node with a **rt_get_node** message, including a reference to ourselves as the reply-to address and the index to be set.

The lookup performs an overlay routing by passing the massage until the responsible node is found. There, the message is delivered to the **cs_node**. At the destination the message is handled in **cs_node.erl**:

File **cs_node.erl**:

```
on({rt_get_node, Source_PID, Cookie}, State) ->
    cs_send:send(Source_PID, {rt_get_node_response, Cookie, cs_state:me(State)}),
    State;
```

The remote node just sends the requested information back directly in a **rt_get_node_response** message including a reference to itself. When receiving the routing table entry, we call **stabilize**/5.

File **rt_chord.erl**:

```
%% @doc updates one entry in the routing table
%%      and triggers the next update
-spec(stabilize/5 :: (key(), node:node_type(), rt(), pos_integer(),
                      node:node_type()) -> rt()).
stabilize(Id, Succ, RT, Index, Node) ->
    case node:is_null(Node)                         % do not add null nodes
        orelse (node:id(Succ) == node:id(Node))       % there is nothing shorter than succ
        orelse (util:is_between(Id, node:id(Node), node:id(Succ))) of % there should not be anything shorter t
        true ->
            RT;
        false ->
            NewRT = gb_trees:enter(Index, Node, RT),
            Key = calculateKey(Id, next_index(Index)),
            cs_lookup:unreliable_lookup(Key, {rt_get_node, cs_send:this(),
                                              next_index(Index)}),
            NewRT
    end.
```

stabilize/5 assigns the received routing table entry and triggers to fill the next shorter one using the same mechanisms as described.

When the shortest finger is the successor, then filling the routing table is stopped, as no further new entries would occur. It is not necessary, that **Index** reaches 1 to make that happen. If less than 2^{128} nodes participate in the system, it may happen earlier.

filterDeadNode removes dead entries from the **gb_tree**.

File **rt_chord.erl**:

```
%% @doc remove all entries
-spec(filterDeadNode/2 :: (rt(), cs_send:mypid()) -> rt()).
filterDeadNode(RT, DeadPid) ->
    DeadIndices = [Index|| {Index, Node}  <- gb_trees:to_list(RT),
                           node:pidX(Node) == DeadPid],
    lists:foldl(fun (Index, Tree) -> gb_trees:delete(Index, Tree) end,
                RT, DeadIndices).
```

8 Directory Structure of the Source Code

The directory tree of Scalaris is structured as follows:

bin	contains shell scripts needed to work with Scalaris (e.g. start the boot services, start a node, . . .)
contrib	necessary third party packages (yaws and log4erl)
doc	generated erlang documentation
docroot	root directory of the bootserver's webserver
docroot_node	root directory of the normal node's webserver
ebin	the compiled Erlang code (beam files)
java-api	a java api to Scalaris
log	log files
src	contains the Scalaris source code
test	unit tests for Scalaris
user-dev-guide	contains the sources for this document

9 Java API

For the Java API documentation, we refer the reader to Javadoc resp. doxygen. The following commands create the documentation:

```
%> cd java-api
%> ant doc
%> doxygen
```

The Javadoc can be found in **`java-api/doc/index.html`**. The doxygen is in **`doc-doxygen/html/index.html`**.
We provide two kinds of APIs:

- high-level access with **`de.zib.scalaris.Scalaris`**
- low-level access with **`de.zib.scalaris.Transaction`**

The former provides general functions for reading and writing single key-value pairs and an API for the built-in PubSub-service. The latter allows the user to write custom transactions which can modify an arbitrary number of key-value pairs within one transaction.

Bibliography

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

[2] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, 2006.