Scalaris

Users and Developers Guide

Version 0.1

Florian Schintke, Thorsten Schütt

April 20, 2009

Contents

Part I

Users Guide

1 Download and Installation

1.1 Requirements

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

- Erlang R12
- Erlang OTP (included in Erlang R12)
- Erlang yaws
- GNU Make

Note, the Version 12 of Erlang is required. Scalaris will not work with older versions.
To build the Java API 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.

1.2 Download

The sources can be obtained from http://code.google.com/p/scalaris.

1.2.1 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
```

1.2.2 Releases

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

1.3 Configuration

Scalaris reads the 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**. A local configuration file is necessary to run Scalaris on distributed nodes:

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}}.
```

boot_host defines the node where the boot server is running, which is contacted to join the system.

1.4 Build

1.4.1 Linux

Read the file **README** in the main Scalaris checkout directory.

1.4.2 Java-API

The following commands will build the Java API for Scalaris:

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

This will build chordsharp4j.jar, which is the library for accessing the overlay network. Optionally, the documentation can be build:

```
%> ant doc
```

1.5 Running Scalaris

In Scalaris there are two kinds of processes:

- boot servers
- regular servers

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.

1.5.1 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.
The boot server should show output similar to the following, when starting the first Scalaris nodes. The first line is printed when the Scalaris is spawned. Afterwards it will try to connect the boot server. When the third line is printed, it managed to contact the boot server and joined the ring. In this case, it was the first node in the ring.

```
[ I | Node    | <0.97.0> ] joining "23947834870"
[ I | Node    | <0.97.0> ] join as first [50,51,57,52,55,56,51,52,56,55,48]
[ I | Node    | <0.97.0> ] joined
```

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.

1.5.2 Running distributed

Scalaris can be installed on other machines in the same way as described in Sect. 1.6. Please make sure, that the **scalaris.local.cfg** refers to available boot servers. 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.

1.5.3 Running on PlanetLab

1.5.4 Replication Degree

1.5.5 Routing Scheme

1.6 Installation

Note: there is no **make install** at the moment! The nodes have to be started from the bin directory.

2 Using the system

2.1 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" }
      ]
```

← 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" }
```

2.2 Erlang

2.3 Java command line interface

The jar file contains a small command line interface client.

```
%> java -jar chordsharp4j.jar -help
usage: chordsharp
 -getsubscribers <topic>   get subscribers of a topic
 -help                     print this message
 -publish <params>         publish a new message for a topic: <topic>
                           <message>
 -read <key>               read an item
 -subscribe <params>       subscribe to a topic: <topic> <url>
 -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.

```
%> java -jar chordsharp4j.jar -write foo bar
write(foo, bar)
%> java -jar chordsharp4j.jar -read foo
read(foo) == bar
```

The chordsharp4j 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.

2.4 Java API

3 Testing the system

3.1 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 use the **`common_test`** package from the Erlang system.

Part II

Developers Guide

4 How a node joins the system

4.1 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.

4.2 Starting additional local nodes after boot

After booting a new Scalaris-System as described in Section 1.5.1 on page 8, ten additional local nodes can be started by typing **admin:add_nodes**(10) in the Erlang-Shell that the boot process opened.

```
scalaris/bin> ./boot.sh
[...]
=INFO REPORT==== 18-Jul-2008::20:08:48 ===
Yaws: Listening to 0.0.0.0:8000 for servers
 - http://localhost:8000 under ../docroot
[ I | Fail    | <0.102.0> ] start detector
this() == {{127,0,0,1},14195}
[ I | Node    | <0.105.0> ] joining "22500913918"
[ I | Node    | <0.105.0> ] join as first [50,50,53,48,48,57,49,51,57,49,56]
[ I | Node    | <0.105.0> ] joined

(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:**

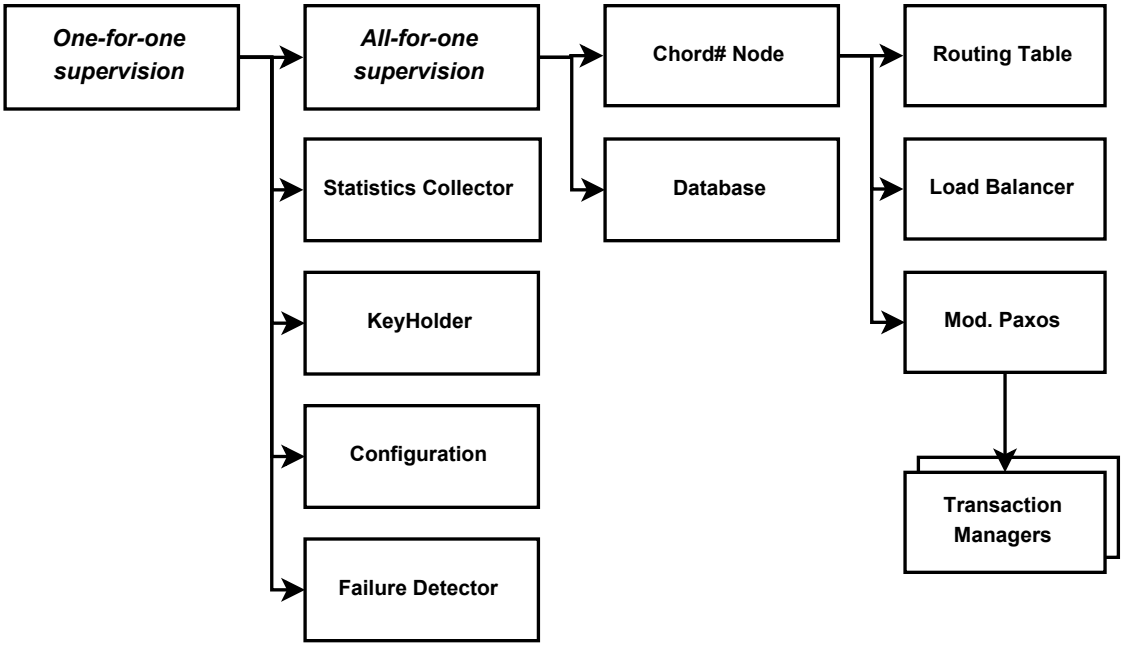
```
37 %%------------------------------------------------------------------------
38 %% Function: add_nodes(int()) -> ok
39 %% Description: add new Scalaris nodes
40 %%------------------------------------------------------------------------
41 % @doc add new Scalaris nodes on the local node
42 % @spec add_nodes(int()) -> ok
43
44 add_nodes(Count) ->
45     add_nodes(Count, 0).
46
47 % @spec add_nodes(int(), int()) -> ok
48 add_nodes(Count, Delay) ->
49     add_nodes_loop(Count, Delay).
50
51 add_nodes_loop(0, _) ->
```

```
52      ok;
53  add_nodes_loop(Count, Delay) ->
54      supervisor:start_child(main_sup, {randoms:getRandomId(),
55                                        {cs_sup_or, start_link, []},
56                                        permanent,
57                                        brutal_kill,
58                                        worker,
59                                        []}),
60      timer:sleep(Delay),
61      add_nodes_loop(Count - 1, Delay).
```

It first initializes the random number generator and then calls **`add_nodes_loop(Count)`**.
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.

4.2.1 Supervisor-tree of a Scalaris node

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



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

The supervisor mechanism first calls the **`init()`** function of the defined module (**`cs_sup_or:init()`** in this case) before calling the defined start function (**`start_link`** here.
So, lets have a look at **`cs_sup_or:init`**, the 'Scalaris *or* supervisor'.

File **`cs_sup_or.erl`**:

```
61  init([Options]) ->
62      InstanceId = string:concat("cs_node_", randoms:getRandomId()),
63      boot_server:connect(),
64      KeyHolder =
65          {cs_keyholder,
66           {cs_keyholder, start_link, [InstanceId]},
67           permanent,
68           brutal_kill,
```

```
69          worker,
70          []},
71      RSE =
72          {rse_chord,
73           {rse_chord, start_link, [InstanceId]},
74           permanent,
75           brutal_kill,
76           worker,
77           []},
78      Supervisor_AND =
79          {cs_supervisor_and,
80           {cs_sup_and, start_link, [InstanceId, Options]},
81           permanent,
82           brutal_kill,
83           supervisor,
84           []},
85      RingMaintenance =
86          {?RM,
87           {?RM, start_link, [InstanceId]},
88           permanent,
89           brutal_kill,
90           worker,
91           []},
92      RoutingTable =
93          {routingtable,
94           {rt_loop, start_link, [InstanceId]},
95           permanent,
96           brutal_kill,
97           worker,
98           []},
99      DeadNodeCache =
100         {deadnodecache,
101          {dn_cache, start_link, [InstanceId]},
102          permanent,
103          brutal_kill,
104          worker,
105          []},
106     {ok, {{one_for_one, 10, 1},
107           [
108            KeyHolder,
109       DeadNodeCache,
110           RingMaintenance,
111           RoutingTable,
112           Supervisor_AND,
113           RSE
114          ]}}.
```

The **init**() function must return process descriptions that the supervisor should start, and how the processes have to be observed. Here, we define a list of processes to be observed by a **one_for_one** supervisor. The processes are: **Config**, **KeyHolder**, **MessageStatisticsCollector** and a **Supervisor_AND** process.
The term **{one_for_one, 10, 1}** defines that the supervisor should retry 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 their corresponding **init**() functions and afterwards, 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, which was the last one in the list. At that point in time, all other defined processes are already started and running.

4.2.3 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:**

```
58 init([InstanceId, Options]) ->
59     Node =
60         {cs_node,
61          {cs_node, start_link, [InstanceId, Options]},
62          permanent,
63          brutal_kill,
64          worker,
65          []},
66     DB =
67         {?DB,
68          {?DB, start_link, [InstanceId]},
69          permanent,
70          brutal_kill,
71          worker,
72          []},
73     Cyclon =
74         {cyclon,
75          {cyclon.cyclon, start_link, [InstanceId]},
76          permanent,
77          brutal_kill,
78          worker,
79          []},
80     {ok, {{one_for_all, 10, 1},
81           [
82            DB,
83            Node,
84            Cyclon
85           ]}}.
```

It defines two 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.
After the **cs_sup_and:init()** finished, the DB is initialized and afterwards **cs_node:start_link** is called by the and-supervisor. We only go into details here, for the latter.

File **cs_node.erl:**

```
376 %% @doc spawns a scalaris node, called by the scalaris supervisor process
377 %% @spec start_link(term()) -> {ok, pid()}
378 start_link(InstanceId) ->
379     start_link(InstanceId, []).
380
381 start_link(InstanceId, Options) ->
382     gen_component:start_link(?MODULE, [InstanceId, Options], [{register, InstanceId, cs_node}]).
```

This spawns a new process, that executes **cs_node:start()** and maintains a link to that process. This function is called in the context of the and-supervisor process, so the and-supervisor process will be informed, when the spawned process finishes. After the spawn is submitted, **cs_node:start_link** returns to the supervisor, which starts observing the processes in a loop.
Note: **?MODULE** is a predefined Erlang macro, which expands to the module name, the code belongs to (here: **cs_node**).

4.2.4 Initializing a cs_node-process

File **cs_node.erl:**

```
354 %% @doc joins this node in the ring and calls the main loop
355 -spec(init/1 :: ([any()]) -> cs_state:state()).
```

```
356 init([_InstanceId, Options]) ->
357     case lists:member(first, Options) of
358         true ->
359             ok;
360         false ->
361             timer:sleep(crypto:rand_uniform(1, 100) * 100)
362     end,
363     Id = cs_keyholder:get_key(),
364     {First, State} = cs_join:join(Id),
365     if
366         not First ->
367             cs_replica_stabilization:recreate_replicas(cs_state:get_my_range(State));
368         true ->
369             ok
370     end,
371     log:log(info,"[ Node ~w ] joined",[self()]),
372     State.
```

A **cs_node** first registers itself in the process dictionary. Then, the process sleeps for a random amount of time. Otherwise, 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.
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)?
Next, the **cs_node** registers itself as the owner of the failuredetector **failuredetector:set_owner**(**self**()) to become informed, when the failuredetector detects failing nodes.
With **boot_server:connect**() a connection to the boot-server is established.

4.2.5 Actually joining the ring

cs_join:join is called next in **cs_node:start**().

File **cs_join.erl**:

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

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.
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**:

```
50 %% @doc join an empty ring
51 join_first(Id) ->
52         log:log(info,"[ Node ~w ] join as first ~w",[self(), Id]),
53     Me = node:make(cs_send:this(), Id),
54     ?RM:initialize(Id, Me, Me, Me),
55     routingtable:initialize(Id, Me, Me),
56     cs_state:new(?RT:empty(Me), Me, Me, Me, {Id, Id}, cs_lb:new(), ?DB:new()).
```

The macro `?RT` maps to the configured routing algorithm. It is defined in **chordsharp.erl**. For further details on the routing see Chapter 5 on page 27.
The state is defined in

File **cs_state.erl:**

```
57 new(RT, Successor, Predecessor, Me, MyRange, LB, DB) ->
58     #state{
59      routingtable = RT,
60      successor = Successor,
61      predecessor = Predecessor,
62      me = Me,
63      my_range = MyRange,
64      lb=LB,
65      join_time=now(),
66      deadnodes = gb_sets:new(),
67      trans_log = #translog{
68        tid_tm_mapping = dict:new(),
69        decided = gb_trees:empty(),
70        undecided = gb_trees:empty()
71       },
72      db = DB
73     }.
```

If a node joins an existing ring, **reliable_get_node** for the own **Id** is called in **cs_join:join()**. This lookup delivers the successor for the joining node. The node, that is currently responsible for **Id**, but which has a larger **Id** itself. 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:**

```
61 join_ring(Id, Succ) ->
62     log:log(info,"[ Node ~w ] join_ring ~w",[self(), Id]),
63     Me = node:make(cs_send:this(), Id),
64     UniqueId = node:uniqueId(Me),
65     cs_send:send(node:pidX(Succ), {join, cs_send:this(), Id, UniqueId}),
66     receive
67         {join_response, Pred, Data} ->
68             log:log(info,"[ Node ~w ] got pred ~w",[self(), Pred]),
69             case node:is_null(Pred) of
70                 true ->
71                     DB = ?DB:add_data(?DB:new(), Data),
72                     ?RM:initialize(Id, Me, Pred, Succ),
73                     routingtable:initialize(Id, Pred, Succ),
74                     cs_state:new(?RT:empty(Succ), Succ, Pred, Me, {Id, Id}, cs_lb:new(), DB);
75                 false ->
76                     cs_send:send(node:pidX(Pred), {update_succ, Me}),
77                     DB = ?DB:add_data(?DB:new(), Data),
78                     ?RM:initialize(Id, Me, Pred, Succ),
79                     routingtable:initialize(Id, Pred, Succ),
80                     cs_state:new(?RT:empty(Succ), Succ, Pred, Me, {node:id(Pred), Id},
81                                  cs_lb:new(), DB)
82             end
83     end.
```

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**:

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

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

File **cs_join.erl**:

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

This resets the local predecessor to the new node and splits the data. Then it sends a **join_response** to the new node with its former predecessor, the data, it has to host, and its successorlist. Additionally, the new node will be observed by the failuredetector.
Note: Here, the observation of the former predecessor could be deleted from the failure detector. But periodically, all known hosts are reregistered with the failure detector, and then all other nodes are thrown away from the list of nodes to observe in the failure detector. So, we do not care here.
Finally, the new predecessor is set and the **join_request()** is finished.
Back on the new 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 register the predecessor, to register all new nodes in the failure detector, to send a message to the predecessor, that its successor has changed.
Finally, all data are added to the database and the succlist is build by putting the successor to the head of the successor list of the successor (old node): **[Succ | SuccList]**.
The next step in **cs_join:join()** is to call **cs_reregister:reregister** which reregisters the node periodically with the boot server. Then **{false, State}** is returned, which means 'this was not the first node in the ring'. Now, the **cs_join:join()** is completed.

4.2.6 Beginning to serve requests

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

File **cs_node.erl**:

```
354 %% @doc joins this node in the ring and calls the main loop
355 -spec(init/1 :: ([any()]) -> cs_state:state()).
356 init([_InstanceId, Options]) ->
357     case lists:member(first, Options) of
358         true ->
359             ok;
360         false ->
361             timer:sleep(crypto:rand_uniform(1, 100) * 100)
362     end,
363     Id = cs_keyholder:get_key(),
364     {First, State} = cs_join:join(Id),
365     if
366         not First ->
367             cs_replica_stabilization:recreate_replicas(cs_state:get_my_range(State));
368         true ->
369             ok
370     end,
371     log:log(info,"[ Node ~w ] joined",[self()]),
372     State.
```

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 all the self-management services are started with **timer:after** and the loop for request handling is started.

4.2.7 FAQ

Question: How are the routing-table, load-balancing and paxos processes started, that can be seen in the supervisor tree?

Answer: They are currently not implemented as separate Erlang processes. All the requests are handled by the loop started by **cs_node:start()**. Nevertheless, they could be implemented as separate processes to make the software architecture cleaner.

5 Routing and routing tables in the Overlay

2008-07-25, SVN revision 12

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 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. With the function **util:is_between** it is decided, 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, like a Java interface. In Erlang interfaces can be defined using a 'behaviour'. The files **rt_simple** and **rt_chord** implement the behaviour 'routingtable'.
The macro ?RT in other files replaces to the active module, which is defined in **chordsharp.hrl**.

File **chordsharp.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},
     % key space -> identifier space
     {hash_key, 1}, {getRandomNodeId, 0},
     % routing
     {next_hop, 2},
     % trigger for new stabilization round
     {stabilize, 1},
     % dead nodes filtering
     {filterDeadNodes, 2},
     % statistics
     {to_pid_list, 1}, {to_node_list, 1}, {get_size, 1},
     % for symmetric replication
     {get_keys_for_replicas, 1},
     % for debugging
     {dump, 1}
    ];

behaviour_info(_Other) ->
    undefined.
[...]
```

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 translates it to an id for the overlay.

getrandomnodeid/0 returns a random node id.

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

stabilize/1 is called periodically to rebuild the routing table.

filterDeadNodes 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.

to_node_list/1 get all nodes in the routing table as complete reference including IP, range, etc.

get_size/1 get the size of the routing table.

get_keys_for_replicas/1 Returns for a given **Key** the keys of its replicas. Thereby one can implement successorlist replication or symmetric replication or any other schema.

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

5.1 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.

5.1.1 Data types

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

File **rt_simple.erl:**

```
[...]
%% @type rt() = {node:node(), gb_trees:gb_tree()}. Sample routing table
%% @type key() = int(). Identifier
[...]
```

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

5.1.2 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(node:node()) -> rt()
empty(Succ) ->
    {Succ, gb_trees:empty()}.
[...]
```

The empty routing table consists of the successor.

File **rt_simple.erl:**

```
[...]
%% @doc hashes the key to the identifier space.
%% @spec hash_key(string()) -> 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).

%% @doc generates a random node id
%%      In this case it is a random 128-bit string.
%% @spec getRandomNodeId() -> key()
getRandomNodeId() ->
    % generates 128 bits of randomness
    hash_key(integer_to_list(random:uniform(65536 * 65536))).
[...]
```

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(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 triggers a new stabilization round
%% @spec stabilize(cs_state:state()) -> cs_state:state()
stabilize(State) ->
```

```
    % trigger the next stabilization round
    timer:send_after(config:pointerStabilizationInterval(), self(), {stabilize_pointers}),
    % renew routing table
    cs_state:set_rt(State, empty(cs_state:succ(State))).
[...]
```

The **stabilize**/1 first triggers to be called again after the pointer stabilization interval defined in the configuration. Then it resets its routing table with the current successor.

File **rt_simple.erl:**

```
[...]
%% @doc removes dead nodes from the routing table
%% @spec filterDeadNodes(rt(), [node:node()]) -> rt()
filterDeadNodes(RT, _DeadNodes) ->
    RT.
[...]
```

FilterDeadNodes/2 does nothing, as only the successor is listed in the routing table and that is reset periodically in **stabilize**/1.

File **rt_simple.erl:**

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

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

File **rt_simple.erl:**

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

Return whole node definitions as defined in **node.erl**

File **node.erl:**

```
[...]
-record(node, {pid, id, uniqueId}).
[...]
```

File **node.erl:**

```
[...]
%% @doc returns the size of the routing table.
%%      inefficient standard implementation
%% @spec get_size(rt()) -> int()
get_size(RT) ->
    length(to_pid_list(RT)).

%% @doc returns the replicas of the given key
%% @spec get_keys_for_replicas(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)).

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

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

File **node.erl**:

```
[...]
%% @doc
%% @spec dump(cs_state:state()) -> term()
dump(_State) ->
    ok.
[...]
```

dump/1 is not implemented.

5.2 Chord routing table

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

5.2.1 Data types

File **node.erl**:

```
[...]
%% @type rt() = gb_trees:gb_tree(). Chord routing table
%% @type key() = int(). Identifier
[...]
```

The routing table is a **gb_tree**. Identifiers in the ring are integers.

5.2.2 The routingtable behaviour for Chord

File **node.erl**:

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

This function should return **_Succ**?

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
%% @spec next_hop(cs_state:state(), key()) -> node()
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 ->
```

```
            RT = cs_state:rt(State),
            next_hop(cs_state:id(State), RT, Id, 127, cs_state:succ_pid(State))
    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**:

```
[...]
% @private
next_hop(N, RT, Id, Index, Candidate) ->
    case gb_trees:is_defined(Index, RT) of
        true ->
            case gb_trees:get(Index, RT) of
                null ->
                    Candidate;
                Entry ->
                    case util:is_between_closed(N, node:id(Entry), Id) of
                        true ->
                            node:pidX(Entry);
                        false ->
                            next_hop(N, RT, Id, Index - 1, Candidate)
                    end
            end;
        false ->
            Candidate
    end.
[...]
```

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, recursivly try the next shorter finger.

File **rt_chord.erl**:

```
[...]
%% @doc starts the stabilization routine
%% @spec stabilize(cs_state:state()) -> cs_state:state()
stabilize(State) ->
    % trigger the next stabilization round
    timer:send_after(config:pointerStabilizationInterval(), self(), {stabilize_pointers}),
    % calculate the longest finger
    Key = calculateKey(State, 127),
    % trigger a lookup for Key
    cs_lookup:unreliable_lookup(Key, {rt_get_node, cs_send:this(), 127}),
    State.
[...]
calculateKey(State, Idx) ->
    % n + 2^Idx
    rt_simple:normalize(cs_state:id(State) + (1 bsl Idx)).
```

The routing table stabilization is triggered with the index 127 and then runs asynchronously, as we do not want to block the **cs_node** 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**:

```
[...]
        % for chord_rt
        {rt_get_node, Source_PID, Idx} ->
            cs_send:send(Source_PID, {rt_get_node_response, Idx, cs_state:me(State)}),
            loop(State, ?DEBUG(Debug));

        {rt_get_node_response, Index, Node} = _Message ->
            NewState = ?RT:stabilize(State, Index, Node),
            loop(NewState, ?DEBUG(cs_debug:debug(Debug, NewState, _Message)));
[...]
```

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**/3!, which is different from the **stabilize**/1 above.

File **rt_chord.erl**:

```
[...]
stabilize(State, Index, Node) ->
    RT = cs_state:rt(State),
    case node:is_null(Node) of
        true ->
            State;
        false ->
            case cs_state:succ(State) == Node of
                true ->
                    State;
                false ->
                    NewRT = gb_trees:enter(Index, Node, RT),
                    Key = calculateKey(State, Index - 1),
                    self() ! {lookup_aux, Key, {rt_get_node, cs_send:this(), Index - 1}},
                    cs_state:set_rt(State, NewRT)
            end
    end.
[...]
```

stabilize/3 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.
filterDeadNodes updates the **gb_tree** accordingly.

File **rt_chord.erl**:

```
[...]
%% @doc remove all entries with the given ids
%% @spec filterDeadNodes(gb_trees:gb_tree(), term()) -> gb_trees:gb_tree()
filterDeadNodes(RT, DeadNodes) ->
    gb_sets:fold(fun (Id, RT2) -> filter_intern(gb_trees:iterator(RT2), RT2, Id) end, RT, DeadNodes).

% @private
filter_intern(nil, RT, _) ->
    RT;
filter_intern([], RT, _) ->
    RT;
filter_intern(Iterator, RT, Id) ->
    {Index, Value, Next} = gb_trees:next(Iterator),
```

```
    case node:uniqueId(Value) == Id of
        true ->
            filter_intern(Next, gb_trees:delete(Index, RT), Id);
        false ->
            filter_intern(Next, RT, Id)
    end.


%% @doc returns the pids of the routing table entries .
%% @spec to_pid_list(rt()) -> [pid()]
to_pid_list(RT) ->
    lists:map(fun ({_Idx, Node}) -> node:pidX(Node) end, gb_trees:to_list(RT)).

%% @doc returns the pids of the routing table entries .
%% @spec to_node_list(rt()) -> [node:node()]
to_node_list(RT) ->
    lists:map(fun ({_Idx, Node}) -> Node end, gb_trees:to_list(RT)).

%% @doc returns the size of the routing table.
%%      inefficient standard implementation
%% @spec get_size(rt()) -> int()
get_size(RT) ->
    length(to_pid_list(RT)).

%% @doc returns the replicas of the given key
%% @spec get_keys_for_replicas(key() | string()) -> [key()]
get_keys_for_replicas(Key) ->
    rt_simple:get_keys_for_replicas(Key).
[...]
```

Also the remaining functions are implemented as one would expect.

6 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
doc	generated erlang documentation
docroot	root directory of the integrated webserver
java-api	a java api to Scalaris
log	log files
src	contains the Scalaris source code
tests	unit tests for Scalaris
user-dev-guide	contains the sources for this document

7 System Components

8 Processes

9 Troubleshooting

9.1 ApplicationMonitor appmon:start()

A Java API

A.1 de.zib.chordsharp.ChordSharp

```
public class ChordSharp
```

Public ChordSharp Interface.

Version: 1.1

Author: Nico Kruber, kruber@zib.de

Method Summary	
static Vector<String>	**getSubscribers**(String topic) Gets a list of subscribers of a topic.
static void	**publish**(String topic, String content) Publishes an event under a given topic.
static String	**read**(String key) Gets the value stored with the given key.
static void	**subscribe**(String topic, String url) Subscribes a url for a topic.
static void	**write**(String key, String value) Stores the given key/value pair.

read

```
public static String read(String key)
                  throws ConnectionException,
                         TimeoutException,
                         UnknownException,
                         NotFoundException
```

Gets the value stored with the given key.

Parameters: key - the key to look up

Returns: the value stored under the given key

Throws:

ConnectionException	if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie
TimeoutException	if a timeout occurred while trying to fetch the value
NotFoundException	if the requested key does not exist
UnknownException	if any other error occurs

write

```
public static void write(String key,
                         String value)
                  throws ConnectionException,
                         TimeoutException,
                         UnknownException
```

Stores the given key/value pair.

Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

publish

```
public static void publish(String topic,
                           String content)
                    throws ConnectionException
```

Publishes an event under a given topic.

Parameters: topic - the topic to publish the content under content - the content to publish

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

subscribe

```
public static void subscribe(String topic,
                             String url)
                      throws ConnectionException
```

Subscribes a url for a topic.

Parameters: topic - the topic to subscribe the url for url - the url of the subscriber (this is where the events are send to)

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

getSubscribers

```
public static Vector<String> getSubscribers(String topic)
                                     throws ConnectionException,
                                            UnknownException
```

Gets a list of subscribers of a topic.

Parameters: topic - the topic to get the subscribers for

Returns: the subscriber URLs

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie UnknownException - is thrown if the return type of the erlang method does not match the expected one

de.zib.chordsharp.Transaction

```
public class \textbf{Transaction}
```

Provides means to realise a transaction with the chordsharp ring using Java.

It reads the connection parameters from a file called ChordSharpConnection.properties or uses default properties defined in ChordSharpConnection.defaultProperties.

```
OtpErlangString otpKey;
OtpErlangString otpValue;
OtpErlangString otpResult;
String key;
String value;
String result;

Transaction transaction = new Transaction(); // Transaction()
transaction.start(); // start()
transaction.write(otpKey, otpValue); // write(OtpErlangString, OtpErlangString)
transaction.write(key, value); // write(String, String)
otpResult = transaction.read(otpKey); //read(OtpErlangString)
result = transaction.read(key); //read(String)
transaction.commit(); // commit()
```

For more examples, have a look at TransactionReadExample, TransactionParallelReadsExample, TransactionWriteExample and TransactionReadWriteExample.

Attention:

If a read or write operation fails within a transaction all subsequent operations on that key will fail as well. This behaviour may particularly be undesirable if a read operation just checks whether a value already exists or not. To overcome this situation call revertLastOp() immediately after the failed operation which restores the state as it was before that operation.

The TransactionReadWriteExample example shows such a use case.

Version: 1.0

Author: Nico Kruber, kruber@zib.de

Constructor Summary
Transaction() Creates the object's connection to the chordsharp node specified in the "ChordSharpConnection.properties" file.

Method Summary	
void	**abort**() Cancels the current transaction.
void	**commit**() Commits the current transaction.
OtpErlangString	**read**(OtpErlangString key) Gets the value stored under the given key.
String	**read**(String key) Gets the value stored under the given key.
void	**revertLastOp**() Reverts the last (read or write) operation by restoring the last state.
void	**start**() Starts a new transaction by generating a new transaction log.
void	**write**(OtpErlangString key, OtpErlangString value) Stores the given key/value pair.
void	**write**(String key, String value) Stores the given key/value pair.

Transaction

```
public Transaction()
            throws ConnectionException
```

Creates the object's connection to the chordsharp node specified in the "ChordSharpConnection.properties" file.

Throws: ConnectionException - if the connection fails

start

```
public void start()
           throws ConnectionException,
                  TransactionNotFinishedException,
                  UnknownException
```

Starts a new transaction by generating a new transaction log.

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TransactionNotFinishedException - if an old transaction is not finished (via commit() or abort()) yet UnknownException - if the returned value from erlang does not have the expected type/structure

commit

```
public void commit()
            throws UnknownException,
                   ConnectionException
```

Commits the current transaction. The transaction's log is reset if the commit was successful, otherwise it still retains in the transaction which must be successfully committed or aborted in order to be restarted.

Throws: UnknownException - If the commit fails or the returned value from erlang is of an unknown type/structure, this exception is thrown. Neither the transaction log nor the local operations buffer is emptied, so that the commit can be tried again. ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

See Also: abort()

abort

```
public void abort()
```

Cancels the current transaction.
For a transaction to be cancelled, only the transLog needs to be reset. Nothing else needs to be done since the data was not modified until the transaction was committed.

read

```
public OtpErlangString read(OtpErlangString key)
                     throws ConnectionException,
                            TimeoutException,
                            UnknownException,
                            NotFoundException
```

Gets the value stored under the given key.

Parameters: key - the key to look up

Returns: the value stored under the given key

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to fetch the value NotFoundException - if the requested key does not exist UnknownException - if any other error occurs

read

```
public String read(String key)
            throws ConnectionException,
                   TimeoutException,
                   UnknownException,
                   NotFoundException
```

Gets the value stored under the given key.

Parameters: key - the key to look up

Returns: the value stored under the given key

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to fetch the value NotFoundException - if the requested key does not exist UnknownException - if any other error occurs

write

```
public void write(OtpErlangString key,
                  OtpErlangString value)
           throws ConnectionException,
                  TimeoutException,
                  UnknownException
```

Stores the given key/value pair.

Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

write

```
public void write(String key,
                  String value)
           throws ConnectionException,
                  TimeoutException,
                  UnknownException
```

Stores the given key/value pair.

Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

revertLastOp

```
public void revertLastOp()
```

Reverts the last (read, parallelRead or write) operation by restoring the last state. If no operation was initiated yet, this method does nothing.
This method is especially useful if after an unsuccessful read a value with the same key should be written which is not possible if the failed read is still in the transaction's log.

Bibliography

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