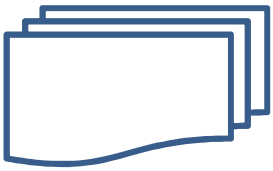
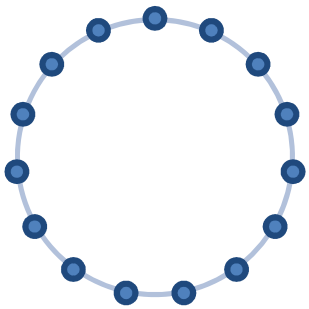
TRANSACTIONS





Scalaris:

Users and Developers Guide

Version 0.2.0 draft August 3, 2010

Contents

I. Users Guide 5

1. Introduction 6
   1.1. Brewer's CAP Theorem . . . 6

2. Download and Installation 8
   2.1. Requirements . . . 8
   2.2. Download . . . 8
      2.2.1. Development Branch . . . 8
      2.2.2. Releases . . . 8
   2.3. Configuration . . . 8
   2.4. Build . . . 9
      2.4.1. Linux . . . 9
      2.4.2. Windows . . . 9
      2.4.3. Java-API . . . 10
   2.5. Running Scalaris . . . 10
      2.5.1. Running on a local machine . . . 10
      2.5.2. Running distributed . . . 11
   2.6. Installation . . . 11
   2.7. Logging . . . 12

3. Using the system 13
   3.1. JSON API . . . 13
      3.1.1. Deleting a key . . . 16
   3.2. Java command line interface . . . 16
   3.3. Java API . . . 17

4. Testing the system 18
   4.1. Running the unit tests . . . 18

5. Troubleshooting 19
   5.1. Network . . . 19

II. Developers Guide 20

6. General Hints 21
   6.1. Coding Guidelines . . . 21
   6.2. Testing Your Modifications and Extensions . . . 21
   6.3. Help with Digging into the System . . . 21

7. System Infrastructure 22
   7.1. The Process Dictionary . . . 22

7.2. The Communication Layer comm . . . 22
7.3. The gen_component . . . 22
7.3.1. A basic gen_component including a message handler . . . 23
7.3.2. How to start a gen_component? . . . 23
7.3.3. When does a gen_component terminate? . . . 24
7.3.4. What happens when unexpected events / messages arrive? . . . 24
7.3.5. What if my message handler generates an exception or crashes the process? . . . 24
7.3.6. Changing message handlers and implementing state dependent message responsiveness as a state-machine . . . 25
7.3.7. Halting and pausing a gen_component . . . 25
7.3.8. Integration with process_dictionary: Redirecting events / messages to other gen_components . . . 25
7.3.9. Integration with fd_pinger: Replying to failure detectors . . . 26
7.3.10. The debugging interface of gen_component: Breakpoints and step-wise execution . . . 26
7.3.11. Future use and planned extensions for gen_component . . . 29
7.4. The Process' Database (pdb) . . . 29
7.5. Writing Unittests . . . 29
7.5.1. Plain unittests . . . 29
7.5.2. Randomized Testing using tester.erl . . . 29

8. Basic Structured Overlay **30**
8.1. Ring Maintenance . . . 30
8.2. T-Man . . . 30
8.3. Routing Tables . . . 30
8.3.1. The routing table process (rt_loop) . . . 32
8.3.2. Common methods for routing table implementations (rt_generic.hrl) . . . 33
8.3.3. Simple routing table (rt_simple) . . . 34
8.3.4. Chord routing table (rt_chord) . . . 37
8.4. Local Datastore . . . 40
8.5. Cyclon . . . 40
8.6. Vivaldi Coordinates . . . 40
8.7. Estimated Global Information (Gossiping) . . . 40
8.8. Load Balancing . . . 40
8.9. Broadcast Trees . . . 40

9. Transactions in Scalaris **41**
9.1. The Paxos Module . . . 41
9.2. Transactions using Paxos Commit . . . 41
9.3. Applying the Tx-Modules to replicated DHTs . . . 41

10. How a node joins the system **42**
10.1. General Erlang server loop . . . 42
10.2. Starting additional local nodes after boot . . . 42
10.2.1. Supervisor-tree of a Scalaris node . . . 43
10.2.2. Starting the or-supervisor and general processes of a node . . . 44
10.2.3. Starting the and-supervisor with a peer and its local database . . . 45
10.2.4. Initializing a dht_node-process . . . 46
10.2.5. Actually joining the ring . . . 46

11. Directory Structure of the Source Code **50**

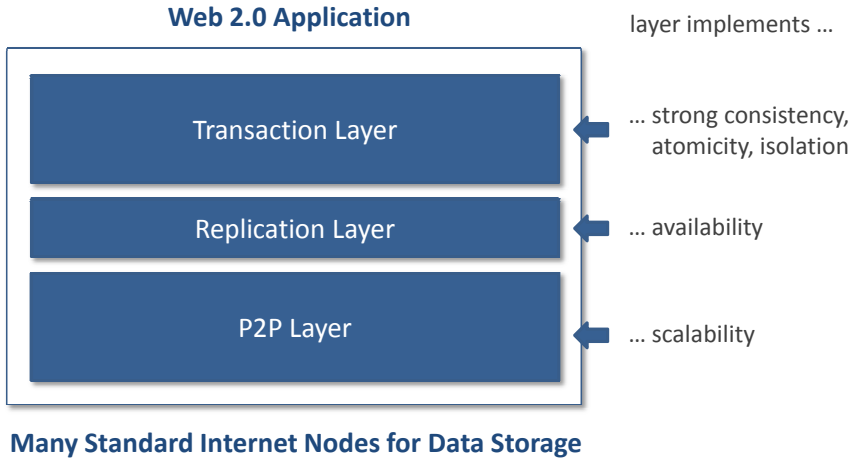
Part I.

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 was 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/scalarix.html`.

1.1. Brewer's CAP Theorem

In distributed computing there exists the so called CAP theorem. It basically says that there are three desirable properties for distributed systems but one can only have 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. Later on the networks may re-join 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.

The goal of Scalaris is 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 Scalaris with 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 R13B01 or newer
- GNU-like Make

To build the Java API (and the command-line client) the following modules are required additionally:

- Java Development Kit 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 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 and DEBs 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 provided as `bin/scalaris.local.cfg.example`. To run Scalaris distributed over several nodes, each node requires a `bin/scalaris.local.cfg`:

File `scalaris.local.cfg`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for distributed Erlang
% (see scalaris.hrl to switch)

% {boot_host, {boot,'boot@foo.bar.com'}}.
% {known_hosts, [{service_per_vm, 'boot@foo.bar.com'}]}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for TCP mode.
% (see scalaris.hrl 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 a node which is already in the system
{known_hosts, [{{127,0,0,1},14195, service_per_vm}]}.
```

Scalaris currently distinguishes 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 scalaris nodes. It seems to work but we make no guarantees.

- Install Erlang
  http://www.erlang.org/download.html
- Install OpenSSL (for crypto module)
  http://www.slproweb.com/products/Win32OpenSSL.html
- Checkout scalaris code from SVN
- adapt the path to your Erlang installation in `build.bat`
- start a `cmd.exe`
- go to the scalaris directory
- run `build.bat` in the cmd window

- check that there were no errors during the compilation; warnings are fine
- go to the bin sub-directory
- adapt the path to your Erlang installation in `boot.bat`, `cs_local.bat`, `cs_local2.bat` and `cs_local3.bat`
- run `boot.bat` or one of the other start scripts in the cmd window

`build.bat` will generate a `Emakefile` if there is none yet. If you have Erlang < R13B04, you will need to adapt the `Emakefile`. There will be empty lines in the first three blocks ending with "`]}.`": add the following to these lines and try to compile again. It should work now.

```
, {d, type_forward_declarations_are_not_allowed}
, {d, forward_or_recursive_types_are_not_allowed}
```

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 taking 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, inside the Scalaris directory, start the boot script (`boot.bat` on Windows):

```
%> ./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 but should not be used by applications to access Scalaris. See Chapter 3 on page 13 for application APIs.

In a second shell, you can now start a second Scalaris node. This will be a 'regular server':

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

The second node will read the configuration file and use this information to contact the boot server and 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:

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

In a fourth shell:

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

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

On linux you can also use the `scalarisctl` script to start boot and 'regular' nodes.

2.5.2. Running distributed

Scalaris can be installed on other machines in the same way as described in Section 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` and place `scalarisctl` into `/usr/local/bin`. But is more convenient to build an RPM and install it.

```
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 build RPMs and Debs using checkouts from svn and provide them using the openSUSE BuildService at http://download.opensuse.org/repositories/home:/tschuett/. Packages are available for

- Fedora 9, 10, 11, 12, 13,
- Mandriva 2008, 2009, 2009.1, 2010,
- openSUSE 11.0, 11.1, 11.2, 11.3, Factory,
- SLE 10, 11,
- CentOS 5.4,
- RHEL 5,
- Debian 5.0 and
- Ubuntu 9.04, 9.10.

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 10.2 on page 42, we are explaining the startup process of Scalaris nodes 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 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 often as necessary. To finish the transaction, the request list can contain a 'commit' request as the 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. Also include 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 sets up the Java environment:

```
%> ./java-api/scalaris -help
Script Options:
  --help, --h             print this message and scalaris help
  --noconfig              suppress sourcing of /etc/scalaris/scalaris-java.conf
                          and $HOME/.scalaris/scalaris-java.conf config files
  --execdebug             print scalaris exec line generated by this
                          launch script

usage: scalaris [Options]
 -b,--minibench                    run mini benchmark
 -d,--delete <key> <[timeout]>     delete an item (default timeout: 2000ms)
                                   WARNING: This function can lead to inconsistent data
                                   (e.g. deleted items can re-appear). Also when
                                   re-creating an item the version before the delete can
                                   re-appear.
 -g,--getsubscribers <topic>       get subscribers of a topic
 -h,--help                         print this message
 -lh,--localhost                   gets the local host's name as known to
                                   Java (for debugging purposes)
 -p,--publish <topic> <message>    publish a new message for the given
                                   topic
 -r,--read <key>                   read an item
 -s,--subscribe <topic> <url>      subscribe to a topic
 -u,--unsubscribe <topic> <url>    unsubscribe from a topic
 -v,--verbose                      print verbose information, e.g. the
                                   properties read
 -w,--write <key> <value>          write an item
```

`read`, `write` and `delete` can be used to read, write and delete from/to the overlay, respectively. `getsubscribers`, `publish`, and `subscribe` are the PubSub functions. The others provide debugging and testing functionality.

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

Per default, the `scalaris` script tries to connect to a boot server at `localhost`. You can change the node it connects to (and further connection properties) by adapting the values defined in `java-api/scalaris.properties`.

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 (on erlang $<$ R14) 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 3 minutes, depending on your machine. Only if the complete suite finishes, it will present statistics on failed and successful tests.

5. Troubleshooting

5.1. Network

Scalaris uses 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. General Hints

6.1. Coding Guidelines

- Keep the code short
- Use `gen_component` to implement additional processes
- Don't use receive by yourself (Exception: to implement single threaded user API calls (cs_api, yaws_calls, etc)
- Don't use `erlang:now()` , `erlang:send_after()` , `receive after` etc. in performance critical code, consider using `msg_delay` instead.
- Don't use `tc:timer()` as it catches exceptions

6.2. Testing Your Modifications and Extensions

- Run the testsuites using `make test`
- Run the java api test using `make java-test` (Scalaris output will be printed if a test fails; if you want to see it during the tests, start a `bin/boot.sh` and run the tests by `cd java; ant test`)
- Run the Ruby client by starting Scalaris and running `cd contrib; ./jsonrpc.rb`

6.3. Help with Digging into the System

- use `ets:i(t)` o get details on the local state of some processes
- consider changing pdb.erl to use ets instead of erlang:put/get
- Have a look at strace -f -p PID of beam process
- Get message statistics via the Web-interface
- enable/disable tracing for certain modules
- Use etop and look at the total memory size and atoms generated
- send processes sleep or kill messages to test certain behaviour (see gen_component.erl
- use `boot_server:number_of_nodes(). flush().`
- use `admin_checkring(). flush().`

7. System Infrastructure

7.1. The Process Dictionary

- What is it? How to distinguish from Erlangs internal process dictionary?
- Joining a process group (InstanceId is a group name)
- Why do we do this... (managing several independent nodes inside a single Erlang VM)

7.2. The Communication Layer `comm`

- in general
- format of messages (tuples)
- use messages with cookies (server and client side)
- What is a message tag?

7.3. The `gen_component`

Description is based on SVN revision r832.

The generic component model implemented by `gen_component` allows to add some common functionality to all the components that build up the Scalaris system. It supports:

event-handlers: message handling with a similar syntax as used in [2].
FIFO order of messages: components cannot be inadvertently locked as we do not use selective receive statements in the code.
sleep and halt: for testing components can sleep or be halted.
debugging, breakpoints, stepwise execution: to debug components execution can be steered via breakpoints, step-wise execution and continuation based on arriving events and user defined component state conditions.
basic profiling ,
state dependent message handlers: depending on its state, different message handlers can be used and switched during runtime. Thereby a kind of state-machine based message handling is supported.
prepared for `process_dictionary`: allows to send events to named processes inside the same group as the actual component itself (`send_to_group_member`) when just holding a reference to any group member, and
unit-testing of event-handlers: as message handling is separated from the main loop of the component, the handling of individual messages and thereby performed state manipulation can easily tested in unit-tests by directly calling message handlers.

In Scalaris all Erlang processes should be implemented as gen_component. The only exception are functions interfacing to the client, where a transition from asynchronous to synchronous request handling is necessary and that are executed in the context of a client's process or a process that behaves as a proxy for a client (cs_api).

7.3.1. A basic gen_component including a message handler

To implement a gen_component, the component has to provide the gen_component behaviour:

File gen_component.erl:

```
45 -spec behaviour_info(atom()) -> [{atom(), arity()}] | undefined.
46 behaviour_info(callbacks) ->
47     [
48      {init, 1},      % initialize component
49      {on, 2}         % handle a single message
50                      % on(Msg, State) -> NewState | unknown_event | kill
51     ];
```

This is illustrated by the following example:

File idholder.erl:

```
89 %% @doc Initialises the idholder with a random key and a counter of 0.
90 -spec init([]) -> state().
91 init(Options) ->
92     case lists:keyfind({idholder, id}, 1, Options) of
93         {{idholder, id}, Id} -> {Id, 0};
94         _ -> {get_initial_key(config:read(key_creator)), 0}
95     end.
96
97 -spec on(message(), state()) -> state().
98 on({reinit}, _State) ->
99     {get_initial_key(config:read(key_creator)), 0};
100 on({get_id, PID}, {Id, IdVersion} = State) ->
101     comm:send_local(PID, {idholder_get_id_response, Id, IdVersion}),
102     State;
103 on({set_id, NewId, NewIdVersion}, _State) ->
104     {NewId, NewIdVersion}.
```

your_gen_component:init/1 is called during start-up of a gen_component and should return the initial state to be used for this gen_component.

To react on messages / events, a message handler is used. The default message handler is called your_gen_component:on/2. This can be changed by calling gen_component:change_handler/2 (see Section 7.3.6). When an event / message for the component arrives, this handler is called with the event itself and the current state of the component. In the handler, the state of the component may be adjusted depending upon the event. The handler itself may trigger new events / messages for itself or other components and has finally to return the updated state of the component or the atoms unknown_event or kill. It must neither call receive nor timer:sleep/1 nor erlang:exit/1.

7.3.2. How to start a gen_component?

A gen_component can be started using one of:

gen_component:start(Module, Args, GenCOptions = [])
gen_component:start_link(Module, Args, GenCOptions = [])
Module: the the name of the module your component is implemented in

`Args`: List of parameters passed to `Module:init/1` for initialization
`GenCOptions`: optional parameter. List of options for `gen_component`

`{register, ProcessGroup, ProcessName}`: registers the new process with the given process group (also called instanceid) and name in the `process_dictionary`.
`{register_native, ProcessName}`: registers the process as a named Erlang process.
`wait_for_init`: wait for `Module:init/1` to return before returning to the caller.

These functions are compatible to the Erlang/OTP supervisors. They spawn a new process for the component which itself calls `Module:init/1` with the given `Args` to initialize the component. `Module:init/1` should return the initial state for your component. For each message sent to this component, the default message handler `Module:on(Message, State)` will be called, which should react on the message and return the updated state of your component.

`gen_component:start()` and `gen_component:start_link()` return the pid of the spawned process as `{ok, Pid}`.

7.3.3. When does a `gen_component` terminate?

A `gen_component` can be stopped using:

`gen_component:kill(Pid)` or by returning `kill` from the current message handler.

7.3.4. What happens when unexpected events / messages arrive?

Your message handler (default is `your_gen_component:on/2`) should return `unknown_event` in the final clause (`your_gen_component:on(_,_)`). `gen_component` then will nicely report on the unhandled message, the component's name, its state and currently active message handler, as shown in the following example:

```
# bin/boot.sh
[...]
(boot@localhost)10> process_dictionary ! {no_message}.
{no_message}
[error] unknown message: {no_message} in Module: process_dictionary and
handler on in State null
(boot@localhost)11>
```

The `process_dictionary` (see Section 7.1) is a `gen_component` which registers itself as named Erlang process with the `gen_component` option `register_native` and therefore can be addressed by its name in the Erlang shell. We send it a `{no_message}` and `gen_component` reports on the unhandled message. The `process_dictionary` itself continues to run and waits for further messages.

7.3.5. What if my message handler generates an exception or crashes the process?

`gen_component` catches exceptions generated by message handlers and reports them with a stack trace, the message, that generated the exception, and the current state of the component.

If a message handler terminates the process via `erlang:exit/1`, this is out of the responsibility scope of `gen_component`. As usual in Erlang, all linked processes will be informed. If for example `gen_component:start_link/2` or `/3` was used for starting the `gen_component`, the spawning process will be informed, which may be an Erlang supervisor process taking further actions.

7.3.6. Changing message handlers and implementing state dependent message responsiveness as a state-machine

Sometimes it is beneficial to handle messages depending on the state of a component. One possibility to express this is implementing different clauses depending on the state variable, another is introducing case clauses inside message handlers to distinguish between current states. Both approaches may become tedious, error prone, and may result in confusing source code.

Sometimes the use of several different message handlers for different states of the component leads to clearer arranged code, especially if the set of handled messages changes from state to state. For example, if we have a component with an initialization phase and a production phase afterwards, we can handle in the first message handler messages relevant during the initialization phase and simply queue all other requests for later processing using a common default clause.

When initialization is done, we handle the queued user requests and switch to the message handler for the production phase. The message handler for the initialization phase does not need to know about messages occurring during production phase and the message handler for the production phase does not need to care about messages used during initialization. Both handlers can be made independent and may be extended later on without any adjustments to the other.

One can also use this scheme to implement complex state-machines by changing the message handler from state to state.

To switch the message handler `gen_component:change_handler(State, new_handler)` is called as the last operation after a message in the active message handler was handled, so that the return value of `gen_component:change_handler/2` is propagated to `gen_component`. The new handler is given as an atom, which is the name of the 2-ary function in your component module to be called.

Starting with non-default message handler.

It is also possible to change the message handler right from the start in your `your_gen_component:init/1` to avoid the default message handler `your_gen_component:on/2`. Just create your initial state as usual and call `gen_component:change_handler(State, my_handler)` as the final call in your `your_gen_component:init/1`. We prepared `gen_component:change_handler/2` to return `State` itself, so this will work properly.

7.3.7. Halting and pausing a `gen_component`

Using `gen_component:kill(Pid)` and `gen_component:sleep(Pid, Time)` components can be terminated or paused.

7.3.8. Integration with `process_dictionary`: Redirecting events / messages to other `gen_components`

Each `gen_component` by itself is prepared to support `comm:send_to_group_member/3` which forwards messages inside a group of processes registered via the `process_dictionary` (see Section 7.1) by their name. So, if you hold a Pid of one member of a process group, you can send messages to other members of this group, if you know their registered Erlang name. You do not necessarily have to know their individual Pid.

In consequence, no gen_component can individually handle messages of the form {send_to_group_member, _, _} as such messages are consumed by gen_component itself.

7.3.9. Integration with fd_pinger: Replying to failure detectors

Each gen_component replies automatically to {ping, Pid} requests with a {pong} send to the given Pid. Such messages are generated, for example, by fd_pinger which is used by our fd failure detectors.

In consequence, no gen_component can individually handle messages of the form: {ping, _} as such messages are consumed by gen_component itself.

7.3.10. The debugging interface of gen_component: Breakpoints and step-wise execution

We equipped gen_component with a debugging interface, which especially is beneficial, when testing the interplay between several gen_components. It supports breakpoints which can pause the gen_component depending on the arriving messages or depending on user defined conditions. If a breakpoint is reached, the execution can be continued step-wise (message by message) or until the next breakpoint is reached.

We use it in our unit tests to steer protocol interleavings and to perform tests using random protocol interleavings between several processes (see paxos_SUITE). It allows also to reproduce given protocol interleavings for better testing.

Managing breakpoints.

Breakpoints are managed by the following functions:

gen_component:bp_set(Pid, MsgTag, BPName) : For the component running under Pid a breakpoint BPName is set. It is reached, when a message with a message tag MsgTag is next to be handled by the component (See comm:get_msg_tag/1 and Section 7.2 for more information on message tags). The BPName is used as a reference for this breakpoint, for example to delete it later.

gen_component:bp_set_cond(Pid, Cond, BPName) : The same as gen_component:bp_set/3 but a user defined condition implemented in {Module, Function, Params = 2}= Cond is checked by calling Module:Function(Message, State) to decide whether a breakpoint is reached or not. Message is the next message to be handled by the component and State is the current state of the component. Module:Function/2 should return a boolean.

gen_component:bp_del(Pid, BPName) : The breakpoint BPName is deleted. If the component is in this breakpoint, it will not be released by this call. This has to be done separately by gen_component:cont/1. But the deleted breakpoint will no longer be considered for newly entering a breakpoint.

gen_component:bp_barrier(Pid) : Delay all further handling of breakpoint requests until a breakpoint is actually entered.

Note, that the following call sequence may not catch the breakpoint at all, as during the sleep the component not necessarily consumes a ping message and the set breakpoint may already be deleted before a ping arrives.

```
gen_component:bp_set(Pid, ping, bp_ping),
timer:sleep(10),
gen_component:bp_del(Pid, bp_ping),
gen_component:cont(Pid).
```

This is where gen_component:bp_barrier/1 can be used:

```
gen_component:bp_set(Pid, ping, bp_ping),
gen_component:bp_barrier(Pid),
%% the following breakpoint requests will not be handled before a
%% breakpoint is reached.
%% the gen_component itself is still active and handles messages as usual
%% up to the next breakpoint
gen_component:bp_del(Pid, bp_ping),
% the breakpoint was entered once, so we delete.
% next we leave the breakpoint and continue the gen_component
gen_component:cont(Pid).
```

None of the calls in the sample listing above is blocking. It just schedules all the operations, including the bp_barrier, for the gen_component and immediately finishes. The actual events of entering and continuing the breakpoint in the gen_component can happen independently later on, when the next ping message arrives.

Managing execution.

The execution of a gen_component can be managed by the following functions:

gen_component:bp_step(Pid) : This is the only blocking breakpoint function. It waits until the gen_component is in a breakpoint and has handled a single message. It returns the module, the active message handler, and the handled message as a tuple {Module, On, Message}. This function does not actually finish the breakpoint, but just lets a single message pass through. For further messages, no breakpoint condition has to be valid, the original breakpoint is still active. To leave a breakpoint, use gen_component:bp_cont/1.

gen_component:bp_cont(Pid) : Leaves a breakpoint. gen_component runs as usual until the next breakpoint is reached.

If no further breakpoints should be entered after continuation, you should delete the registered breakpoint using gen_component:bp_del/2 before continuing the execution with gen_component:bp_cont/1. To ensure, that the breakpoint is entered at least once, gen_component:bp_barrier/1 should be used before deleting the breakpoint (see the example above). Otherwise it could happen, that the delete request arrives at your gen_component before it was actually triggered. The following continuation request would then unintentional apply to an unrelated breakpoint that may be entered later on.

gen_component:runnable(Pid) : Returns whether a gen_component has messages to handle and is runnable. If you know, that a gen_component is in a breakpoint, you can use this to check, whether a gen_component:bp_step/1 or gen_component:bp_cont/1 is applicable to the component.

Tracing handled messages – getting a message interleaving protocol.

We use the debugging interface of gen_component to test protocols with random interleaving. First we start all the components involved, set breakpoints on the initialization messages for a new

Paxos consensus and then start a single Paxos instance on all of them. The outcome of the Paxos consensus is a `learner_decide` message. So, in `paxos_SUITE:step_until_decide/3` we look for runnable processes and select randomly one of them to perform a single step until the protocol finishes with a decision.

File `paxos_SUITE.erl`:

```
223 -spec(prop_rnd_interleave/3 :: (1..4, 4..16, {pos_integer(), pos_integer(), pos_integer()})
224  -> boolean()).
225 prop_rnd_interleave(NumProposers, NumAcceptors, Seed) ->
226     ct:pal("Called with: paxos_SUITE:prop_rnd_interleave(~p, ~p, ~p).~n",
227            [NumProposers, NumAcceptors, Seed]),
228     Majority = NumAcceptors div 2 + 1,
229     {Proposers, Acceptors, Learners} =
230         make(NumProposers, NumAcceptors, 1, rnd_interleave),
231     %% set bp on all processes
232     [ gen_component:bp_set(element(3, X), proposer_initialize, bp)
233       ||  X <- Proposers],
234     [ gen_component:bp_set(element(3, X), acceptor_initialize, bp)
235       ||  X <- Acceptors ],
236     [ gen_component:bp_set(element(3, X), learner_initialize, bp)
237       || X <- Learners],
238     %% start paxos instances
239     [ proposer:start_paxosid(X, paxidrndinterl, Acceptors,
240                              proposal, Majority, NumProposers, Y)
241       || {X,Y} <- lists:zip(Proposers, lists:seq(1, NumProposers)) ],
242     [ acceptor:start_paxosid(X, paxidrndinterl, Learners)
243       || X <- Acceptors ],
244     [ learner:start_paxosid(X, paxidrndinterl, Majority,
245                             comm:this(), cpaxidrndinterl)
246       || X <- Learners],
247     %% randomly step through protocol
248     OldSeed = random:seed(Seed),
249     Steps = step_until_decide(Proposers ++ Acceptors ++ Learners, cpaxidrndinterl, 0),
250     ct:pal("Needed ~p steps~n", [Steps]),
251     case OldSeed of
252         undefined -> ok;
253         _ -> random:seed(OldSeed)
254     end,
255     true.
256
257 step_until_decide(Processes, PaxId, SumSteps) ->
258     %% io:format("Step ~p~n", [SumSteps]),
259     Runnable = [ X || X <- Processes, gen_component:runnable(element(3,X)) ],
260     case Runnable of
261         [] ->
262             ct:pal("No runnable processes of ~p~n", [length(Processes)]),
263             timer:sleep(5), step_until_decide(Processes, PaxId, SumSteps);
264         _ -> ok
265     end,
266     Num = random:uniform(length(Runnable)),
267     gen_component:bp_step(element(3,lists:nth(Num, Runnable))),
268     receive
269         {learner_decide, cpaxidrndinterl, _, _Res} = _Any ->
270             %% io:format("Received ~p~n", [_Any]),
271             SumSteps
272     after 0 -> step_until_decide(Processes, PaxId, SumSteps + 1)
273     end.
```

To get a message interleaving protocol, we either can output the results of each `gen_component:bp_step/1` call together with the Pid we selected for stepping, or alter the definition of the macro `TRACE_BP_STEPS` in `gen_component`, when we execute all `gen_components` locally in the same Erlang virtual machine.

File `gen_component.erl`:

```
30 %-define(TRACE_BP_STEPS(X,Y), io:format(X,Y)). %% output on console
31 %-define(TRACE_BP_STEPS(X,Y), ct:pal(X,Y)).    %% output even if called by unittest
```

```
32 -define(TRACE_BP_STEPS(X,Y), ok).
```

7.3.11. Future use and planned extensions for `gen_component`

`gen_component` could be further extended. For example it could support hot-code upgrade or could be used to implement algorithms that have to be run across several components of Scalaris like snapshot algorithms or similar extensions.

7.4. The Process' Database (`pdb`)

- How to use it and how to switch from erlang:put/set to ets and implied limitations.

7.5. Writing Unittests

7.5.1. Plain unittests

7.5.2. Randomized Testing using `tester.erl`

8. Basic Structured Overlay

8.1. Ring Maintenance

8.2. T-Man

8.3. Routing Tables

Description is based on SVN revision r934.

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. Scalaris provides a general interface to route a message to the (other) peer, which is currently responsible for a given `key`.

File `lookup.erl`:

```
30 -spec unreliable_lookup(Key::?RT:key(), Msg::comm:message()) -> ok.
31 unreliable_lookup(Key, Msg) ->
32     comm:send_local(process_dictionary:get_group_member(dht_node),
33                     {lookup_aux, Key, 0, Msg}).
34
35 -spec unreliable_get_key(Key::?RT:key()) -> ok.
36 unreliable_get_key(Key) ->
37     unreliable_lookup(Key, {get_key, comm:this(), Key}).
38
39 -spec unreliable_get_key(CollectorPid::comm:mypid(),
40                          ReqId::{rdht_req_id, {pos_integer(), comm:mypid()}},
41                          Key::?RT:key()) -> ok.
42 unreliable_get_key(CollectorPid, ReqId, Key) ->
43     unreliable_lookup(Key, {get_key, CollectorPid, ReqId, Key}).
```

The message `Msg` could be a `get_key` 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 `dht_node.erl`.

The message routing is implemented in `dht_node_lookup.erl`

File `dht_node_lookup.erl`:

```
27 %% @doc Find the node responsible for Key and send him the message Msg.
28 -spec lookup_aux(State::dht_node_state:state(), Key::intervals:key(),
29                  Hops::non_neg_integer(), Msg::comm:message()) -> ok.
30 lookup_aux(State, Key, Hops, Msg) ->
31     case intervals:in(Key, dht_node_state:get(State, succ_range)) of
32         true -> % found node -> terminate
33             comm:send(dht_node_state:get(State, succ_pid),
34                       {lookup_fin, Hops + 1, Msg});
35         _ ->
36             P = ?RT:next_hop(State, Key),
37             comm:send(P, {lookup_aux, Key, Hops + 1, Msg})
38     end.
```

Each node is responsible for a certain key interval. The function `intervals:in/2` is used to decide, whether the key is between the current node and its successor. If that is the case, the final step is delivers a `lookup_fin` 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/2`.

`rt_beh.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 'rt_beh'.

The macro `?RT` is used to select the current implementation of routing tables. It is defined in `include/scalaris.hrl`.

File `scalaris.hrl`:

```
25 %%The RT macro determines which kind of routingtable is used. Uncomment the
26 %%one that is desired.
27
28 %%Standard Chord routingtable
29 -define(RT, rt_chord).
30
31 %%Simple routingtable
32 %-define(RT, rt_simple).
```

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

File `rt_beh.erl`:

```
32 -spec behaviour_info(atom()) -> [{atom(), arity()}] | undefined.
33 behaviour_info(callbacks) ->
34     [
35      % create a default routing table
36      {empty, 1}, {empty_ext, 1},
37      % mapping: key space -> identifier space
38      {hash_key, 1}, {get_random_node_id, 0},
39      % routing
40      {next_hop, 2},
41      % trigger for new stabilization round
42      {init_stabilize, 3},
43      % adapt RT to changed node ID, pred and/or succ
44      {update, 7},
45      % dead nodes filtering
46      {filter_dead_node, 2},
47      % statistics
48      {to_pid_list, 1}, {get_size, 1},
49      % gets all (replicated) keys for a given (hashed) key
50      % (for symmetric replication)
51      {get_replica_keys, 1},
52      % address space size (throws 'throw:not_supported' if unsupported by the RT)
53      {n, 0},
54      % for debugging and web interface
55      {dump, 1},
56      % for bulkowner
57      {to_list, 1},
58      % convert from internal representation to version for dht_node
59      {export_rt_to_dht_node, 4},
60      % handle messages specific to a certain routing-table implementation
61      {handle_custom_message, 2},
62      % common methods, e.g. from rt_generic.hrl
63      {check, 5}, {check, 6}, {check, 7},
64      {check_config, 0}
65     ];
```

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

`empty_ext/1` similarly creates an empty external routing table for use by the `dht_node`. This process might not need all the information a routing table implementation requires and can thus work with less data.

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

`get_random_node_id/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 `dht_node`'s state (including the external routing table representation) and a key and returns the node, that should be contacted next when searching for the key, i.e. the known node nearest to the id.

`init_stabilize/3` is called periodically to rebuild the routing table. The parameters are the identifier of the node, its successor and the old (internal) routing table state. This method may send messages to the `routing_table` process which need to be handled by the `handle_custom_message/2` handler since they are implementation-specific.

`update/7` is called when the node's ID, predecessor and/or successor changes. It updates the (internal) routing table with the (new) information.

`filter_dead_node/2` is called by the failure detector and tells the routing table about dead nodes. This function gets the (internal) routing table and a node to remove from it. A new routing table state is returned.

`to_pid_list/1` get the PIDs of all (internal) routing table entries.

`get_size/1` get the (internal or external) routing table's size.

`get_replica_keys/1` Returns for a given (hashed) `Key` the (hashed) keys of its replicas. This used for implementing symmetric replication.

`n/0` gets the number of available keys. An implementation may throw `throw:not_supported` if the operation is unsupported by the routing table.

`dump/1` dump the (internal) routing table state for debugging, e.g. by using the web interface. Returns a list of `{Index, Node_as_String}` tuples which may just as well be empty.

`to_list/1` convert the (external) representation of the routing table inside a given `dht_node_state` to a sorted list of known nodes from the routing table, i.e. first=succ, second=next known node on the ring, ... This is used by bulk-operations to create a broadcast tree.

`export_rt_to_dht_node/4` convert the internal routing table state to an external state. Gets the internal state, the node's ID, the predecessor and the successor for doing so.

`handle_custom_message/2` handle messages specific to the routing table implementation. `rt_loop` will forward unknown messages to this function.

`check/5`, `check/6` are implemented in `rt_generic.hrl`, check for routing table changes and send an updated (external) routing table to the `dht_node` process.

`check_config/0` check that all required configuration parameters exist and satisfy certain restrictions.

8.3.1. The routing table process (`rt_loop`)

The `rt_loop` module implements the process for all routing tables. It processes messages and calls the appropriate methods in the specific routing table implementations.

File `rt_loop.erl`:

```
38 -opaque(state_init() :: {Id           :: ?RT:key(),
39                          Pred         :: node:node_type(),
40                          Succ         :: node:node_type(),
41                          RTState      :: ?RT:rt(),
42                          TriggerState :: trigger:state()}).
43 -type(state_uninit() :: {uninit, TriggerState :: trigger:state()}).
44 -type(state() :: state_init() | state_uninit()).
```

If initialized, the node's id, its predecessor, successor and the routing table state of the selected implementation (the macro `RT` refers to).

File `rt_loop.erl`:

```
126 on({trigger}, {Id, Pred, Succ, RTState, TriggerState}) ->
127     % start periodic stabilization
128     % log:log(info, "[ RT ] stabilize~n"),
129     NewRTState = ?RT:init_stabilize(Id, Succ, RTState),
130     ?RT:check(RTState, NewRTState, Id, Pred, Succ),
131     % trigger next stabilization
132     NewTriggerState = trigger:next(TriggerState),
133     new_state(Id, Pred, Succ, NewRTState, NewTriggerState);
```

Periodically (see `routingtable_trigger` and `pointer_base_stabilization_interval` config parameters) a `trigger` message is sent to the `rt_loop` process that starts the periodic stabilization implemented by each routing table.

File `rt_loop.erl`:

```
111 % update routing table with changed ID, pred and/or succ
112 on({update_rt, NewId, NewPred, NewSucc}, {OldId, OldPred, OldSucc, OldRT, TriggerState}) ->
113     case ?RT:update(NewId, NewPred, NewSucc, OldRT, OldId, OldPred, OldSucc) of
114         {trigger_rebuild, NewRT} ->
115             % trigger immediate rebuild
116             NewTriggerState = trigger:now(TriggerState),
117             ?RT:check(OldRT, NewRT, NewId, OldPred, NewPred, OldSucc, NewSucc),
118             new_state(NewId, NewPred, NewSucc, NewRT, NewTriggerState);
119         {ok, NewRT} ->
120             ?RT:check(OldRT, NewRT, NewId, OldPred, NewPred, OldSucc, NewSucc),
121             new_state(NewId, NewPred, NewSucc, NewRT, TriggerState)
122     end;
```

Every time a node's neighborhood changes, the `dht_node` sends an `update_rt` message to the routing table which will call `?RT:update/7` that decides whether the routing table should be re-build. If so, it will stop any waiting trigger and schedule an immideate (periodic) stabilization.

8.3.2. Common methods for routing table implementations (`rt_generic.hrl`)

This file can be included by any routing table implementation and provides a default implementation of the `check/5` and `check/6` functions.

File `rt_generic.hrl`:

```
23 %% @doc Notifies the dht_node and failure detector if the routing table changed.
24 %%      Provided for convenience (see check/6).
25 -spec check(Old::rt(), New::rt(), key(), node:node_type(),
26             node:node_type()) -> ok.
27 check(Old, New, Id, Pred, Succ) ->
28     check(Old, New, Id, Pred, Pred, Succ, Succ, true).
29
30 -spec check(Old::rt(), New::rt(), key(), node:node_type(),
31             node:node_type(), ReportToFD::boolean()) -> ok.
32 check(Old, New, Id, Pred, Succ, ReportToFD) ->
33     check(Old, New, Id, Pred, Pred, Succ, Succ, ReportToFD).
34
35 -spec check(Old::rt(), New::rt(), Id::key(),
36             OldPred::node:node_type(), NewPred::node:node_type(),
37             OldSucc::node:node_type(), NewSucc::node:node_type()) -> ok.
38 check(Old, New, Id, OldPred, NewPred, OldSucc, NewSucc) ->
39     check(Old, New, Id, OldPred, NewPred, OldSucc, NewSucc, true).
40
41 %% @doc Notifies the dht_node if the routing table changed. Also updates the
```

```
42 %%      failure detector if ReportToFD is set.
43 -spec check(Old::rt(), New::rt(), MyId::key(),
44             OldPred::node:node_type(), NewPred::node:node_type(),
45             OldSucc::node:node_type(), NewSucc::node:node_type(),
46             ReportToFD::boolean()) -> ok.
47 check(X, X, _Id, Pred, Pred, Succ, Succ, _) ->
48     ok;
49 check(OldRT, NewRT, Id, _, NewPred, _, NewSucc, true) ->
50     Pid = process_dictionary:get_group_member(dht_node),
51     comm:send_local(Pid, {rt_update, export_rt_to_dht_node(NewRT, Id, NewPred, NewSucc)}),
52     % update failure detector:
53     NewPids = to_pid_list(NewRT), OldPids = to_pid_list(OldRT),
54     fd:update_subscriptions(OldPids, NewPids);
55 check(_OldRT, NewRT, Id, _, NewPred, _, NewSucc, false) ->
56     Pid = process_dictionary:get_group_member(dht_node),
57     comm:send_local(Pid, {rt_update, export_rt_to_dht_node(NewRT, Id, NewPred, NewSucc)}).
```

Checks whether the routing table changed and in this case sends the `dht_node` an updated (external) routing table state. Optionally the failure detector is updated. This may not be necessary, e.g. if `check` is called after a crashed node has been reported by the failure detector (the failure detector already unsubscribes the node in this case).

8.3.3. Simple routing table (`rt_simple`)

One implementation of a routing table is the `rt_simple`, which routes via the successor. Note that this 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 also not very efficient in the presence of churn.

Data types

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

File `rt_simple.erl`:

```
36 % @type key(). Identifier.
37 -opaque(key()::non_neg_integer()).
38 % @type rt(). Routing Table.
39 -opaque(rt()::Succ::node:node_type()).
40 -type(external_rt()::rt()).
41 -opaque(custom_message() :: any()).
```

The routing table only consists of a node (the successor). Keys in the overlay are identified by integers ≥ 0.

A simple `rm_beh` behaviour

File `rt_simple.erl`:

```
49 %% @doc Creates an "empty" routing table containing the successor.
50 -spec empty(node:node_type()) -> rt().
51 empty(Succ) -> Succ.
```

File `rt_simple.erl`:

```
150 -spec empty_ext(node:node_type()) -> external_rt().
151 empty_ext(Succ) -> empty(Succ).
```

The empty routing table (internal or external) consists of the successor.

File `rt_simple.erl`:

```
55 %% @doc Hashes the key to the identifier space.
56 -spec hash_key(iodata() | integer()) -> key().
57 hash_key(Key) when is_integer(Key) ->
58     <<N:128>> = erlang:md5(erlang:term_to_binary(Key)),
59     N;
60 hash_key(Key) ->
61     <<N:128>> = erlang:md5(Key),
62     N.
```

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

File `rt_simple.erl`:

```
66 %% @doc Generates a random node id, i.e. a random 128-bit string.
67 -spec get_random_node_id() -> key().
68 get_random_node_id() -> hash_key(randoms:getRandomId()).
```

Random node id generation uses the helpers provided by the `randoms` module.

File `rt_simple.erl`:

```
155 %% @doc Returns the next hop to contact for a lookup.
156 -spec next_hop(dht_node_state:state(), key()) -> comm:mypid().
157 next_hop(State, _Key) -> node:pidX(dht_node_state:get(State, rt)).
```

Next hop is always the successor.

File `rt_simple.erl`:

```
76 %% @doc Triggered by a new stabilization round, renews the routing table.
77 -spec init_stabilize(key(), node:node_type(), rt()) -> rt().
78 init_stabilize(_Id, Succ, _RT) -> empty(Succ).
```

`init_stabilize/3` resets its routing table to the current successor.

File `rt_simple.erl`:

```
82 %% @doc Updates the routing table due to a changed node ID, pred and/or succ.
83 -spec update(Id::key(), Pred::node:node_type(), Succ::node:node_type(),
84               OldRT::rt(), OldId::key(), OldPred::node:node_type(),
85               OldSucc::node:node_type()) -> {ok, rt()}.
86 update(_Id, _Pred, Succ, _OldRT, _OldId, _OldPred, _OldSucc) ->
87     {ok, Succ}.
```

`update/7` updates the routing table with the new successor.

File `rt_simple.erl`:

```
91 %% @doc Removes dead nodes from the routing table (rely on periodic
92 %%      stabilization here).
93 -spec filter_dead_node(rt(), comm:mypid()) -> rt().
94 filter_dead_node(RT, _DeadPid) -> RT.
```

`filter_dead_node/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`:

```
98 %% @doc Returns the pids of the routing table entries.
99 -spec to_pid_list(rt() | external_rt()) -> [comm:mypid()].
100 to_pid_list(Succ) -> [node:pidX(Succ)].
```

`to_pid_list/1` returns the pid of the successor.

File `rt_simple.erl`:

```
104 %% @doc Returns the size of the routing table.
105 -spec get_size(rt() | external_rt()) -> 1.
106 get_size(_RT) -> 1.
```

The size of the routing table is always `1`.

File `rt_simple.erl`:

```
116 %% @doc Returns the replicas of the given key.
117 -spec get_replica_keys(key()) -> [key()].
118 get_replica_keys(Key) ->
119     [Key,
120      Key bxor 16#40000000000000000000000000000000,
121      Key bxor 16#80000000000000000000000000000000,
122      Key bxor 16#C0000000000000000000000000000000
123     ].
```

This `get_replica_keys/1` implements symmetric replication.

File `rt_simple.erl`:

```
110 %% @doc Returns the size of the address space.
111 -spec n() -> non_neg_integer().
112 n() -> 16#100000000000000000000000000000000.
```

There are 2^{128} available keys.

File `rt_simple.erl`:

```
127 %% @doc Dumps the RT state for output in the web interface.
128 -spec dump(RT::rt()) -> KeyValueList::[{Index::non_neg_integer(), Node::string()}].
129 dump(Succ) -> [{0, lists:flatten(io_lib:format("~p", [Succ]))}].
```

`dump/1` lists the successor.

File `rt_simple.erl`:

```
169 %% @doc Converts the (external) representation of the routing table to a list
170 %%      in the order of the fingers, i.e. first=succ, second=shortest finger,
171 %%      third=next longer finger,...
172 -spec to_list(dht_node_state:state()) -> nodelist:snodelist().
173 to_list(State) -> [dht_node_state:get(State, rt)].
```

`to_list/1` lists the successor from the external routing table state.

File `rt_simple.erl`:

```
161 %% @doc Converts the internal RT to the external RT used by the dht_node. Both
162 %%      are the same here.
163 -spec export_rt_to_dht_node(rt(), ID::key(), Pred::node:node_type(),
164                              Succ::node:node_type()) -> external_rt().
165 export_rt_to_dht_node(RT, _Id, _Pred, _Succ) -> RT.
```

`export_rt_to_dht_node/1` states that the external routing table is the same as the internal table.

File `rt_simple.erl`:

```
140 %% @doc There are no custom messages here.
141 -spec handle_custom_message(custom_message(), rt_loop:state_init()) -> unknown_event.
142 handle_custom_message(_Message, _State) -> unknown_event.
```

Custom messages could be send from a routing table process on one node to the routing table process on another node and are independent from any other implementation.

8.3.4. Chord routing table (rt_chord)

The file rt_chord.erl implements Chord's routing.

Data types

File rt_chord.erl:

```
40 -opaque(key()::non_neg_integer()).
41 -opaque(rt()::gb_tree()).
42 -type(external_rt()::gb_tree()).         %% @todo: make opaque
43 -type(index() :: {pos_integer(), pos_integer()}).
44 -opaque(custom_message() ::
45       {rt_get_node_response, Index::pos_integer(), Node::node:node_type()}).
```

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.

The rm_beh behaviour for Chord (excerpt)

File rt_chord.erl:

```
53 %% @doc Creates an empty routing table.
54 -spec empty(node:node_type()) -> rt().
55 empty(_Succ) -> gb_trees:empty().
```

File rt_chord.erl:

```
211 -spec empty_ext(node:node_type()) -> external_rt().
212 empty_ext(_Succ) -> gb_trees:empty().
```

empty/1 returns an empty gb_tree, same for empty_ext/1.

rt_chord:hash_key/1, rt_chord:get_random_node_id/0, rt_chord:get_replica_keys/1 and rt_chord:-n/0 are implemented like their counterparts in rt_simple.erl.

File rt_chord.erl:

```
216 %% @doc Returns the next hop to contact for a lookup.
217 %%      Note, that this code will be called from the dht_node process and
218 %%      it will thus have an external_rt!
219 -spec next_hop(dht_node_state:state(), key()) -> comm:mypid().
220 next_hop(State, Id) ->
221     RT = dht_node_state:get(State, rt),
222     case get_size(RT) =:= 0 orelse
223              intervals:in(Id, dht_node_state:get(State, succ_range)) of
224         true -> dht_node_state:get(State, succ_pid); % -> succ
225         _ -> % check routing table:
226             case util:gb_trees_largest_smaller_than(Id, RT) of
227                 {value, _Key, Node} -> node:pidX(Node);
228                 nil -> % forward to largest finger
229                     {_Key, Node} = gb_trees:largest(RT),
230                     node:pidX(Node)
231             end
232     end.
```

If the (external) routing table contains at least one item, the next hop is retrieved from the `gb_tree`. It will be the node with the largest id that is smaller than the id we are looking for. If the routing table is empty, the successor is chosen. However, if we haven't found the key in our routing table, the next hop will be our largest finger, i.e. entry.

File `rt_chord.erl`:

```
78  %% @doc Starts the stabilization routine.
79  -spec init_stabilize(key(), node:node_type(), rt()) -> rt().
80  init_stabilize(Id, _Succ, RT) ->
81      % calculate the longest finger
82      Key = calculateKey(Id, first_index()),
83      % trigger a lookup for Key
84      lookup:unreliable_lookup(Key, {rt_get_node, comm:this(), first_index()}),
85      RT.
```

The routing table stabilization is triggered for the first index 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 `dht_node`. At the destination the message is handled in `dht_node.erl`:

File `dht_node.erl`:

```
149 on({rt_get_node, Source_PID, Index}, State) ->
150     comm:send(Source_PID, {rt_get_node_response, Index, dht_node_state:get(State, node)}),
151     State;
```

The remote node sends the requested information back directly. It includes a reference to itself in a `rt_get_node_response` message which will be handled by `rt_chord:handle_custom_message/2` that calls `rt_chord:stabilize/5`:

File `rt_chord.erl`:

```
193 %% @doc Chord reacts on 'rt_get_node_response' messages in response to its
194 %%      'rt_get_node' messages.
195 -spec handle_custom_message(custom_message(), rt_loop:state_init()) -> rt_loop:state_init().
196 handle_custom_message({rt_get_node_response, Index, Node}, State) ->
197     OldRT = rt_loop:get_rt(State),
198     Id = rt_loop:get_id(State),
199     Succ = rt_loop:get_succ(State),
200     Pred = rt_loop:get_pred(State),
201     NewRT = stabilize(Id, Succ, OldRT, Index, Node),
202     check(OldRT, NewRT, Id, Pred, Succ),
203     rt_loop:set_rt(State, NewRT).
```

File `rt_chord.erl`:

```
133 %% @doc Updates one entry in the routing table and triggers the next update.
134 -spec stabilize(key(), node:node_type(), rt(), pos_integer(), node:node_type())
135         -> rt().
136 stabilize(Id, Succ, RT, Index, Node) ->
137     case node:is_valid(Node)                          % do not add null nodes
138         andalso (node:id(Succ) =/= node:id(Node))     % reached succ?
139         andalso (not intervals:in(                    % there should be nothing shorter
140                     node:id(Node),                    %   than succ
141                     node:mk_interval_between_ids(Id, node:id(Succ)))) of
142         true ->
```

```
143             NewRT = gb_trees:enter(Index, Node, RT),
144             Key = calculateKey(Id, next_index(Index)),
145             lookup:unreliable_lookup(Key, {rt_get_node, comm:this(),
146                                            next_index(Index)}),
147             NewRT;
148         _ -> RT
149     end.
```

`stabilize/5` assigns the received routing table entry and triggers the routing table stabilization for the the next shorter entry using the same mechanisms as described above.

If 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.

File `rt_chord.erl`:

```
153 %% @doc Updates the routing table due to a changed node ID, pred and/or succ.
154 -spec update(Id::key(), Pred::node:node_type(), Succ::node:node_type(),
155               OldRT::rt(), OldId::key(), OldPred::node:node_type(),
156               OldSucc::node:node_type()) -> {trigger_rebuild, rt()}.
157 update(_Id, _Pred, Succ, _OldRT, _OldId, _OldPred, _OldSucc) ->
158     % to be on the safe side ...
159     {trigger_rebuild, empty(Succ)}.
```

Tells the `rt_loop` process to rebuild the routing table starting with an empty (internal) routing table state.

File `rt_chord.erl`:

```
89 %% @doc Removes dead nodes from the routing table.
90 -spec filter_dead_node(rt(), comm:mypid()) -> rt().
91 filter_dead_node(RT, DeadPid) ->
92     DeadIndices = [Index || {Index, Node}  <- gb_trees:to_list(RT),
93                             node:same_process(Node, DeadPid)],
94     lists:foldl(fun(Index, Tree) -> gb_trees:delete(Index, Tree) end,
95                 RT, DeadIndices).
```

`filter_dead_node` removes dead entries from the `gb_tree`.

File `rt_chord.erl`:

```
236 -spec export_rt_to_dht_node(rt(), key(), node:node_type(), node:node_type())
237         -> external_rt().
238 export_rt_to_dht_node(RT, Id, Pred, Succ) ->
239     Tree = gb_trees:enter(node:id(Succ), Succ,
240                           gb_trees:enter(node:id(Pred), Pred, gb_trees:empty())),
241     util:gb_trees_foldl(fun (_K, V, Acc) ->
242                                 % only store the ring id and the according node structure
243                                 case node:id(V) =:= Id of
244                                     true  -> Acc;
245                                     false -> gb_trees:enter(node:id(V), V, Acc)
246                                 end
247                         end, Tree, RT).
```

`export_rt_to_dht_node` converts the internal `gb_tree` structure based on indices into the external representation optimised for look-ups, i.e. a `gb_tree` with node ids and the nodes themselves.

8.4. Local Datastore

8.5. Cyclon

8.6. Vivaldi Coordinates

8.7. Estimated Global Information (Gossiping)

8.8. Load Balancing

8.9. Broadcast Trees

9. Transactions in Scalaris

9.1. The Paxos Module

9.2. Transactions using Paxos Commit

9.3. Applying the Tx-Modules to replicated DHTs

Introduces transaction processing on top of a Overlay

10. How a node joins the system

10.1. General Erlang server loop

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

```
loop(State) ->
  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.

10.2. Starting additional local nodes after boot

After booting a new Scalaris-System as described in Section 2.5.1 on page 10, 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 in order to add additional nodes to the system.

The function `admin:add_nodes(int)` is defined as follows.

File `admin.erl`:

```
34 % @doc add new Scalaris nodes on the local node
```

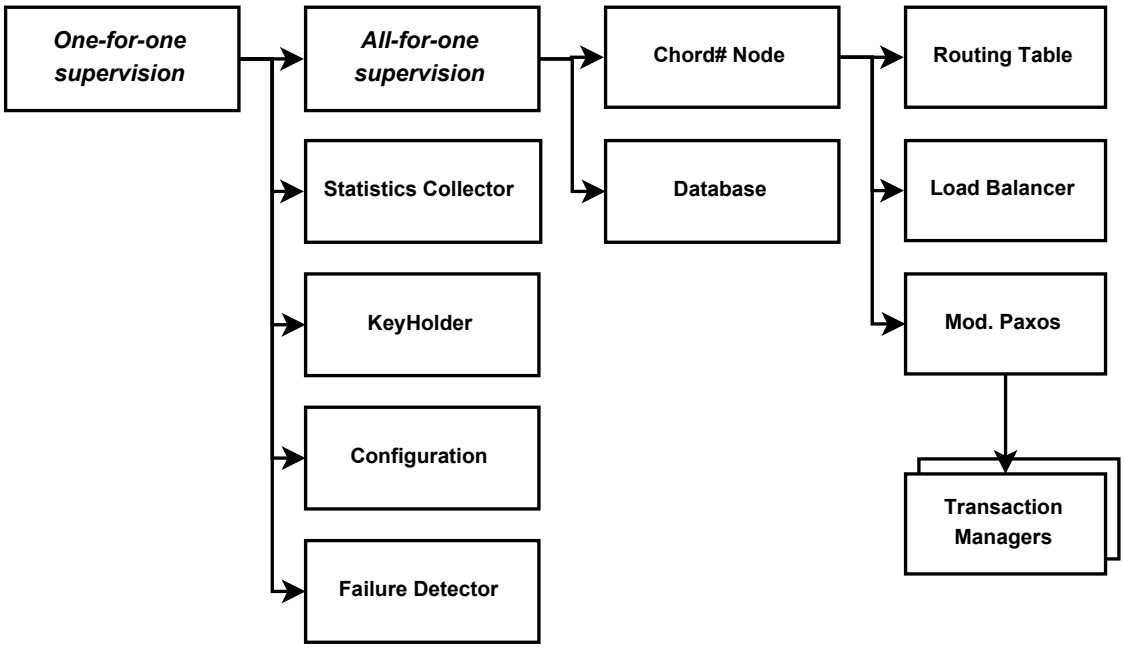
[1] Increase the log level to `info` to get the detailed startup logs. See Section 2.7 on page 12

```
35 -spec add_node_at_id(?RT:key()) -> ok.
36 add_node_at_id(Id) ->
37     add_node([{{idholder, id}, Id}]).
38
39 -spec add_node(list(tuple())) -> ok.
40 add_node(Options) ->
41     Desc = util:sup_supervisor_desc(randoms:getRandomId(),
42                                     sup_dht_node, start_link, [Options]),
43     supervisor:start_child(main_sup, Desc),
44     ok.
45
46 -spec add_nodes(non_neg_integer()) -> ok.
47 add_nodes(0) ->
48     ok;
49 add_nodes(Count) ->
50     [ begin
51           Desc = util:sup_supervisor_desc(randoms:getRandomId(),
52                                           sup_dht_node, start_link),
53           supervisor:start_child(main_sup, Desc)
54       end || _ <- lists:seq(1, Count) ],
55     ok.
```

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 `sup_dht_node:start_link(i)` s 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.

10.2.1. Supervisor-tree of a Scalaris node

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



10.2.2. 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 (`sup_dht_node:init(i)` n this case) and then calls the start function (`start_link` here.

So, lets have a look at `sup_dht_node:init(,)` the 'Scalaris *or* supervisor'.

File `sup_dht_node.erl`:

```
44 -spec init([[any()]]) -> {ok, {{one_for_one, MaxRetries::pos_integer(),
45                                  PeriodInSeconds::pos_integer()},
46                                 [ProcessDescr::any()]}}.
47 init([Options]) ->
48     InstanceId = string:concat("dht_node_", randoms:getRandomId()),
49     process_dictionary:register_process(InstanceId, sup_dht_node, self()),
50     boot_server:connect(),
51     KeyHolder =
52         util:sup_worker_desc(idholder, idholder, start_link,
53                              [InstanceId, Options]),
54     Supervisor_AND =
55         util:sup_supervisor_desc(cs_supervisor_and, sup_dht_node_core, start_link,
56                                  [InstanceId, Options]),
57     RingMaintenance =
58         util:sup_worker_desc(?RM, ?RM, start_link, [InstanceId]),
59     RoutingTable =
60         util:sup_worker_desc(routingtable, rt_loop, start_link, [InstanceId]),
61     DeadNodeCache =
62         util:sup_worker_desc(deadnodecache, dn_cache, start_link, [InstanceId]),
63     Vivaldi =
64         util:sup_worker_desc(vivaldi, vivaldi, start_link, [InstanceId]),
65     Reregister =
66         util:sup_worker_desc(dht_node_reregister, dht_node_reregister, start_link,
67                              [InstanceId]),
68     DC_Clustering =
69         util:sup_worker_desc(dc_clustering, dc_clustering, start_link,
70                              [InstanceId]),
71     Cyclon =
72         util:sup_worker_desc(cyclon, cyclon, start_link, [InstanceId]),
73     Gossip =
74         util:sup_worker_desc(gossip, gossip, start_link, [InstanceId]),
75     {ok, {{one_for_one, 10, 1},
76           [
77            Reregister,
78            KeyHolder,
79            RoutingTable,
80            Supervisor_AND,
81            Cyclon,
82            DeadNodeCache,
83            RingMaintenance,
84            Vivaldi,
85            DC_Clustering,
86            Gossip
87            %% _RSE
88           ]}}.
```

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 `sup_dht_node:init()` is finished and the supervisor module, starts all the defined processes

by calling the functions that were defined in the list of the `sup_dht_node: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.

10.2.3. 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 `sup_dht_node_core.erl`:

```
40 -spec init([instanceid() | [any()]]) -> {ok, {{one_for_all, MaxRetries::pos_integer(),
41                                                 PeriodInSeconds::pos_integer()},
42                                                [ProcessDescr::any()]}}.
43 init([InstanceId, Options]) ->
44     process_dictionary:register_process(InstanceId, sup_dht_node_core, self()),
45     Proposer =
46         util:sup_worker_desc(proposer, proposer, start_link, [InstanceId]),
47     Acceptor =
48         util:sup_worker_desc(acceptor, acceptor, start_link, [InstanceId]),
49     Learner =
50         util:sup_worker_desc(learner, learner, start_link, [InstanceId]),
51     Node =
52         util:sup_worker_desc(dht_node, dht_node, start_link,
53                              [InstanceId, Options]),
54     Delayer =
55         util:sup_worker_desc(msg_delay, msg_delay, start_link,
56                              [InstanceId]),
57     TX =
58         util:sup_supervisor_desc(sup_dht_node_core_tx, sup_dht_node_core_tx, start_link,
59                                  [InstanceId]),
60     {ok, {{one_for_all, 10, 1},
61           [
62            Proposer, Acceptor, Learner,
63            Node,
64            Delayer,
65            TX
66           ]}}.
```

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 `dht_node` uses the database, but not vice versa. The supervisor first completely initializes the DB process and afterwards calls `dht_node:start_link()` . We only go into details here, for the latter.

File `dht_node.erl`:

```
423 %% @doc spawns a scalaris node, called by the scalaris supervisor process
424 -spec start_link(instanceid()) -> {ok, pid()}.
425 start_link(InstanceId) ->
426     start_link(InstanceId, []).
427
428 -spec start_link(instanceid(), [any()]) -> {ok, pid()}.
429 start_link(InstanceId, Options) ->
430     gen_component:start_link(?MODULE, [InstanceId, Options],
431                              [{register, InstanceId, dht_node}, wait_for_init]).
```

`dht_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: dht_node).

10.2.4. Initializing a dht_node-process

File dht_node.erl:

```
403 %% @doc joins this node in the ring and calls the main loop
404 -spec init([instanceid() | [any()]]) -> {join, {as_first}, msg_queue:msg_queue()} | {join, {phase1}, 
405 init([_InstanceId, Options]) ->
406     %io:format("~p~n", [Options]),
407     % first node in this vm and also vm is marked as first
408     % or unit-test
409     case lists:member(first, Options) andalso
410           (is_unittest() orelse
411           application:get_env(boot_cs, first) =:= {ok, true}) of
412         true ->
413             trigger_known_nodes(),
414             idholder:get_id(),
415             {join, {as_first}, msg_queue:new()};
416         _ ->
417             idholder:get_id(),
418             {join, {phase1}, msg_queue:new()}
419     end.
```

The gen_component behaviour registers the dht_node in the process dictionary. Formerly, the process had to do this itself, but we moved this code into the behaviour. If the dht_node is the first node, it will start immediately by triggering all known nodes (to initialize the comm layer) and entering the join process accordingly. The node also retrieves its Id from the idholder: idholder:get_id(). In the first call, a random identifier is returned, otherwise the latest set value. If the dht_node-process failed and is restarted by its supervisor, this call to the idholder ensures, that the node still keeps its Id, assuming that the idholder 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.

If a node changes its position in the ring for load-balancing, the idholder will be informed and the dht_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 idholder and join the ring there.

TODO: 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)?

10.2.5. Actually joining the ring

After retrieving its identifier, the node starts the join protocol which processes the appropriate messages calling dht_node_join:process_join_msg(Message, State).

File dht_node_join.erl:

```
72 process_join_msg({idholder_get_id_response, Id, IdVersion},
73                  {join, {as_first}, QueuedMessages}) ->
74     log:log(info,"[ Node ~w ] joining as first: ~p",[self(), Id]),
75     Me = node:new(comm:this(), Id, IdVersion),
76     rt_loop:initialize(Id, Me, Me),
77     NewState = dht_node_state:new(?RT:empty_ext(Me),
78                                   nodelist:new_neighborhood(Me),
```

```
                                  ?DB:new(Id)),
    comm:send_local(get_local_dht_node_reregister_pid(), {go}),
    msg_queue:send(QueuedMessages),
    %log:log(info,"[ Node ~w ] joined",[self()]),
    NewState;  % join complete, State is the first "State"
```

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.

The state is defined in

File `dht_node_state.erl`:

```
-spec new(?RT:external_rt(), Neighbors::nodelist:neighborhood(),
          ?DB:db()) -> state().
new(RT, Neighbors, DB) ->
    #state{rt = RT,
           neighbors = Neighbors,
           join_time = now(),
           trans_log = #translog{tid_tm_mapping = dict:new(),
                                 decided        = gb_trees:empty(),
                                 undecided      = gb_trees:empty()
                                },
           db = DB,
           tx_tp_db = tx_tp:init(),
           proposer = process_dictionary:get_group_member(paxos_proposer)
          }.
```

If a node joins an existing ring, it will at first try to contact all `dht_node` processes in any VM configured in `known_hosts`.

File `dht_node_join.erl`:

```
% 1. get my key
process_join_msg({idholder_get_id_response, Id, IdVersion},
                 {join, {phase1}, QueuedMessages}) ->
    %io:format("p1: got key~n"),
    log:log(info,"[ Node ~w ] joining",[self()]),
    % send message to avoid code duplication
    comm:send_local(self(), {known_hosts_timeout}),
    {join, {phase2, Id, IdVersion}, QueuedMessages};

% 2. Find known hosts
process_join_msg({known_hosts_timeout},
                 {join, {phase2, _Id, _IdVersion}, _QueuedMessages} = State) ->
    %io:format("p2: known hosts timeout~n"),
    KnownHosts = config:read(known_hosts),
    % contact all known VMs
    _Res = [comm:send(KnownHost, {get_dht_nodes, comm:this()})
            || KnownHost <- KnownHosts],
    %io:format("~p~n", [_Res]),
    % timeout just in case
    comm:send_local_after(1000, self(), {known_hosts_timeout}),
    State;

process_join_msg({get_dht_nodes_response, []},
                 {join, {phase2, _Id, _IdVersion}, _QueuedMessages} = State) ->
    %io:format("p2: got empty dht_nodes_response~n"),
    % there is a VM with no nodes
    State;

process_join_msg({get_dht_nodes_response, Nodes = [_|_]},
                 {join, {phase2, Id, IdVersion}, QueuedMessages} = State) ->
    %io:format("p2: got dht_nodes_response ~p~n", [lists:delete(comm:this(), Nodes)]),
    case lists:delete(comm:this(), Nodes) of
```

```
122         [] ->
123             State;
124         [First | Rest] ->
125             comm:send(First, {lookup_aux, Id, 0, {get_node, comm:this(), Id}}),
126             comm:send_local_after(3000, self(), {lookup_timeout}),
127             {join, {phase3, Rest, Id, IdVersion}, QueuedMessages}
128     end;
```

These nodes will be send a lookup request for the node currently responsible for the new node's id – the successor for the joining node. If this lookup fails for some reason, it is tried again.

File `dht_node_join.erl`:

```
132 % 3. lookup my position
133 process_join_msg({lookup_timeout},
134                 {join, {phase3, [], Id, IdVersion}, QueuedMessages}) ->
135     %io:format("p3: lookup_timeout~n"),
136     % no more nodes left, go back to step 2
137     comm:send_local(self(), {known_hosts_timeout}),
138     {join, {phase2, Id, IdVersion}, QueuedMessages};
139
140 process_join_msg({get_node_response, Id, Succ},
141                 {join, {phase3, _DHTNodes, Id, IdVersion}, QueuedMessages}) ->
142     %io:format("p3: lookup success~n"),
143     % got my successor
144     Me = node:new(comm:this(), Id, IdVersion),
145     % announce join request
146     comm:send(node:pidX(Succ), {join, Me}),
147     {join, {phase4, Succ, Me}, QueuedMessages};
```

If its (future) successor is found, this new node will send a `join` message including a reference to itself and the chosen `Id`. This message is received by the old node in `dht_node.erl`:

File `dht_node.erl`:

```
380 on({join, NewPred}, State) ->
381     dht_node_join:join_request(State, NewPred);
```

This triggers a call to `dht_node_join:join_request/2` on the old node.

File `dht_node_join.erl`:

```
47 -spec join_request(dht_node_state:state(), NewPred::node:node_type()) ->
48         dht_node_state:state().
49 join_request(State, NewPred) ->
50     MyNewInterval =
51         node:mk_interval_between_nodes(NewPred,
52                                        dht_node_state:get(State, node)),
53     {DB, HisData} = ?DB:split_data(dht_node_state:get(State, db), MyNewInterval),
54
55     %%TODO: split data [{Key, Value, Version}], schedule transfer
56
57     comm:send(node:pidX(NewPred),
58               {join_response, dht_node_state:get(State, pred), HisData}),
59     % TODO: better already update our range here directly than waiting for an
60     % updated state from the ring_maintenance?
61     rm_beh:notify_new_pred(comm:this(), NewPred),
62     dht_node_state:set_db(State, DB).
```

The `dht_node` will update the interval it is responsible for and notify the ring maintenance of its new predecessor. It will also remove all key-value pairs from its database which are now in the responsibility of the joining node and send a `join_response` message to the new node with its former predecessor and the data the new node has to host.

File dht_node_join.erl:

```
151 % 4. joining my neighbors
152 process_join_msg({join_response, Pred, Data},
153                  {join, {phase4, Succ, Me}, QueuedMessages}) ->
154     Id = node:id(Me),
155     %io:format("p4: join_response~n"),
156     % @TODO data shouldn't be moved here, might be large
157     log:log(info, "[ Node ~w ] got pred ~w",[self(), Pred]),
158     DB = ?DB:add_data(?DB:new(Id), Data),
159     rt_loop:initialize(Id, Pred, Succ),
160     rm_beh:notify_new_succ(node:pidX(Pred), Me),
161     State = dht_node_state:new(?RT:empty_ext(Succ),
162                                nodelist:new_neighborhood(Pred, Me, Succ), DB),
163     cs_replica_stabilization:recreate_replicas(dht_node_state:get(State, my_range)),
164     comm:send_local(get_local_dht_node_reregister_pid(), {go}),
165     msg_queue:send(QueuedMessages),
166     State;
```

Back on the joining node: it waits for the `join_response` message in phase 4 of the join protocol. The next steps after the message is received from the old node are to initialize the maintenance components for the ring and routing table, the database and the state of the `dht_node`. The `cs_replica_stabilization:recreate_replicas()` function is called, which is not yet implemented. It would recreat necessary replicas that were lost due to load-balancing, node failures and lost updates during the data transfer.

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 8.3 on page 30.

Note that join-related messages arriving in other phases than those handling them will be ignored. Any other messages during a `dht_node`'s join will be queued and re-send when the join is complete.

TODO: 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!!!

11. 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 boot server'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

12. Java API

For the Java API documentation, we refer the reader to the documentation generated by javadoc or doxygen. The following commands create the documentation:

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

The documentation can then be found in `java-api/doc/index.html` (javadoc) and `java-api/doc-doxygen/html/index.html` (doxygen).

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, writing and deleting 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.

Index

?RT
    next_hop, 31
    update, 33

comm, 3, **22**, 22
    get_msg_tag, 26
    send_to_group_member, 25
cs_api, 23
cs_replica_stabilization
    recreate_replicas, 49

dht_node, 32–34, 38, 45
    start_link, 45
dht_node_join
    join_request, 48
    process_join_msg, 46

erlang
    exit, 23, 24
    now, 21
    send_after, 21
ets
    i, 21

fd, 26
fd_pinger, 3, 26

gen_component, 3, 21, **22**, 22–29
    bp_barrier, 26, 27
    bp_cont, 27
    bp_del, 26, 27
    bp_set, 26
    bp_set_cond, 26
    bp_step, 27, 28
    change_handler, 23, **25**, 25
    cont, 26
    kill, 24, 25
    runnable, 27
    sleep, 25
    start, 23, 24
    start_link, 23, 24
gen_components, 28

intervals
    in, 31

msg_delay, 21

paxos_SUITE, 26
    step_until_decide, 28
pdb, **29**
process_dictionary, 3, **22**, 22, 24, 25

randoms, 35
receive after, 21
rm_beh, 34, 37
rt_beh, **30**
    check, **32**
    check_config, **32**
    dump, **32**
    empty, **31**
    empty_ext, **31**
    export_rt_to_dht_node, **32**
    filter_dead_node, **32**
    get_random_node_id, **32**
    get_replica_keys, **32**
    get_size, **32**
    handle_custom_message, **32**
    hash_key, **32**
    init_stabilize, **32**
    n, **32**
    next_hop, **32**
    to_list, **32**
    to_pid_list, **32**
    update, **32**
rt_chord, **37**
    empty, **37**
    empty_ext, **37**
    export_rt_to_dht_node, **39**
    filter_dead_node, **39**
    get_random_node_id, 37
    get_replica_keys, 37
    handle_custom_message, **38**, 38
    hash_key, 37
    init_stabilize, **38**
    n, 37
    next_hop, **37**

stabilize, **38**, 38
update, **39**
rt_generic, **33**
rt_loop, **32**, 32, 39
rt_simple, **34**
dump, **36**
empty, **34**
empty_ext, **34**
export_rt_to_dht_node, **36**
filter_dead_node, **35**
get_random_node_id, **35**
get_replica_keys, **36**
get_size, **36**
handle_custom_message, **36**
hash_key, **35**
init_stabilize, **35**
n, **36**
next_hop, **35**
to_list, **36**
to_pid_list, **35**
update, **35**

sup_dht_node
init, 44, 45
start_link, 43

tc
timer, 21
timer
sleep, 23

your_gen_component
init, 23, 25
on, 23–25