TRANSACTIONS

Scalaris:

Users and Developers Guide

Version 0.2.0 draft January 3, 2011

Contents

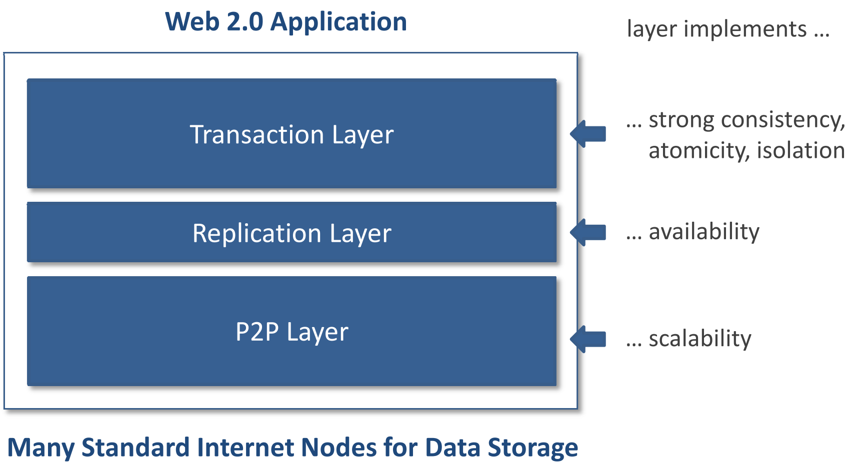
Part I.

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.

1.2. Scientific Background

Basics. The general structure of Scalaris is modelled after Chord. The Chord paper [4] describes the ring structure, the routing algorithms, and basic ring maintenance.

The main routines of our Chord node are in `src/dht_node.erl` and the join protocol is implemented in `src/dht_node_join.erl` (see also Chap. 10). Our implementation of the routing algorithms is described in more detail in Sect. 8.3 and the actual implementation is in `src/rt_chord.erl`.

Transactions. The most interesting part is probably the transaction algorithms. The most current description of the algorithms and background is in [6].

We have currently two generations of transaction algorithms. The older one is in `src/transstore`. The newer one is more modular. It provides an implementation of the paxos algorithm in `src/paxos` and the transaction algorithms itself in `src/transactions` (see also Chap. 9).

Ring Maintenance. We changed the ring maintenance algorithm in Scalaris. It is not the standard Chord one, but a variation of T-Man [5]. It is supposed to fix the ring structure faster. In some situations, the standard Chord algorithm is not able to fix the ring structure while T-Man can still fix it. For node sampling, our implementation relies on Cyclon [7].

The T-Man implementation can be found in `src/rm_tman.erl` and the Cyclon implementation in `src/cyclon`.

Vivaldi Coordinates. For some experiments, we implemented so called Vivaldi coordinates [2]. They can be used to estimate the network latency between arbitrary nodes.

The implementation can be found in `src/vivaldi.erl`.

Gossipping. For some algorithms, we use estimates of global information. These estimates are aggregated with the help of gossipping techniques [8].

The implementation can be found in `src/gossip.erl`.

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, 10.04.

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

2.7. Logging

Description is based on SVN revision r1083.

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 for both the stdout and file logger. The default value is warn; only warnings, errors and severe problems are logged.

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

In some cases, it might be necessary to get more complete logging information, e.g. for debugging. In Chapter 10 on page 43, 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}.
{log_level_file, 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/0`, `erlang:send_after/3`, `receive after` etc. in performance critical code, consider using `msg_delay` instead.
- Don't use `timer:tc/3` as it catches exceptions. Use `util:tc/3` instead.

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/0,1` to 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().`

6.4. 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.

7. System Infrastructure

7.1. Groups of Processes

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

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 r993.

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 [3].

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 `pid_groups`: 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:

```
-spec behaviour_info(atom()) -> [{atom(), arity()}] | undefined.
behaviour_info(callbacks) ->
    [
     {init, 1}     % initialize component
     % note: can use arbitrary on-handler, but by default on/2 is used:
%%      {on, 2}         % handle a single message
%%                      % on(Msg, State) -> NewState | unknown_event | kill
    ];
```

This is illustrated by the following example:

File idholder.erl:

```
%% @doc Initialises the idholder with a random key and a counter of 0.
-spec init([{{idholder, id}, Id::?RT:key()} | tuple()]) -> state().
init(Options) ->
    Id = case lists:keyfind({idholder, id}, 1, Options) of
             {{idholder, id}, IdX} -> IdX;
             _ -> ?RT:get_random_node_id()
         end,
    log:log(info, "[ idholder ~w ] init: ~p", [comm:this(), Id]),
    {Id, 0}.

-spec on(message(), state()) -> state().
on({reinit}, _State) ->
    {?RT:get_random_node_id(), 0};
on({get_id, PID}, {Id, IdVersion} = State) ->
    comm:send_local(PID, {idholder_get_id_response, Id, IdVersion}),
    State;
on({set_id, NewId, NewIdVersion}, _State) ->
    {NewId, NewIdVersion};
on({web_debug_info, Requestor}, {Id, IdVersion} = State) ->
    KeyValueList =
        [{"id", lists:flatten(io_lib:format("~p", [Id]))},
         {"id_version", lists:flatten(io_lib:format("~p", [IdVersion]))}],
    comm:send_local(Requestor, {web_debug_info_reply, KeyValueList}),
    State.
```

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. Later, the current state of the component can be retrieved using gen_component:get_state/1.

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 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

- {pid_groups_join_as, ProcessGroup, ProcessName}: registers the new process with the given process group (also called instanceid) and name using pid_groups.
- {erlang_register, 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> pid_groups ! {no_message}.
{no_message}
[error] unknown message: {no_message} in Module: pid_groups and
handler on in State null
(boot@localhost)11>
```

The pid_groups (see Section 7.1) is a gen_component which registers itself as named Erlang process with the gen_component option erlang_register 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 pid_groups module 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 `pid_groups`: 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 `pid_groups` (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. Replying to `ping` messages

Each `gen_component` replies automatically to `{ping, Pid}` requests with a `{pong}` send to the given `Pid`. Such messages are generated, for example, by `vivaldi_latency` which is used by our `vivaldi` module.

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 (bp) 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:bp_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 '`sample_bp`' may already be deleted before a ping message arrives.

```
gen_component:bp_set(Pid, ping, sample_bp),
timer:sleep(10),
gen_component:bp_del(Pid, sample_bp),
gen_component:bp_cont(Pid).
```

To overcome this, `gen_component:bp_barrier/1` can be used:

```
gen_component:bp_set(Pid, ping, sample_bp),
gen_component:bp_barrier(Pid),
%% After the bp_barrier request, following breakpoint requests
%% will not be handled before a breakpoint is actually entered.
%% The gen_component itself is still active and handles messages as usual
%% until it enters a breakpoint.
gen_component:bp_del(Pid, sample_bp),
% Delete the breakpoint after it was entered once (ensured by bp_barrier).
% Release the gen_component from the breakpoint and continue.
gen_component:bp_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 happens 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:

```erlang
-spec(prop_rnd_interleave/3 :: (1..4, 4..16, {pos_integer(), pos_integer(), pos_integer()})
 -> boolean()).
prop_rnd_interleave(NumProposers, NumAcceptors, Seed) ->
    ct:pal("Called with: paxos_SUITE:prop_rnd_interleave(~p, ~p, ~p).~n",
           [NumProposers, NumAcceptors, Seed]),
    Majority = NumAcceptors div 2 + 1,
    {Proposers, Acceptors, Learners} =
        make(NumProposers, NumAcceptors, 1, rnd_interleave),
    %% set bp on all processes
    [ gen_component:bp_set(element(3, X), proposer_initialize, bp)
      ||  X <- Proposers],
    [ gen_component:bp_set(element(3, X), acceptor_initialize, bp)
      ||  X <- Acceptors ],
    [ gen_component:bp_set(element(3, X), learner_initialize, bp)
      || X <- Learners],
    %% start paxos instances
    [ proposer:start_paxosid(X, paxidrndinterl, Acceptors,
                             proposal, Majority, NumProposers, Y)
      || {X,Y} <- lists:zip(Proposers, lists:seq(1, NumProposers)) ],
    [ acceptor:start_paxosid(X, paxidrndinterl, Learners)
      || X <- Acceptors ],
    [ learner:start_paxosid(X, paxidrndinterl, Majority,
                            comm:this(), cpaxidrndinterl)
      || X <- Learners],
    %% randomly step through protocol
    OldSeed = random:seed(Seed),
    Steps = step_until_decide(Proposers ++ Acceptors ++ Learners, cpaxidrndinterl, 0),
    ct:pal("Needed ~p steps~n", [Steps]),
    case OldSeed of
        undefined -> ok;
        _ -> random:seed(OldSeed)
    end,
    true.

step_until_decide(Processes, PaxId, SumSteps) ->
    %% io:format("Step ~p~n", [SumSteps]),
    Runnable = [ X || X <- Processes, gen_component:runnable(element(3,X)) ],
    case Runnable of
        [] ->
            ct:pal("No runnable processes of ~p~n", [length(Processes)]),
            timer:sleep(5), step_until_decide(Processes, PaxId, SumSteps);
        _ -> ok
    end,
    Num = random:uniform(length(Runnable)),
    gen_component:bp_step(element(3,lists:nth(Num, Runnable))),
    receive
        {learner_decide, cpaxidrndinterl, _, _Res} = _Any ->
            %% io:format("Received ~p~n", [_Any]),
            SumSteps
    after 0 -> step_until_decide(Processes, PaxId, SumSteps + 1)
    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:

```
%-define(TRACE_BP_STEPS(X,Y), io:format(X,Y)). %% output on console
%-define(TRACE_BP_STEPS(X,Y), ct:pal(X,Y)).    %% output even if called by unittest
-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 r1236.

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`:

```
-spec unreliable_lookup(Key::?RT:key(), Msg::comm:message()) -> ok.
unreliable_lookup(Key, Msg) ->
    comm:send_local(pid_groups:find_a(dht_node),
                    {lookup_aux, Key, 0, Msg}).

-spec unreliable_get_key(Key::?RT:key()) -> ok.
unreliable_get_key(Key) ->
    unreliable_lookup(Key, {get_key, comm:this(), Key}).

-spec unreliable_get_key(CollectorPid::comm:mypid(),
                         ReqId::{rdht_req_id, pos_integer()},
                         Key::?RT:key()) -> ok.
unreliable_get_key(CollectorPid, ReqId, Key) ->
    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`:

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

```
%%The RT macro determines which kind of routingtable is used. Uncomment the
%%one that is desired.

%%Standard Chord routingtable
-define(RT, rt_chord).
-define(MINUS_INFINITY, 0).
-define(PLUS_INFINITY, 16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF).

%%Simple routingtable
%-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`:

```
-spec behaviour_info(atom()) -> [{atom(), arity()}] | undefined.
behaviour_info(callbacks) ->
    [
     % create a default routing table
     {empty, 1}, {empty_ext, 1},
     % mapping: key space -> identifier space
     {hash_key, 1}, {get_random_node_id, 0},
     % routing
     {next_hop, 2},
     % trigger for new stabilization round
     {init_stabilize, 2},
     % adapt RT to changed neighborhood
     {update, 3},
     % dead nodes filtering
     {filter_dead_node, 2},
     % statistics
     {to_pid_list, 1}, {get_size, 1},
     % gets all (replicated) keys for a given (hashed) key
     % (for symmetric replication)
     {get_replica_keys, 1},
     % address space size, range and split key
     % (may all throw 'throw:not_supported' if unsupported by the RT)
     {n, 0}, {get_range, 2}, {get_split_key, 2},
     % for debugging and web interface
     {dump, 1},
     % for bulkowner
     {to_list, 1},
     % convert from internal representation to version for dht_node
     {export_rt_to_dht_node, 2},
     % handle messages specific to a certain routing-table implementation
     {handle_custom_message, 2},
     % common methods
     {check, 4}, {check, 5},
     {check_config, 0}
    ];
```

`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/2` 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/2` convert the internal routing table state to an external state. Gets the internal state and the node's neighborhood 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` 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`:

```
-opaque(state_active() :: {NeighbTable   :: tid(),
                           RTState       :: ?RT:rt(),
                           TriggerState  :: trigger:state()}).
```

```
-type(state_inactive() :: {inactive,
                           MessageQueue::msg_queue:msg_queue(),
                           TriggerState::trigger:state()}).
%% -type(state() :: state_active() | state_inactive()).
```

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`:

```
on_active({trigger_rt}, {NeighbTable, OldRT, TriggerState}) ->
    % start periodic stabilization
    % log:log(debug, "[ RT ] stabilize"),
    Neighbors = rm_loop:get_neighbors(NeighbTable),
    NewRT = ?RT:init_stabilize(Neighbors, OldRT),
    ?RT:check(OldRT, NewRT, Neighbors, true),
    % trigger next stabilization
    NewTriggerState = trigger:next(TriggerState),
    new_state(NeighbTable, NewRT, 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`:

```
% update routing table with changed ID, pred and/or succ
on_active({update_rt, OldNeighbors}, {NeighbTable, OldRT, TriggerState}) ->
    NewNeighbors = rm_loop:get_neighbors(NeighbTable),
    case ?RT:update(OldRT, OldNeighbors, NewNeighbors) of
        {trigger_rebuild, NewRT} ->
            % trigger immediate rebuild
            NewTriggerState = trigger:now(TriggerState),
            ?RT:check(OldRT, NewRT, OldNeighbors, NewNeighbors, true),
            new_state(NeighbTable, NewRT, NewTriggerState);
        {ok, NewRT} ->
            ?RT:check(OldRT, NewRT, OldNeighbors, NewNeighbors, true),
            new_state(NeighbTable, NewRT, TriggerState)
    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. 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`:

```
-type key_t() :: 0..16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. % 128 bit numbers
-type rt_t() :: Succ::node:node_type().
```

```
-type external_rt_t() :: Succ::node:node_type().
-type custom_message() :: none().
```

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:

```
%% @doc Creates an "empty" routing table containing the successor.
empty(Neighbors) -> nodelist:succ(Neighbors).
```

File rt_simple.erl:

```
empty_ext(Neighbors) -> empty(Neighbors).
```

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

File rt_simple.erl:

```
```

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

File rt_simple.erl:

```
%% @doc Generates a random node id, i.e. a random 128-bit number.
get_random_node_id() ->
    case config:read(key_creator) of
        random -> hash_key_(randoms:getRandomId());
        random_with_bit_mask ->
            {Mask1, Mask2} = config:read(key_creator_bitmask),
            (hash_key_(randoms:getRandomId()) band Mask2) bor Mask1
    end.
```

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

File rt_simple.erl:

```
%% @doc Returns the next hop to contact for a lookup.
next_hop(State, _Key) -> node:pidX(dht_node_state:get(State, rt)).
```

Next hop is always the successor.

File rt_simple.erl:

```
%% @doc Triggered by a new stabilization round, renews the routing table.
init_stabilize(Neighbors, _RT) -> empty(Neighbors).
```

init_stabilize/2 resets its routing table to the current successor.

File rt_simple.erl:

```
%% @doc Updates the routing table due to a changed node ID, pred and/or succ.
-spec update(OldRT::rt(), OldNeighbors::nodelist:neighborhood(),
             NewNeighbors::nodelist:neighborhood()) -> {ok, rt()}.
update(_OldRT, _OldNeighbors, NewNeighbors) ->
    {ok, nodelist:succ(NewNeighbors)}.
```

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

File rt_simple.erl:

```
%% @doc Removes dead nodes from the routing table (rely on periodic
%%      stabilization here).
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/2.

File rt_simple.erl:

```
%% @doc Returns the pids of the routing table entries.
to_pid_list(Succ) -> [node:pidX(Succ)].
```

to_pid_list/1 returns the pid of the successor.

File rt_simple.erl:

```
%% @doc Returns the size of the routing table.
get_size(_RT) -> 1.
```

The size of the routing table is always 1.

File rt_simple.erl:

```
%% @doc Returns the replicas of the given key.
get_replica_keys(Key) ->
    [Key,
     Key bxor 16#40000000000000000000000000000000,
     Key bxor 16#80000000000000000000000000000000,
     Key bxor 16#C0000000000000000000000000000000
    ].
```

This get_replica_keys/1 implements symmetric replication.

File rt_simple.erl:

```
%% @doc Returns the size of the address space.
n() -> n_().
%% @doc Helper for n/0 to make dialyzer happy with internal use of n/0.
-spec n_() -> key_t().
n_() -> 16#100000000000000000000000000000000.
```

There are 2^{128} available keys.

File rt_simple.erl:

```
%% @doc Dumps the RT state for output in the web interface.
dump(Succ) -> [{"0", lists:flatten(io_lib:format("~p", [Succ]))}].
```

dump/1 lists the successor.

File rt_simple.erl:

```
%% @doc Converts the (external) representation of the routing table to a list
%%      in the order of the fingers, i.e. first=succ, second=shortest finger,
%%      third=next longer finger,...
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:

```
%% @doc Converts the internal RT to the external RT used by the dht_node. Both
%%      are the same here.
export_rt_to_dht_node(RT, _Neighbors) -> RT.
```

export_rt_to_dht_node/2 states that the external routing table is the same as the internal table.

File rt_simple.erl:

```
%% @doc There are no custom messages here.
-spec handle_custom_message
        (custom_message() | any(), rt_loop:state_active()) -> unknown_event.
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.

File rt_simple.hrl:

```
%% @doc Notifies the dht_node and failure detector if the routing table changed.
%%      Provided for convenience (see check/5).
check(OldRT, NewRT, Neighbors, ReportToFD) ->
    check(OldRT, NewRT, Neighbors, Neighbors, ReportToFD).

%% @doc Notifies the dht_node if the (external) routing table changed.
%%      Also updates the failure detector if ReportToFD is set.
%%      Note: the external routing table only changes the internal RT has
%%      changed.
check(OldRT, NewRT, _OldNeighbors, NewNeighbors, ReportToFD) ->
    case OldRT =:= NewRT of
        true -> ok;
        _ ->
            Pid = pid_groups:get_my(dht_node),
            RT_ext = export_rt_to_dht_node(NewRT, NewNeighbors),
            comm:send_local(Pid, {rt_update, RT_ext}),
            % update failure detector:
            case ReportToFD of
                true ->
                    NewPids = to_pid_list(NewRT),
                    OldPids = to_pid_list(OldRT),
                    fd:update_subscriptions(OldPids, NewPids);
                _ -> ok
            end
    end.
```

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. Chord routing table (rt_chord)

The file rt_chord.erl implements Chord's routing.

Data types

File rt_chord.erl:

```
-type key_t() :: 0..16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. % 128 bit numbers
-type rt_t() :: gb_tree().
```

```
-type external_rt_t() :: gb_tree().
-type index() :: {pos_integer(), non_neg_integer()}.
-opaque custom_message() ::
       {rt_get_node, Source_PID::comm:mypid(), Index::index()} |
       {rt_get_node_response, Index::index(), 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`:

```
%% @doc Creates an empty routing table.
empty(_Neighbors) -> gb_trees:empty().
```

File `rt_chord.erl`:

```
empty_ext(_Neighbors) -> 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`:

```
%% @doc Returns the next hop to contact for a lookup.
%%      If the routing table has less entries than the rt_size_use_neighbors
%%      config parameter, the neighborhood is also searched in order to find a
%%      proper next hop.
%%      Note, that this code will be called from the dht_node process and
%%      it will thus have an external_rt!
next_hop(State, Id) ->
    case intervals:in(Id, dht_node_state:get(State, succ_range)) of
        true -> dht_node_state:get(State, succ_pid);
        _ ->
            % check routing table:
    RT = dht_node_state:get(State, rt),
            RTSize = get_size(RT),
            NodeRT = case util:gb_trees_largest_smaller_than(Id, RT) of
                         {value, _Key, N} ->
                             N;
                         nil when RTSize =:= 0 ->
                             dht_node_state:get(State, succ);
                         nil -> % forward to largest finger
                             {_Key, N} = gb_trees:largest(RT),
                             N
                     end,
            FinalNode =
                case RTSize < config:read(rt_size_use_neighbors) of
                    false -> NodeRT;
                    _     ->
                        % check neighborhood:
                        nodelist:largest_smaller_than(
                          dht_node_state:get(State, neighbors), Id, NodeRT)
                end,
            node:pidX(FinalNode)
    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`:

```erlang
%% @doc Starts the stabilization routine.
init_stabilize(Neighbors, RT) ->
    % calculate the longest finger
    Id = nodelist:nodeid(Neighbors),
    Key = calculateKey(Id, first_index()),
    % trigger a lookup for Key
    lookup:unreliable_lookup(Key, {send_to_group_member, routing_table,
                                   {rt_get_node, comm:this(), first_index()}}),
    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 message until the responsible node is found. There, the message is delivered to the `routing_table` process The remote node sends the requested information back directly. It includes a reference to itself in a `rt_get_node_response` message. Both messages are handled by `rt_chord:handle_custom_message/2`:

File `rt_chord.erl`:

```erlang
%% @doc Chord reacts on 'rt_get_node_response' messages in response to its
%%      'rt_get_node' messages.
-spec handle_custom_message
        (custom_message(), rt_loop:state_active()) -> rt_loop:state_active();
        (any(), rt_loop:state_active()) -> unknown_event.
handle_custom_message({rt_get_node, Source_PID, Index}, State) ->
    MyNode = nodelist:node(rt_loop:get_neighb(State)),
    comm:send(Source_PID, {rt_get_node_response, Index, MyNode}),
    State;
handle_custom_message({rt_get_node_response, Index, Node}, State) ->
    OldRT = rt_loop:get_rt(State),
    Id = rt_loop:get_id(State),
    Succ = rt_loop:get_succ(State),
    NewRT = stabilize(Id, Succ, OldRT, Index, Node),
    check(OldRT, NewRT, rt_loop:get_neighb(State), true),
    rt_loop:set_rt(State, NewRT);
handle_custom_message(_Message, _State) ->
    unknown_event.
```

File `rt_chord.erl`:

```erlang
%% @doc Updates one entry in the routing table and triggers the next update.
-spec stabilize(MyId::key() | key_t(), Succ::node:node_type(), OldRT::rt(),
                Index::index(), Node::node:node_type()) -> NewRT::rt().
stabilize(Id, Succ, RT, Index, Node) ->
    case (node:id(Succ) =/= node:id(Node))   % reached succ?
        andalso (not intervals:in(           % there should be nothing shorter
                   node:id(Node),            %   than succ
                   node:mk_interval_between_ids(Id, node:id(Succ)))) of
        true ->
            NewRT = gb_trees:enter(Index, Node, RT),
            Key = calculateKey(Id, next_index(Index)),
            Msg = {rt_get_node, comm:this(), next_index(Index)},
            lookup:unreliable_lookup(Key,
                                     {send_to_group_member, routing_table, Msg}),
            NewRT;
        _ -> RT
```

```
    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`:

```
%% @doc Updates the routing table due to a changed node ID, pred and/or succ.
-spec update(OldRT::rt(), OldNeighbors::nodelist:neighborhood(),
             NewNeighbors::nodelist:neighborhood()) -> {trigger_rebuild, rt()}.
update(_OldRT, _OldNeighbors, NewNeighbors) ->
    % to be on the safe side ...
    {trigger_rebuild, empty(NewNeighbors)}.
```

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

File `rt_chord.erl`:

```
%% @doc Removes dead nodes from the routing table.
filter_dead_node(RT, DeadPid) ->
    DeadIndices = [Index || {Index, Node}  <- gb_trees:to_list(RT),
                            node:same_process(Node, DeadPid)],
    lists:foldl(fun(Index, Tree) -> gb_trees:delete(Index, Tree) end,
                RT, DeadIndices).
```

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

File `rt_chord.erl`:

```
export_rt_to_dht_node(RT, Neighbors) ->
    Id = nodelist:nodeid(Neighbors),
    Pred = nodelist:pred(Neighbors),
    Succ = nodelist:succ(Neighbors),
    Tree = gb_trees:enter(node:id(Succ), Succ,
                          gb_trees:enter(node:id(Pred), Pred, gb_trees:empty())),
    util:gb_trees_foldl(fun (_K, V, Acc) ->
                                 % only store the ring id and the according node structure
                                 case node:id(V) =:= Id of
                                     true  -> Acc;
                                     false -> gb_trees:enter(node:id(V), V, Acc)
                                 end
                        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.

File `rt_chord.hrl`:

```
%% @doc Notifies the dht_node and failure detector if the routing table changed.
%%      Provided for convenience (see check/5).
check(OldRT, NewRT, Neighbors, ReportToFD) ->
    check(OldRT, NewRT, Neighbors, Neighbors, ReportToFD).

%% @doc Notifies the dht_node if the (external) routing table changed.
%%      Also updates the failure detector if ReportToFD is set.
%%      Note: the external routing table also changes if the Pred or Succ
%%      change.
check(OldRT, NewRT, OldNeighbors, NewNeighbors, ReportToFD) ->
    case OldRT =:= NewRT andalso
```

```erlang
             nodelist:pred(OldNeighbors) =:= nodelist:pred(NewNeighbors) andalso
             nodelist:succ(OldNeighbors) =:= nodelist:succ(NewNeighbors) of
        true -> ok;
        _ ->
            Pid = pid_groups:get_my(dht_node),
            RT_ext = export_rt_to_dht_node(NewRT, NewNeighbors),
            comm:send_local(Pid, {rt_update, RT_ext}),
            % update failure detector:
            case ReportToFD of
                true ->
                    NewPids = to_pid_list(NewRT),
                    OldPids = to_pid_list(OldRT),
                    fd:update_subscriptions(OldPids, NewPids);
                _ -> ok
            end
    end.
```

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).

8.4. 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

Description is based on SVN revision r1329.

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
[...]
(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`:

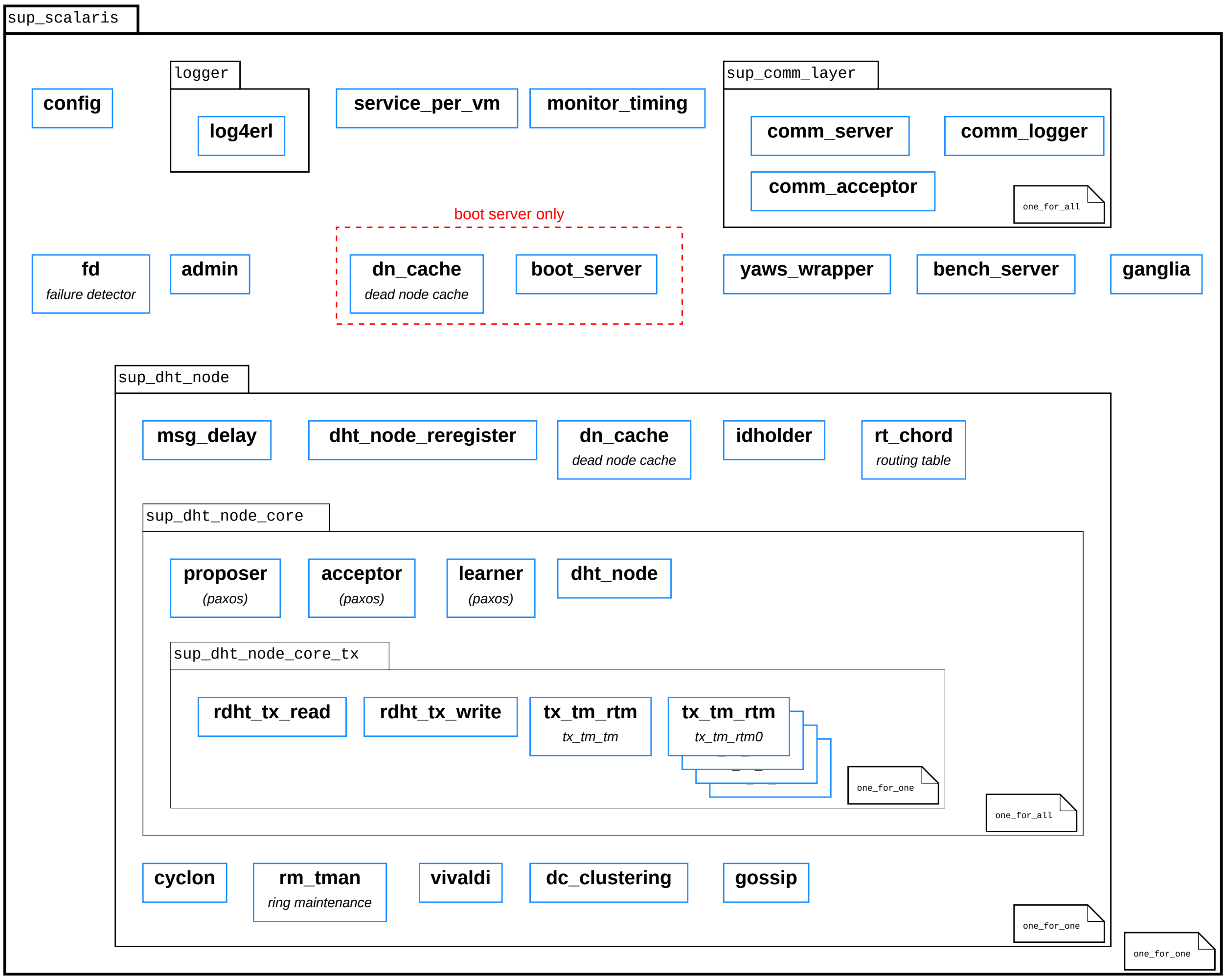
```
% @doc add new Scalaris nodes on the local node
-spec add_node_at_id(?RT:key()) ->
        ok | {error, already_present | {already_started, pid() | undefined} | term()}.
add_node_at_id(Id) ->
    add_node([{{idholder, id}, Id}, {skip_psv_lb}]).

-spec add_node([tuple()]) ->
        ok | {error, already_present | {already_started, pid() | undefined} | term()}.
add_node(Options) ->
    DhtNodeId = randoms:getRandomId(),
    Desc = util:sup_supervisor_desc(
             DhtNodeId, config:read(dht_node_sup), start_link,
             [[{my_sup_dht_node_id, DhtNodeId} | Options]]),
    case supervisor:start_child(main_sup, Desc) of
        {ok, _Child}        -> ok;
        {ok, _Child, _Info} -> ok;
        {error, _Error} = X -> X
    end.

-spec add_nodes(non_neg_integer()) ->
        nothing_to_do | [ok | {error, already_present |
                              {already_started, pid() | undefined} | term()},...].
add_nodes(0) -> nothing_to_do;
add_nodes(Count) ->
    [add_node([]) || _X <- lists:seq(1, Count)].
```

It calls `admin:add_node([])` Count times. This function starts a new child with the given options for the main supervisor `main_sup`. In particular, it sets a random ID that is passed to the new node as its suggested ID to join at. To actually perform the start, the function `sup_dht_node:start_link/1` is called by the Erlang supervisor mechanism. For more details on the OTP supervisor mechanism see Chapter 18 of the Erlang book [1] or the online documentation at `http://www.erlang.org/doc/man/supervisor.html`.

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

10.1. Supervisor-tree of a Scalaris node

When a new Erlang VM with a Scalaris node is started, a `sup_scalaris` supervisor is started that creates further workers and supervisors according to the following scheme (processes starting order: left to right, top to bottom):



When new nodes are started using `admin:add_node/1`, only new `sup_dht_node` supervisors are started.

10.2. Starting the sup_dht_node supervisor and general processes of a node

Starting supervisors is a two step process: a call to `supervisor:start_link/2,3`, e.g. from a custom supervisor's own `start_link` method, will start the supervisor process. It will then call `Module:init/1` to find out about the restart strategy, maximum restart frequency and child processes. Note that `supervisor:start_link/2,3` will not return until `Module:init/1` has returned and all child processes have been started.

Let's have a look at `sup_dht_node:init/1`, the 'DHT node supervisor'.

File `sup_dht_node.erl`:

```
-spec init([tuple()]) -> {ok, {{one_for_one, MaxRetries::pos_integer(),
                                PeriodInSeconds::pos_integer()},
                               [ProcessDescr::any()]}}.
init(Options) ->
    DHTNodeGroup = pid_groups:new("dht_node_"),
    pid_groups:join_as(DHTNodeGroup, ?MODULE),
    boot_server:connect(),

    Cyclon = util:sup_worker_desc(cyclon, cyclon, start_link, [DHTNodeGroup]),
    DC_Clustering =
        util:sup_worker_desc(dc_clustering, dc_clustering, start_link,
                             [DHTNodeGroup]),
    DeadNodeCache =
        util:sup_worker_desc(deadnodecache, dn_cache, start_link,
                             [DHTNodeGroup]),
    Delayer =
        util:sup_worker_desc(msg_delay, msg_delay, start_link,
                             [DHTNodeGroup]),
    Gossip =
        util:sup_worker_desc(gossip, gossip, start_link, [DHTNodeGroup]),
    IdHolder =
        util:sup_worker_desc(idholder, idholder, start_link,
                             [DHTNodeGroup, Options]),
    Reregister =
        util:sup_worker_desc(dht_node_reregister, dht_node_reregister,
                             start_link, [DHTNodeGroup]),
    RingMaintenance =
        util:sup_worker_desc(ring_maintenance, rm_loop, start_link, [DHTNodeGroup]),
    RoutingTable =
        util:sup_worker_desc(routing_table, rt_loop, start_link,
                             [DHTNodeGroup]),
    SupDHTNodeCore_AND =
        util:sup_supervisor_desc(sup_dht_node_core, sup_dht_node_core,
                                 start_link, [DHTNodeGroup, Options]),
    Vivaldi =
        util:sup_worker_desc(vivaldi, vivaldi, start_link, [DHTNodeGroup]),

    %% order in the following list is the start order
    {ok, {{one_for_one, 10, 1},
          [
           Delayer,
           Reregister,
           DeadNodeCache,
           IdHolder,
           RingMaintenance,
           RoutingTable,
           Cyclon,
           Vivaldi,
           DC_Clustering,
           Gossip,
           SupDHTNodeCore_AND
          ]}}.
```

The return value of the `init/1` 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: `Delayer`, `Reregister`, `DeadNodeCache`, `IdHolder`, `RoutingTable`, `SupDHTNodeCore_AND`, `Cyclon`, `RingMaintenance`, `Vivaldi`, `DC_Clustering` and a `Gossip` process in this order.

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.

When the `sup_dht_node:init/1` is finished the supervisor module starts all the defined processes by calling the functions that were defined in the returned list.

For a join of a new node, we are only interested in the starting of the `SupDHTNodeCore_AND` process here. At that point in time, all other defined processes are already started and running.

10.3. Starting the sup_dht_node_core supervisor with a peer and some paxos processes

Like any other supervisor the `sup_dht_node_core` supervisor calls its `sup_dht_node_core:init/1` function:

File `sup_dht_node_core.erl`:

```
-spec init({pid_groups:groupname(), Options::[tuple()]}) ->
                  {ok, {{one_for_all, MaxRetries::pos_integer(),
                         PeriodInSeconds::pos_integer()},
                        [ProcessDescr::any()]}}.
init({DHTNodeGroup, Options}) ->
    pid_groups:join_as(DHTNodeGroup, ?MODULE),
    Proposer =
        util:sup_worker_desc(proposer, proposer, start_link, [DHTNodeGroup]),
    Acceptor =
        util:sup_worker_desc(acceptor, acceptor, start_link, [DHTNodeGroup]),
    Learner =
        util:sup_worker_desc(learner, learner, start_link, [DHTNodeGroup]),
    DHTNode =
        util:sup_worker_desc(dht_node, dht_node, start_link,
                             [DHTNodeGroup, Options]),
    TX =
        util:sup_supervisor_desc(sup_dht_node_core_tx, sup_dht_node_core_tx, start_link,
                                 [DHTNodeGroup]),
    {ok, {{one_for_all, 10, 1},
          [
           Proposer, Acceptor, Learner,
           DHTNode,
           TX
          ]}}.
```

It defines five processes, that have to be observed using a `one_for_all`-supervisor, which means, that if one fails, all have to be restarted. The `dht_node` module implements the main component of a full Scalaris node which glues together all the other processes. Its `dht_node:start_link/2` function will get the following parameters: (a) the processes' group that is used with the `pid_groups` module and (b) a list of options for the `dht_node`. The process group name was calculated a bit earlier in the code. *Exercise: Try to find where.*

File `dht_node.erl`:

```
%% @doc spawns a scalaris node, called by the scalaris supervisor process
-spec start_link(pid_groups:groupname(), [tuple()]) -> {ok, pid()}.
start_link(DHTNodeGroup, Options) ->
    gen_component:start_link(?MODULE, Options,
                             [{pid_groups_join_as, DHTNodeGroup, dht_node}, wait_for_init]).
```

Like many other modules, the `dht_node` module implements the `gen_component` behaviour. This behaviour was developed by us to enable us to write code which is similar in syntax and semantics to the examples in [3]. Similar to the `supervisor` behaviour, a module implementing this behaviour has to provide an `init/1` 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`).

10.4. Initializing a `dht_node`-process

File dht_node.erl:

```
%% @doc joins this node in the ring and calls the main loop
-spec init(Options::[tuple()]) -> {join, {as_first | phase1, Options::[tuple()]},
                                    msg_queue:msg_queue()}.
init(Options) ->
    {my_sup_dht_node_id, MySupDhtNode} = lists:keyfind(my_sup_dht_node_id, 1, Options),
    erlang:put(my_sup_dht_node_id, MySupDhtNode),
    % first node in this vm and also vm is marked as first
    % or unit-test
    case is_first(Options) of
        true ->
            trigger_known_nodes(),
            idholder:get_id(),
            {join, {as_first, Options}, msg_queue:new()};
        _ ->
            idholder:get_id(),
            {join, {phase1, Options}, msg_queue:new()}
    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 case of a re-started node, this will be the last known ID. Otherwise, the idholder will return the ID (first) set by the {{idholder, id}, KEY} option passed to idholder:init/1 or a random key. If the dht_node-process fails and is re-started by its supervisor, this call to the idholder ensures that the node still keeps its Id, assuming that the idholder process is not failing and no passive load balancing takes place. This is important in order not to destroy an existing load balance if a node somehow fails.

10.5. 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).

10.5.1. A single node joining an empty ring

File dht_node_join.erl:

```
process_join_state({idholder_get_id_response, Id, IdVersion},
                   {join, {as_first, _Options}, QueuedMessages}) ->
    log:log(info, "[ Node ~w ] joining as first: ~p",[self(), Id]),
    Me = node:new(comm:this(), Id, IdVersion),
    % join complete, State is the first "State"
    finish_join(Me, Me, Me, ?DB:new(), QueuedMessages);
```

If the ring is empty, the joining node will be the only node in the ring and will thus be responsible for the whole key space. dht_node_join:finish_join/5 just creates a new state for a Scalaris node consisting of the given parameters (the node as itself, its predecessor and successor, an empty database and the queued messages that arrived during the join). It then activates all dependent processes and creates a routing table from this information.

The dht_node_state:state() type is defined in

File dht_node_state.erl:

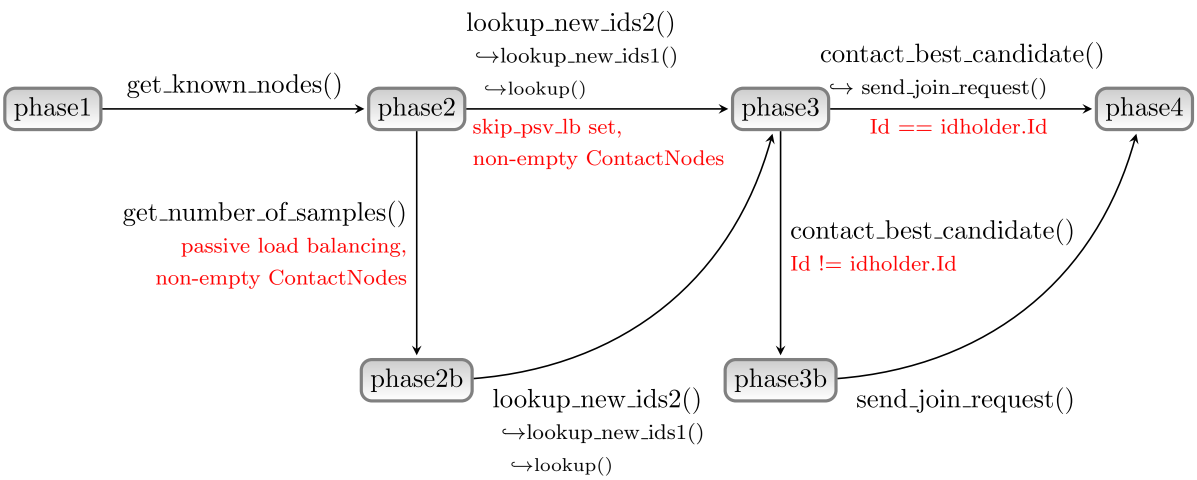
```
-record(state, {rt          = ?required(state, rt)          :: ?RT:external_rt(),
                neighbors   = ?required(state, neighbors)   :: tid(),
                join_time   = ?required(state, join_time)   :: util:time(),
                trans_log   = ?required(state, trans_log)   :: #translog{},
                db          = ?required(state, db)          :: ?DB:db(),
                tx_tp_db    = ?required(state, tx_tp_db)    :: any(),
                proposer    = ?required(state, proposer)    :: pid(),
                % slide with pred (must not overlap with 'slide with succ'!):
                slide_pred = null :: slide_op:slide_op() | null,
                % slide with succ (must not overlap with 'slide with pred'!):
                slide_succ = null :: slide_op:slide_op() | null,
                msg_fwd     = []    :: [{intervals:interval(), comm:mypid()}],
                % additional range to respond to during a move:
                db_range    = intervals:empty() :: intervals:interval()
               }).
-opaque state() :: #state{}.
```

10.5.2. A single node joining an existing (non-empty) ring

If a node joins an existing ring, its join protocol will step through the following 6 phases:

- **phase1** getting the Id from the `idholder`
- **phase2** finding nodes to contact with the help of the configured `known_hosts`
- **phase2b** getting the number of Ids to sample (may be skipped)
- **phase3** lookup nodes responsible for all sampled Ids
- **phase3b** waiting for the `idholder` to acknowledge an Id change (if applicable)
- **phase4** joining a selected node and setting up item movements

The following figure shows a (non-exhaustive) overview of the transitions between the phases in the normal case. We will go through these step by step and discuss what happens if errors occur.



Phase 1

Phase 1 is entered during the initialization of the `dht_node` process after requesting the ID from the `idholder`. It receives its `idholder_get_id_response` message and contacts all nodes set in the `known_hosts` configuration parameter. Their responses are handled in phase 2.

File dht_node_join.erl:

```
% 1. get my key
process_join_state({idholder_get_id_response, Id, IdVersion},
                   {join, {phase1, Options}, QueuedMessages}) ->
    %io:format("p1: got key~n"),
    log:log(info,"[ Node ~w ] joining",[self()]),
    get_known_nodes(),
    msg_delay:send_local(get_join_timeout() div 1000, self(), {join, timeout}),
    {join, {phase2, Options, Id, IdVersion, [], [Id], []}, QueuedMessages};
```

Phase 2 and 2b

Phase 2 collects all dht_node processes inside the contacted VMs. It therefor mainly processes get_dht_nodes_response messages and integrates all received nodes into the list of ContactNodes. The next step depends on whether the {skip_psv_lb} option for skipping any passive load balancing algorithm has been given to the dht_node or not. If it is present, the node will use the ID from the idholder, issue a lookup for the responsible node and move to phase 3. Otherwise, the passive load balancing's lb_psv_*:get_number_of_samples/1 method will be called asking for the number of IDs to sample. Its answer will be processed in phase 2b.

get_dht_nodes_response messages arriving in phase 2b or later will be processed anyway and received dht_node processes will be integrated into the ContactNodes. These phases' operations will not be interrupted and nothing else is changed.

File dht_node_join.erl:

```
% in phase 2 add the nodes and do lookups with them / get number of samples
process_join_state({get_dht_nodes_response, Nodes},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 ->
    %io:format("p2: got dht_nodes_response ~p~n", [lists:delete(comm:this(), Nodes)]),
    ContactNodes = [Node || Node <- Nodes, Node =/= comm:this()],
    NewJoinState =
        case skip_psv_lb(JoinState) of
            true -> % skip phase2b (use only the given ID)
                JoinState1 = add_contact_nodes_back(ContactNodes, JoinState),
                % (re-)issue lookups for all existing IDs
                JoinState2 = lookup(JoinState1),
                % the Id may have been removed -> select a new one in this case:
                lookup_new_ids2(1, JoinState2);
            _ -> get_number_of_samples(JoinState, ContactNodes, true)
        end,
    {join, NewJoinState, QueuedMessages};

% in all other phases, just add the provided nodes:
% note: phase1 should never receive this message!
process_join_state({get_dht_nodes_response, Nodes},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2b orelse
       element(1, JoinState) =:= phase3 orelse
       element(1, JoinState) =:= phase3b orelse
       element(1, JoinState) =:= phase4 ->
    FurtherNodes = [Node || Node <- Nodes, Node =/= comm:this()],
    {join, add_contact_nodes_back(FurtherNodes, JoinState), QueuedMessages};
```

Phase 2b will handle get_number_of_samples messages from the passive load balance algorithm. Once received, new (unique) IDs will be sampled randomly so that the total number of join candidates (selected IDs together with fully processed candidates from further phases) is at least as high as the given number of samples. Afterwards, lookups will be created for all previous IDs as well as the new ones and the node will move to phase 3.

File dht_node_join.erl:

```
% note: although this message was send in phase2, also accept message in
% phase2, e.g. messages arriving from previous calls
process_join_state({join, get_number_of_samples, Samples, Source},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 orelse
         element(1, JoinState) =:= phase2b ->
    %io:format("p2b: got get_number_of_samples ~p~n", [Samples]),
    % prefer node that send get_number_of_samples as first contact node
    JoinState1 = add_contact_nodes_front([Source], JoinState),
    % (re-)issue lookups for all existing IDs
    JoinState2 = lookup(JoinState1),
    % then create additional samples, if required
    {join, lookup_new_ids2(Samples, JoinState2), QueuedMessages};

% ignore message arriving in other phases:
process_join_state({join, get_number_of_samples, _Samples, _Source}, State) ->
    State;
```

Lookups will make Scalaris find the node currently responsible for a given ID and send a request to simulate a join to this node, i.e. a `get_candidate` message. Note that during such an operation, the joining node will become the existing node's predecessor. The simulation will be delegated to the passive load balance algorithm the joining node requested, as set by the `join_lb_psv` configuration parameter.

File dht_node_join.erl:

```
process_join_msg({join, get_candidate, Source_PID, Key, LbPsv}, State) ->
    % if anything goes wrong creating the candidate, do not crash, just report
    % a no_op operation to the other node
    LbPsv:create_join(State, Key, Source_PID);
```

Phase 3

The result of the simulation will be send in a `get_candidate_response` message and will be processed in phase 3 of the joining node. It will be integrated into the list of processed candidates. If there are no more IDs left to process, the best among them will be contacted. Otherwise further `get_candidate_response` messages will be awaited. Such messages will also be processed in the other phases where the candidate will be simply added to the list.

File dht_node_join.erl:

```
process_join_state({join, get_candidate_response, OrigJoinId, Candidate},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase3 ->
    %io:format("p3: lookup success~n"),
    JoinState1 = remove_join_id(OrigJoinId, JoinState),
    JoinState2 = integrate_candidate(Candidate, JoinState1, front),
    NewJoinState =
        case get_join_ids(JoinState2) of
            [] -> % no more join ids to look up -> join with the best:
                contact_best_candidate(JoinState2);
            [_|_] -> % still some unprocessed join ids -> wait
                JoinState2
        end,
    {join, NewJoinState, QueuedMessages};

% In phase 2 or 2b, also add the candidate but do not continue.
process_join_state({join, get_candidate_response, OrigJoinId, Candidate},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 orelse
         element(1, JoinState) =:= phase2b ->
```

```
    JoinState1 = remove_join_id(OrigJoinId, JoinState),
    JoinState2 = integrate_candidate(Candidate, JoinState1, front),
    {join, JoinState2, QueuedMessages};

% In phase 3b or 4, add the candidate to the end of the candidates as they are sorted
% and the join with the first has already started (use this candidate as backup
% if the join fails). Do not start a new join.
process_join_state({join, get_candidate_response, OrigJoinId, Candidate},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase3b orelse
       element(1, JoinState) =:= phase4 ->
    JoinState1 = remove_join_id(OrigJoinId, JoinState),
    JoinState2 = integrate_candidate(Candidate, JoinState1, back),
    {join, JoinState2, QueuedMessages};
```

If dht_node_join:contact_best_candidate/1 is called and candidates are available (there should be at this stage!), it will sort the candidates by using the passive load balance algorithm and get the ID from the best candidate. Note that during simulation at the existing node, this ID may be different from the initial ID send during the lookup. If this ID is the same as the one set at the idholder, we can continue directly by sending a join_request message and going to phase 4.

File dht_node_join.erl:

```
%% @doc Contacts the best candidate among all stored candidates and sends a
%%      join_request (Timeouts = 0).
-spec contact_best_candidate(JoinState::phase_2_4())
        -> phase2() | phase2b() | phase3b() | phase4().
contact_best_candidate(JoinState) ->
    contact_best_candidate(JoinState, 0).
%% @doc Contacts the best candidate among all stored candidates and sends a
%%      join_request. Timeouts is the number of join_request_timeout messages
%%      previously received.
-spec contact_best_candidate(JoinState::phase_2_4(), Timeouts::non_neg_integer())
        -> phase2() | phase2b() | phase3b() | phase4().
contact_best_candidate(JoinState, Timeouts) ->
    JoinState1 = sort_candidates(JoinState),
    case get_candidates(JoinState1) of
        [] -> % no candidates -> start over:
            start_over(JoinState1);
        [BestCand | _] ->
            NewId = node_details:get(lb_op:get(BestCand, n1_new), new_key),
            Id = get_id(JoinState1),
            IdVersion = get_id_version(JoinState1),
            case NewId of
                Id -> send_join_request(JoinState1, Timeouts);
                _  -> idholder:set_id(NewId, IdVersion + 1),
                      idholder:get_id(),
                      set_phase(phase3b, JoinState1)
            end
    end.
```

File dht_node_join.erl:

```
%% @doc Sends a join request to the first candidate. Timeouts is the number of
%%      join_request_timeout messages previously received.
%%      PreCond: the id has been set to the ID to join at and has been updated
%%               in JoinState.
-spec send_join_request(JoinState::phase_2_4(), Timeouts::non_neg_integer())
        -> phase2() | phase2b() | phase4().
send_join_request(JoinState, Timeouts) ->
    case get_candidates(JoinState) of
        [] -> % no candidates -> start over (should not happen):
            start_over(JoinState);
        [BestCand | _] ->
            Id = get_id(JoinState),
            IdVersion = get_id_version(JoinState),
            NewSucc = node_details:get(lb_op:get(BestCand, n1succ_new), node),
```

```
            Me = node:new(comm:this(), Id, IdVersion),
            comm:send(node:pidX(NewSucc), {join, join_request, Me}),
            msg_delay:send_local(get_join_request_timeout() div 1000,
                                 self(), {join, join_request_timeout, Timeouts}),
            set_phase(phase4, JoinState)
    end.
```

Otherwise we need to set the ID at the `idholder` first and wait for its acknowledgement in phase 3b. A `join_request` is send once the `idholder_get_id_response` message arrives.

File `dht_node_join.erl`:

```
process_join_state({idholder_get_id_response, Id, IdVersion},
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase3b->
    %io:format("p3b: got key~n"),
    JoinState1 = set_id(Id, IdVersion, JoinState),
    {join, send_join_request(JoinState1, 0), QueuedMessages};
```

The `join_request` message will be received by the existing node which will set up a slide operation with the new node or deny the request if it is not responsible for the key (anymore) and reply with a `{join, join_response, not_responsible, Node}` message.

File `dht_node_join.erl`:

```
process_join_msg({join, join_request, NewPred}, State) when (not is_atom(NewPred)) ->
    TargetId = node:id(NewPred),
    % only reply to join request with keys in our range:
    KeyInRange = dht_node_state:is_responsible(node:id(NewPred), State),
    case KeyInRange andalso
             dht_node_move:can_slide_pred(State, TargetId, {join, 'rcv'}) of
        true ->
            % TODO: implement step-wise join
            MoveFullId = util:get_global_uid(),
            SlideOp = slide_op:new_sending_slide_join(
                          MoveFullId, NewPred, join, State),
            SlideOp1 = slide_op:set_phase(SlideOp, wait_for_pred_update_join),
            RMSubscrTag = {move, slide_op:get_id(SlideOp1)},
            rm_loop:subscribe(self(), RMSubscrTag,
                              fun(_OldNeighbors, NewNeighbors) ->
                                      NewPred =:= nodelist:pred(NewNeighbors)
                              end,
                              fun dht_node_move:rm_notify_new_pred/4),
            State1 = dht_node_state:add_db_range(
                         State, slide_op:get_interval(SlideOp1)),
            send_join_response(State1, SlideOp1, NewPred);
        _ when not KeyInRange ->
            comm:send(node:pidX(NewPred), {join, join_response, not_responsible,
                                           dht_node_state:get(State, node)}),
            State;
        _ -> State
    end;
```

If it is responsible for the ID and is not participating in a slide with its current predecessor, it will set up a slide with the joining node:

File `dht_node_join.erl`:

```
-spec send_join_response(State::dht_node_state:state(),
                         NewSlideOp::slide_op:slide_op(),
                         NewPred::node:node_type())
        -> dht_node_state:state().
send_join_response(State, SlideOp, NewPred) ->
    MoveFullId = slide_op:get_id(SlideOp),
    NewSlideOp = slide_op:set_timer(SlideOp, get_join_response_timeout(),
                                    {join, join_response_timeout, NewPred, MoveFullId}),
```

```
    MyOldPred = dht_node_state:get(State, pred),
    MyNode = dht_node_state:get(State, node),
    comm:send(node:pidX(NewPred), {join, join_response, MyNode, MyOldPred, MoveFullId}),
    % no need to tell the ring maintenance -> the other node will trigger an update
    % also this is better in case the other node dies during the join
%%      rm_loop:notify_new_pred(comm:this(), NewPred),
    dht_node_state:set_slide(State, pred, NewSlideOp).
```

Phase 4

The joining node will receive the `join_response` message in phase 4 of the join protocol. If everything is ok, it will notify its ring maintenance process that it enters the ring, start all required processes and join the slide operation set up by the existing node in order to receive some of its data.

If the join candidate's node is not responsible for the candidate's ID anymore or the candidate's ID already exists, the next candidate is contacted until no further candidates are available and the join protocol starts over using `dht_node_join:start_over/1`.

Note that the `join_response` message will actually be processed in any phase. Therefore, if messages arrive late, the join can be processed immediately and the rest of the join protocol does not need to be executed again.

File `dht_node_join.erl`:

```
process_join_state({join, join_response, not_responsible, Node},
                   {join, JoinState, QueuedMessages} = State)
  when element(1, JoinState) =:= phase4 ->
    % the node we contacted is not responsible for our key (anymore)
    % -> try the next candidate, if the message is related to the current candidate
    case get_candidates(JoinState) of
        [] -> % no candidates -> should not happen in phase4!
            log:log(error, "[ Node ~w ] empty candidate list in join phase 4, "
                        "starting over", [self()]),
            {join, start_over(JoinState), QueuedMessages};
        [Candidate | _Rest] ->
            CandidateNode = node_details:get(lb_op:get(Candidate, n1succ_new), node),
            case node:same_process(CandidateNode, Node) of
                false -> State; % ignore old/unrelated message
                _ ->
                    log:log(info,
                            "[ Node ~w ] node contacted for join is not responsible for "
                            "the selected ID (anymore) ~w, trying next candidate",
                            [self(), get_id(JoinState)]),
                    JoinState1 = remove_candidate_front(JoinState),
                    {join, contact_best_candidate(JoinState1), QueuedMessages}
            end
    end;

% ignore (delayed) message arriving in other phases:
process_join_state({join, join_response, not_responsible, _Node}, State) -> State;

% note: accept (delayed) join_response messages in any phase
process_join_state({join, join_response, Succ, Pred, MoveId},
                   {join, JoinState, QueuedMessages} = State) ->
    %io:format("p4: join_response~n"),
    % only act on related messages, i.e. messages from the current candidate
    Phase = get_phase(JoinState),
    case get_candidates(JoinState) of
        [] when Phase =:= phase4 -> % no candidates -> should not happen in phase4!
            log:log(error, "[ Node ~w ] empty candidate list in join phase 4, "
                        "starting over", [self()]),
            {join, start_over(JoinState), QueuedMessages};
        [] -> State; % in all other phases, ignore the delayed join_response
                     % if no candidates exist
```

```
        [Candidate | _Rest] ->
%%            io:format("Candidate: ~.0p~n", [Candidate]),
            CandidateNode = node_details:get(lb_op:get(Candidate, n1succ_new), node),
            case node:same_process(CandidateNode, Succ) of
                false -> State; % ignore old/unrelated message
                _ ->
                    MyId = get_id(JoinState),
                    MyIdVersion = get_id_version(JoinState),
                    case MyId =:= node:id(Succ) orelse MyId =:= node:id(Pred) of
                        true ->
                            log:log(warn, "[ Node ~w ] chosen ID already exists, "
                                        "trying next candidate", [self()]),
                            JoinState1 = remove_candidate_front(JoinState),
                            {join, contact_best_candidate(JoinState1), QueuedMessages};
                        _ ->
                            Me = node:new(comm:this(), MyId, MyIdVersion),
                            log:log(info, "[ Node ~w ] joined between ~w and ~w",
                                    [self(), Pred, Succ]),
                            rm_loop:notify_new_succ(node:pidX(Pred), Me),
                            rm_loop:notify_new_pred(node:pidX(Succ), Me),

                            finish_join_and_slide(Me, Pred, Succ, ?DB:new(),
                                                  QueuedMessages, MoveId)
                    end
            end
    end;
```

File `dht_node_join.erl`:

```
-spec finish_join(Me::node:node_type(), Pred::node:node_type(),
                  Succ::node:node_type(), DB::?DB:db(),
                  QueuedMessages::msg_queue:msg_queue())
        -> dht_node_state:state().
finish_join(Me, Pred, Succ, DB, QueuedMessages) ->
    rm_loop:activate(Me, Pred, Succ),
    % wait for the ring maintenance to initialize and tell us its table ID
    NeighbTable = rm_loop:get_neighbors_table(),
    rt_loop:activate(NeighbTable),
    cyclon:activate(),
    vivaldi:activate(),
    dc_clustering:activate(),
    gossip:activate(),
    dht_node_reregister:activate(),
    msg_queue:send(QueuedMessages),
    NewRT_ext = ?RT:empty_ext(rm_loop:get_neighbors(NeighbTable)),
    dht_node_state:new(NewRT_ext, NeighbTable, DB).

-spec finish_join_and_slide(Me::node:node_type(), Pred::node:node_type(),
                  Succ::node:node_type(), DB::?DB:db(),
                  QueuedMessages::msg_queue:msg_queue(), MoveId::slide_op:id())
        -> dht_node_state:state().
finish_join_and_slide(Me, Pred, Succ, DB, QueuedMessages, MoveId) ->
    State = finish_join(Me, Pred, Succ, DB, QueuedMessages),
    SlideOp = slide_op:new_receiving_slide_join(MoveId, Pred, Succ, node:id(Me), join),
    SlideOp1 = slide_op:set_phase(SlideOp, wait_for_node_update),
    State1 = dht_node_state:set_slide(State, succ, SlideOp1),
    State2 = dht_node_state:add_msg_fwd(
               State1, slide_op:get_interval(SlideOp1),
               node:pidX(slide_op:get_node(SlideOp1))),
    RMSubscrTag = {move, slide_op:get_id(SlideOp1)},
    NewMsgQueue = msg_queue:add(QueuedMessages,
                                {move, node_update, RMSubscrTag}),
    msg_queue:send(NewMsgQueue),
    State2.
```

The macro `?RT` maps to the configured routing algorithm. It is defined in `include/scalaris.hrl`. For further details on the routing see Chapter 8.3 on page 31.

Timeouts and other errors

The following table summarizes the timeout messages send during the join protocol on the joining node. It shows in which of the phases each of the messages is processed and describes (in short) what actions are taken. All of these messages are influenced by their respective config parameters, e.g. `join_timeout` parameter in the config files defines an overall timeout for the whole join operation. If it takes longer than `join_timeout` ms, a `{join, timeout}` will be send and processed as given in this table.

<table>
<tr><th></th><th>known_hosts_timeout</th><th>get_number_of_samples_timeout</th><th>lookup_timeout</th><th>join_request_timeout</th><th>timeout</th></tr>
<tr><td>phase1</td><td>ignore</td><td>ignore</td><td>ignore</td><td>ignore</td><td>get ID from idholder</td></tr>
<tr><td>phase2</td><td>get known nodes from configured VMs</td><td>ignore</td><td>ignore</td><td>ignore</td><td rowspan="5">re-start join → phase 2 or 2b</td></tr>
<tr><td>phase2b</td><td>ignore</td><td>remove contact node, re-start join → phase 2 or 2b</td><td>ignore</td><td>ignore</td></tr>
<tr><td>phase3</td><td>ignore</td><td>ignore</td><td>remove contact node, lookup remaining IDs → phase 2 or 3</td><td>ignore</td></tr>
<tr><td>phase3b</td><td>ignore</td><td>ignore</td><td>ignore</td><td>ignore</td></tr>
<tr><td>phase4</td><td>ignore</td><td>ignore</td><td>ignore</td><td>timeouts < 3?[2]
→ contact candidate
otherwise:
remove candidate
no candidates left?
→ phase 2 or 2b
otherwise:
→ contact next one
→ phase 3b or 4</td></tr>
</table>

On the existing node, there is only one timeout message which is part of the join protocol: the `join_response_timeout`. It will be send when a slide operation is set up and if the timeout hits before the next message exchange, it will increase the slide operation's number of timeouts. The slide will be aborted if at least `join_response_timeouts` timeouts have been received. This parameter is set in the config file.

Misc. (all phases)

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.

[2]set by the `join_request_timeouts` config parameter

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 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] Frank Dabek, Russ Cox, Frans Kaahoek, Robert Morris. *Vivaldi: A Decentralized Network Coordinate System.* ACM SIGCOMM 2004.

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

[4] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160. http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[5] Mark Jelasity, Alberto Montresor, Özalp Babaoglu. *T-Man: Gossip-based fast overlay topology construction.* Computer Networks (CN) 53(13):2321-2339, 2009.

[6] F. Schintke, A. Reinefeld, S. Haridi, T. Schütt. *Enhanced Paxos Commit for Transactions on DHTs.* 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing, pp. 448-454, May 2010.

[7] Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. *CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.* J. Network Syst. Manage. 13(2): 2005.

[8] Márk Jelasity, Alberto Montresor, Ozalp Babaoglu. *Gossip-based aggregation in large dynamic networks.* ACM Trans. Comput. Syst. 23(3), 219-252 (2005).

Index