TRANSACTIONS

Scalaris:

Users and Developers Guide

Version 0.2.0 draft June 16, 2010

Contents

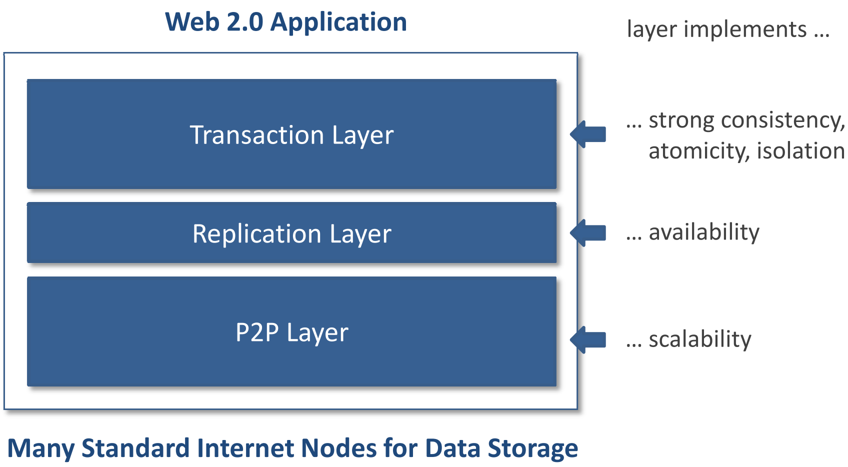
Part I

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

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

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

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

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

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

2 Download and Installation

2.1 Requirements

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

- Erlang R12 or newer
- GNU-like Make

Note, the Version 13 of Erlang is required. Scalaris will not work with older versions.

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

- Java Development Kit 1.6
- Apache Ant

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

2.2 Download

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

2.2.1 Development Branch

You find the latest development version in the svn repository:

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

2.2.2 Releases

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

2.3 Configuration

Scalaris reads two configuration files from the working directory: `bin/scalaris.cfg` (mandatory) and `bin/scalaris.local.cfg` (optional). The former defines default settings and is included in the release. The latter can be created by the user to alter settings. A sample file is `bin/scalaris.local.cfg.example`. To run 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 distinguishes currently two different kinds of nodes: (a) the boot-server and (b) regular nodes. For the moment, we limit the number of boot-servers to exactly one. The remaining nodes are regular nodes. The boot-server is contacted to join the system. On all servers, the `boot_host` option defines the server where the boot server is running. In the example, it is an IP address plus a TCP port.

2.4 Build

2.4.1 Linux

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

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

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

2.4.2 Windows

We are currently not supporting Scalaris on Windows. However, we have two small `bat` files for building and running a boot server. It seems to work but we make no guarantees.

For the most recent description please see the FAQ at http://code.google.com/p/scalaris/wiki/FAQ.

2.4.3 Java-API

The following commands will build the Java API for Scalaris:

```
%> make java
```

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

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

2.5 Running Scalaris

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

- boot servers
- regular nodes

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

2.5.1 Running on a local machine

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

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

This will start the boot server. On success `http://localhost:8000` should point to the management interface page of the boot server. The main page will show you the number of nodes currently in the system. After a couple of seconds a first Scalaris should have started in the boot server and the number should increase to one. The main page will also allow you to store and retrieve key-value pairs.

In a second shell, you can now start a second Scalaris node. This will be a 'regular server'. Go in the bin directory:

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

The second node will read the configuration file and use this information to contact the boot server and will join the ring. The number of nodes on the web page should have increased to two by now.

Optionally, a third and fourth node can be started on the same machine. In a third shell:

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

In a fourth shell:

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

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

2.5.2 Running distributed

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

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

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

2.6 Installation

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

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

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

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

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

2.7 Logging

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

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

In some cases, it might be necessary to get more complete logging information, e.g. for debugging. In 10.2 on page 36, we are explaining the startup process of Scalaris nodes in more detail, here the `info` level provides more detailed information.

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

3 Using the system

3.1 JSON API

Scalaris supports a JSON API for transactions. To minimize the necessary round trips between a client and Scalaris, it uses request lists, which contain all requests that can be done in parallel. The request list is then send over to a Scalaris node with a POST message. The result is an opaque TransLog and a list containing the results of the requests. To add further requests to the transaction, the TransLog and another list of requests may be send to Scalaris. This process may be repeated as necessary. To finish the transaction, the request list can contain a 'commit' request as last element, which triggers the validation phase of the transaction processing.

The JSON-API can be accessed via the Scalaris-Web-Server running on port 8000 by default and the page `jsonrpc.yaws` (For example at: http://localhost:8000/jsonrpc.yaws). The following example illustrates the message flow:

Client		**Scalaris node**
Make a transaction, that sets two keys:	→	

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

← Scalaris sends results back


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


In a second transaction: Read the two keys →


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


← Scalaris sends results back


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

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

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

← Scalaris sends results back

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

A sample usage of the JSON API using Ruby can be found in `contrib/jsonrpc.rb`.

A single request list must not contain a key more than once!

The allowed requests are:

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

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

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

The possible results are:

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

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

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

3.1.1 Deleting a key

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

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

Two sample results

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

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

3.2 Java command line interface

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

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

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

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

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

3.3 Java API

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

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

For details we refer the reader to the Javadoc:

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

4 Testing the system

4.1 Running the unit tests

There are some unit tests in the `test` directory. You can call them by running `make test` in the main directory. The results are stored in a local `index.html` file.

The tests are implemented with the `common-test` package from the Erlang system. For running the tests we rely on `run_test`, which is part of the `common-test` package, but is not installed by default. `configure` will check whether `run_test` is available. If it is not installed, it will show a warning and a short description of how to install the missing file.

Note: for the unit tests, we are setting up and shutting down several overlay networks. During the shut down phase, the runtime environment will print extensive error messages. These error messages do not indicate that tests failed! Running the complete test suite takes about 5 minutes. Only when the complete suite finished, it will present statistics on failed and successful tests.

5 Troubleshooting

5.1 Network

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

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

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

Part II

Developers Guide

6 General Hints

6.1 Coding Guidelines

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

6.2 Testing Your Modifications and Extensions

- Run the testsuites using make test
- Run the java api test using make java-test (or if you want to see the scalaris output 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() 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().`

7 System Infrastructure

7.1 The Process Dictionary

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

7.2 The Communication Layer `comm`

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

7.3 The `gen_component`

Description is based on SVN revision r832.

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

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

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

7.3.1 A basic gen_component including a message handler

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

File gen_component.erl:

```
behaviour_info(callbacks) ->
    [
     {init, 1},     % initialize component
     {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([]) -> state().
init(_Arg) ->
    {get_initial_key(config:read(key_creator)), 0}.

-spec on(message(), state()) -> state() | unknown_event.
on({reinit}, _State) ->
    {get_initial_key(config:read(key_creator)), 0};
on({get_id, PID}, {Key, Count} = State) ->
    comm:send_local(PID, {idholder_get_id_response, Key, Count}),
    State;
on({set_id, NewKey, Count}, _State) ->
    {NewKey, Count};
on(_, _State) ->
    unknown_event.
```

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

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

7.3.2 How to start a gen_component?

A gen_component can be started using one of:

gen_component:start(Module, Args, GenCOptions = [])
gen_component:start_link(Module, Args, GenCOptions = [])
  Module: the the name of the module your component is implemented in
  Args: List of parameters passed to Module:init/1 for initialization
  GenCOptions: optional parameter. List of options for gen_component

{register, ProcessGroup, ProcessName}: registers the new process with the given process group (also called instanceid) and name in the process_dictionary.

{register_native, ProcessName}: registers the process as a named Erlang process.

wait_for_init: wait for Module:init/1 to return before returning to the caller.

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

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

7.3.3 When does a gen_component terminate?

A gen_component can be stopped using:

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

7.3.4 What happens when unexpected events / messages arrive?

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3.7 Halting and pausing a `gen_component`

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

7.3.8 Integration with `process_dictionary`: Redirecting events / messages to other `gen_component`s

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

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

7.3.9 Integration with `fd_pinger`: Replying to failure detectors

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

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

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

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

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

Managing breakpoints.

Breakpoints are managed by the following functions:

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

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

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

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

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

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

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

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

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

Managing execution.

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

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

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

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

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

Tracing handled messages – getting a message interleaving protocol.

We use the debugging interface of gen_component to test protocols with random interleaving. First we start all the components involved, set breakpoints on the initialization messages for a new Paxos consensus and then start a single Paxos instance on all of them. The outcome of the Paxos consensus is a learner_decide message. So, in paxos_SUITE:step_until_decide/3 we look for runnable processes and select randomly one of them to perform a single step until the protocol finishes with a decision.

File paxos_SUITE.erl:

```
-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

Each node of the ring can perform searches in the overlay.

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

File `cs_lookup.erl`:

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

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

The message `Msg` could be a `get` which retrieves content from the responsible node or a `get_node` message, which returns a pointer to the node.

All currently supported messages are listed in the file `dht_node.erl`.

The message routing is implemented in lookup.erl

File `lookup.erl`:

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

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

Each node is responsible for a certain key interval. The function `util:is_between` is used to decide, whether the key is between the current node and its successor. If that is the case, final step is done using `lookup_fin()`, which delivers the message to the local node. Otherwise, the message is forwarded to the next nearest known peer (listed in the routing table) determined by `?RT:next_hop`.

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

File `rt_beh.erl`:

```
```

- `empty/1` gets a successor passed and generates an empty routing table. The data structure of the routing table is undefined. It can be a list, a tree, a matrix . . .
- `hash_key/1` gets a key and maps it into the overlay's identifier space.
- `getRandomNodeId/0` returns a random node id from the overlay's identifier space. This is used for example when a new node joins the system.
- `next_hop/2` gets a routing table and a key and returns the node, that should be contacted next (is nearest to the id).
- `init_stabilize/3` is called periodically to rebuild the routing table. The parameters are the identifier of the node, the successor and the old routing table state.
- `filterDeadNode/2` is called by the failuredetector and tells the routing table about dead nodes to be eliminated from the routing table. This function cleans the routing table.
- `to_pid_list/1` get all PIDs of the routing table entries.
- `get_size/1` get the routing table's size.
- `get_keys_for_replicas/1` Returns for a given `Key` the keys of its replicas. This used for implementing symmetric replication.
- `dump/1` dump the state. Not mandatory, may just return `ok`.
- `to_dict/1` returns the routing tables entries in an array-like structure. This is used by bulk-operations to create a broadcast tree.

8.3.1 Simple routing table

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

Data types

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

File `rt_simple.erl`:

```
% @type key(). Identifier.
-type(key()::non_neg_integer()).
% @type rt(). Routing Table.
```

```
-type(rt()::Succ::node:node_type()).
-type(external_rt()::rt()).
-type(custom_message() :: any()).
```

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

A simple routingtable behaviour

File `rt_simple.erl`:

```
%% @doc Creates an empty routing table.
%%      Per default the empty routing should already include the successor.
-spec empty(node:node_type()) -> rt().
empty(Succ) ->
    Succ.
```

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

File `rt_simple.erl`:

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

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

File `rt_simple.erl`:

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

Next hop is always the successor.

File `rt_simple.erl`:

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

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

File `rt_simple.erl`:

```
%% @doc Removes dead nodes from the routing table.
-spec filterDeadNode(rt(), comm:mypid()) -> rt().
filterDeadNode(RT, _DeadPid) ->
    RT.
```

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

File rt_simple.erl:

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

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

File rt_simple.erl:

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

n() ->
    16#100000000000000000000000000000000.

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

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

File rt_simple.erl:

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

dump/1 is not implemented.

8.3.2 Chord routing table

The file rt_chord.erl implements Chord's routing.

Data types

File rt_chord.erl:

```
-type(key()::non_neg_integer()).
-type(rt()::gb_tree()).
-type(external_rt()::gb_tree()).
-type(index() :: {pos_integer(), pos_integer()}).
-type(custom_message() ::
       {rt_get_node_response, Index::pos_integer(), Node::node:node_type()}).
```

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

The routingtable behaviour for Chord

File rt_chord.erl:

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

empty/1 returns an empty gb_tree.

hash_key(Key) and getRandomNodeId call their counterparts from rt_simple.erl

File rt_chord.erl:

```
%% @doc Returns the next hop to contact for a lookup.
%%      Note, that this code will be called from the dht_node process and
%%      it will thus have an external_rt!
-spec next_hop(dht_node_state:state(), key()) -> comm:mypid().
next_hop(State, Id) ->
    case intervals:in(Id, dht_node_state:get(State, succ_range)) of
        %succ is responsible for the key
        true ->
            dht_node_state:get(State, succ_pid);
        % check routing table
        false ->
            case util:gb_trees_largest_smaller_than(Id, dht_node_state:get(State, rt)) of
                nil ->
                    dht_node_state:get(State, succ_pid);
                {value, _Key, Node} ->
                    node:pidX(Node)
            end
    end.
```

If the entry exists, it is retrieved from the gb_tree. If the id of the routing table entry is between ourselves and the searched id, the finger is chosen. If anything fails, the successor is chosen.

File rt_chord.erl:

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

The routing table stabilization is triggered with the index 127 and then runs asynchronously, as we do not want to block the rt_loop to perform other request while recalculating the routing table.

We have to find the node responsible for the calculated finger and therefore perform a lookup for the node with a rt_get_node message, including a reference to ourselves as the reply-to address and the index to be set.

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

File dht_node.erl:

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

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

File rt_chord.erl:

```
%% @doc Updates one entry in the routing table and triggers the next update.
-spec stabilize(key(), node:node_type(), rt(), pos_integer(), node:node_type())
      -> rt().
stabilize(Id, Succ, RT, Index, Node) ->
    case node:is_valid(Node)                              % do not add null nodes
        andalso (node:id(Succ) =/= node:id(Node))       % there is nothing shorter than succ
        andalso (not intervals:in(node:id(Node), intervals:mk_from_node_ids(Id, node:id(Succ)))) of %
        true ->
            NewRT = gb_trees:enter(Index, Node, RT),
            Key = calculateKey(Id, next_index(Index)),
            lookup:unreliable_lookup(Key, {rt_get_node, comm:this(),
                                          next_index(Index)}),
            NewRT;
        false ->
            RT
    end.
```

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

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

filterDeadNode removes dead entries from the gb_tree.

File rt_chord.erl:

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

8.4 Local Datastore

8.5 Cyclon

8.6 Vivaldi Coordinates

9 Transactions in Scalaris

9.1 The Paxos Module

9.2 Transactions using Paxos Commit

9.3 Applying the Tx-Modules to replicated DHTs

Introduces transaction processing on top of a Overlay

10 How a node joins the system

10.1 General Erlang server loop

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

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

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

10.2 Starting additional local nodes after boot

After booting a new Scalaris-System as described in Section 2.5.1 on page 9, ten additional local nodes can be started by typing `admin:add_nodes(10)` in the Erlang-Shell that the boot process opened [1].

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

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

In the following we will trace, what this function does to join additional nodes to the system.

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

File `admin.erl`:

```
%%--------------------------------------------------------------------
%% Function: add_nodes(int()) -> ok
%% Description: add new Scalaris nodes
```

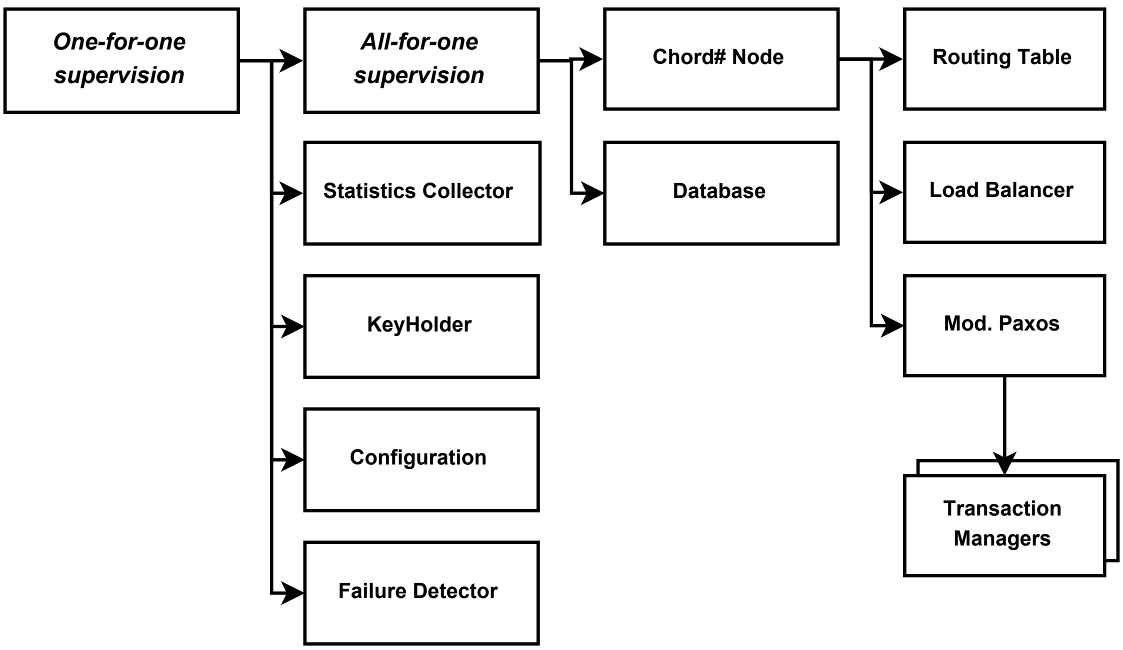
[1] Increase the log level to `info` to get the detailed startup logs. See Sect. 2.7 on page 10

```erlang
%%-----------------------------------------------------------------------------
% @doc add new Scalaris nodes on the local node
-spec(add_nodes/1 :: (non_neg_integer()) -> ok).
add_nodes(0) ->
    ok;
add_nodes(Count) ->
    [ begin
          Desc = util:sup_supervisor_desc(randoms:getRandomId(),
                                          sup_dht_node, start_link),
          supervisor:start_child(main_sup, Desc)
      end || _ <- lists:seq(1, Count) ],
    ok.
```

It calls `add_nodes_loop(Count, Delay)` with a delay of 0. This function starts a new child for the main supervisor `main_sup`. As defined by the parameters, to actually perform the start, the function `sup_dht_node:start_link` 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.

10.2.1 Supervisor-tree of a Scalaris node

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



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

Starting supervisors is a two step process: the supervisor mechanism first calls the `init()` function of the defined module (`sup_dht_node:init()` in this case) and then calls the start function (`start_link` here.

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

File `sup_dht_node.erl`:

```erlang
-spec init([[any()]]) -> {ok, {{one_for_one, MaxRetries::pos_integer(), PeriodInSeconds::pos_integer(
```

```
init([Options]) ->
    InstanceId = string:concat("dht_node_", randoms:getRandomId()),
    process_dictionary:register_process(InstanceId, sup_dht_node, self()),
    boot_server:connect(),
    KeyHolder =
        util:sup_worker_desc(idholder, idholder, start_link,
                                     [InstanceId]),
    Supervisor_AND =
        util:sup_supervisor_desc(cs_supervisor_and, sup_dht_node_core, start_link,
                                         [InstanceId, Options]),
    RingMaintenance =
        util:sup_worker_desc(?RM, ?RM, start_link, [InstanceId]),
    RoutingTable =
        util:sup_worker_desc(routingtable, rt_loop, start_link, [InstanceId]),
    DeadNodeCache =
        util:sup_worker_desc(deadnodecache, dn_cache, start_link, [InstanceId]),
    Vivaldi =
        util:sup_worker_desc(vivaldi, vivaldi, start_link, [InstanceId]),
    Reregister =
        util:sup_worker_desc(dht_node_reregister, dht_node_reregister, start_link,
                                     [InstanceId]),
    DC_Clustering =
        util:sup_worker_desc(dc_clustering, dc_clustering, start_link,
                                     [InstanceId]),
    Cyclon =
        util:sup_worker_desc(cyclon, cyclon, start_link, [InstanceId]),
    Gossip =
        util:sup_worker_desc(gossip, gossip, start_link, [InstanceId]),
    {ok, {{one_for_one, 10, 1},
          [
           Reregister,
           KeyHolder,
           RoutingTable,
           Supervisor_AND,
           Cyclon,
           DeadNodeCache,
           RingMaintenance,
           Vivaldi,
           DC_Clustering,
           Gossip
           %% _RSE
          ]}}.
```

The return value of the `init()` function specifies the child processes of the supervisor and how to start them. Here, we define a list of processes to be observed by a `one_for_one` supervisor. The processes are: `KeyHolder`, `DeadNodeCache`, `RingMaintenance`, `RoutingTable`, and a `Supervisor_AND` process.

The term `{one_for_one, 10, 1}` specifies that the supervisor should try 10 times to restart each process before giving up. `one_for_one` supervision means, that if a single process stops, only that process is restarted. The other processes run independently.

The sup_dht_node:init() is finished and the supervisor module, starts all the defined processes by calling the functions that were defined in the list of the sup_dht_node:init() .

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

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

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

File `sup_dht_node_core.erl`:

```
-spec init([instanceid() | [any()]]) -> {ok, {{one_for_all, MaxRetries::pos_integer(), PeriodInSeconds
init([InstanceId, Options]) ->
    process_dictionary:register_process(InstanceId, sup_dht_node_core, self()),
    Proposer =
        util:sup_worker_desc(proposer, proposer, start_link, [InstanceId]),
    Acceptor =
        util:sup_worker_desc(acceptor, acceptor, start_link, [InstanceId]),
    Learner =
        util:sup_worker_desc(learner, learner, start_link, [InstanceId]),
    Node =
        util:sup_worker_desc(dht_node, dht_node, start_link,
                             [InstanceId, Options]),
    DB =
        util:sup_worker_desc(?DB, ?DB, start_link,
                             [InstanceId]),
    Delayer =
        util:sup_worker_desc(msg_delay, msg_delay, start_link,
                             [InstanceId]),
    TX =
        util:sup_supervisor_desc(sup_dht_node_core_tx, sup_dht_node_core_tx, start_link,
                                 [InstanceId]),
    {ok, {{one_for_all, 10, 1},
          [
           DB,
           Proposer, Acceptor, Learner,
           Node,
           Delayer,
           TX
          ]}}.
```

It defines three processes, that have to be observed using an `one_for_all`-supervisor, which means, that if one fails, all have to be restarted. Passed to the `init` function is the `InstanceId`, a random number to make nodes unique. It was calculated a bit earlier in the code. Exercise: Try to find where.

As you can see from the list, the `DB` is started before the `Node`. This is intended and important, because `dht_node` uses the database, but not vice versa. The supervisor first completely initializes the DB process and afterwards calls `dht_node:start_link()` . We only go into details here, for the latter.

File `dht_node.erl`:

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

-spec start_link(instanceid(), [any()]) -> {ok, pid()}.
start_link(InstanceId, Options) ->
    gen_component:start_link(?MODULE, [InstanceId, Options],
                             [{register, InstanceId, dht_node}, wait_for_init]).
```

`dht_node` implements the `gen_component` behaviour. This component was developed by us to enable us to write code which is similar in syntax and semantics to the examples in [2]. Similar to the `supervisor` behaviour, the component has to provide an `init` function, but here it is used to initialize the state of the component. This function is described in the next section.

Note: `?MODULE` is a predefined Erlang macro, which expands to the module name, the code belongs to (here: `dht_node`).

10.2.4 Initializing a `dht_node`-process

File `dht_node.erl`:

```
%% @doc joins this node in the ring and calls the main loop
-spec init([instanceid() | [any()]]) -> {join, {as_first}, []} | {join, {phase1}, []}.
init([_InstanceId, Options]) ->
    %io:format("~p~n", [Options]),
    % first node in this vm and also vm is marked as first
    % or unit-test
    case lists:member(first, Options) andalso
         (is_unittest() orelse
          application:get_env(boot_cs, first) =:= {ok, true}) of
        true ->
            trigger_known_nodes(),
            idholder:get_id(),
            {join, {as_first}, []};
        _ ->
            idholder:get_id(),
            {join, {phase1}, []}
    end.
```

The `gen_component` behaviour registers the `dht_node` in the process dictionary. Formerly, the process had to do this himself, but we moved this code into the behaviour. If the `dht_node` is the first node, he will start immediately. Otherwise, the process sleeps for a random amount of time. If you would start 1000 processes with `admin:add_nodes(1000)` , the boot-server would receive many join requests at the same time, which is not intended. It will also make the ring stabilization process more complicated. Adding 100s of nodes within a short period of time induces more churn into the system, than the ring maintenance can handle.

Then, the node retrieves its `Id` from the keyholder: `Id = cs_keyholder:get_key()`. In the first call, a random identifier is returned, otherwise the latest set value. If the `dht_node`-process failed and is restarted by its supervisor, this call to the keyholder ensures, that the node still keeps its `Id`, assuming that the keyholder process is not failing. This is important for the load-balancing and for consistent responsibility of nodes to ensure consistent lookup in the structured overlay. Note: the name `Key-holder` actually is an id-holder.

If a node changes its position in the ring for load-balancing, the key-holder will be informed and the `dht_node` finishes itself. This triggers a restart of the corresponding database process via the and-supervisor. When the supervisor restarts both processes, they will retrieve the new position in the ring from the key-holder and join the ring there.

TODO: The supervisor was configured to restart a node at most 10 times. Does that mean, that a node can only change its position in the ring 10 times (caused by load-balancing)?

10.2.5 Actually joining the ring

After retrieving its identifier, the node starts the join process (`dht_node_join:join()`).

File `dht_node_join.erl`:

```
```

The boot-server is contacted to retrieve the known number of nodes in the ring. If the ring is empty, `join_first` is called. Otherwise, `join_ring` is called.

If the ring is empty, the joining node is the only node in the ring and will be responsible for the whole key space. `join_first` just creates a new state for a Scalaris node consisting of an empty routing table, a successorlist containing itself, itself as its predecessor, a reference to itself, its responsibility area from `Id` to `Id` (the full ring), and a load balancing schema.

File `dht_node_join.erl`:

```
```

The macro `?RT` maps to the configured routing algorithm and `?RM` to the configured ring maintenance algorithm. It is defined in `include/scalaris.hrl`. For further details on the routing see Chapter 8.3 on page 29.

The state is defined in

File `dht_node_state.erl`:

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

If a node joins an existing ring, `reliable_get_node` is called for the own `Id` in `dht_node_join:join()` . This lookup delivers the node who is currently responsible for the new node's identifier – the successor for the joining node. If this lookup fails for some reason, it is tried again, by recursively calling the `join()`.

TODO: What, if the `Id` is exactly the same as that of the existing node? This could lead to lookup and responsibility inconsistency? Can this be triggered by the load-balancing? This is a bug, that should be fixed!!!

Then, `dht_node_join:join_ring()` is called:

File `dht_node_join.erl`:

```
```

First the node is initialized. Then it sends a `join` message to the successor including a reference to itself and the chosen `Id`.

The message is received by the old node in `dht_node.erl`. There exists a `{join, X}` handler.

File `dht_node.erl`:

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

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

File `dht_node_join.erl`:

```
-spec join_request(dht_node_state:state(), NewPred::node:node_type()) -> dht_node_state:state().
join_request(State, NewPred) ->
    MyNewInterval =
        intervals:mk_from_nodes(dht_node_state:get(State, node), NewPred),
    {DB, HisData} = ?DB:split_data(dht_node_state:get(State, db), MyNewInterval),

    %%TODO: split data [{Key, Value, Version}], schedule transfer
```

```
    comm:send(node:pidX(NewPred), {join_response, dht_node_state:get(State, pred), HisData}),
    % TODO: better already update our range here directly than waiting for an
    % updated state from the ring_maintenance!
    rm_beh:notify_new_pred(comm:this(), NewPred),
    dht_node_state:set_db(State, DB).
```

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

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

10.2.6 Beginning to serve requests

dht_node_join:join() was called from dht_node:start() , which now continues

File dht_node.erl:

```
%% @doc joins this node in the ring and calls the main loop
-spec init([instanceid() | [any()]]) -> {join, {as_first}, []} | {join, {phase1}, []}.
init([_InstanceId, Options]) ->
    %io:format("~p~n", [Options]),
    % first node in this vm and also vm is marked as first
    % or unit-test
    case lists:member(first, Options) andalso
          (is_unittest() orelse
          application:get_env(boot_cs, first) =:= {ok, true}) of
        true ->
            trigger_known_nodes(),
            idholder:get_id(),
            {join, {as_first}, []};
        _ ->
            idholder:get_id(),
            {join, {phase1}, []}
    end.
```

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

Finally, the loop for request handling is started.

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 bootserver's webserver
`docroot_node`	root directory of the normal node's webserver
`ebin`	the compiled Erlang code (beam files)
`java-api`	a java api to Scalaris
`log`	log files
`src`	contains the Scalaris source code
`test`	unit tests for Scalaris
`user-dev-guide`	contains the sources for this document

12 Java API

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

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

The Javadoc can be found in `java-api/doc/index.html`. The doxygen is in `doc-doxygen/html/index.html`.

We provide two kinds of APIs:

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

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

Bibliography

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

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

Index