Scalaris

Users and Developers Guide

Version 0.8.1+git July 12, 2015

http://scalaris.zib.de

Contents

1. Introduction

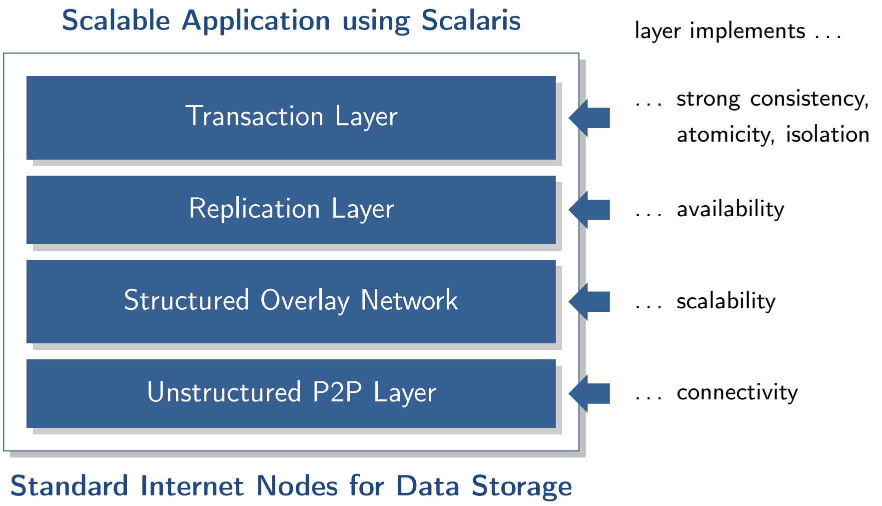
Scalaris is a scalable, transactional, distributed key-value store based on the principles of structured peer-to-peer overlay networks. It can be used as a flexible elastic data store backend to build scalable online services. Without system interruption it scales from a few PCs to thousands of servers. Servers can be added or removed on the fly without any service downtime.

Scalaris takes care of

replication and fail-over	for fault-tolerance
self-management	for low maintenance overhead
automatic data partitioning	for elasticity, load balancing and scalability
strong consistency	to ease development of applications on top of it, as inconsistencies have not to be dealt with
transactions	to support safe atomic updates of several data items at once

The Scalaris project was initiated and is mainly developed by Zuse Institute Berlin (ZIB) and was partly funded by the EU projects Selfman, XtreemOS, Contrail and 4CaaST. Additional information can be found at the project homepage (`http://scalaris.zib.de`) and the corresponding project web page at ZIB (`http://www.zib.de/en/das/projekte/projektdetails/article/scalaris.html`).

The conceptual architecture of Scalaris consists of four layers:



1.1. Scalaris provides strong consistency and partition tolerance

In distributed computing the so called CAP theorem says that there are three desirable properties for distributed systems, but one can only have any two of them.

Strong 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 strong 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 four 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 or retried until the two networks merge again. Note, most other key-value stores tend to sacrifice consistency, which may make it hard for the application developer to detect and handle appearing inconsistencies properly.

1.2. Scientific background

Scalaris is backed by tons of research. It implements both algorithms from the literature and our own research results and combines all of them to a practical overall system. Several aspects of Scalaris were analyzed or/and developed as part of bachelor, diploma, master or PhD theses.

Scalaris in General

Publications of the Scalaris team

F. Schintke. *XtreemFS & Scalaris.* Science & Technology, pp. 54-55, 2013.

A. Reinefeld, F. Schintke, T. Schütt, S. Haridi. *A Scalable, Transactional Data Store for Future Internet Services.* Towards the Future Internet - A European Research Perspective, G. Tselentis et al. (Eds.) IOS Press, pp. 148-159, 2009.

Thorsten Schütt, Monika Moser, Stefan Plantikow, Florian Schintke, Alexander Reinefeld. *A Transactional Scalable Distributed Data Store.* 1st IEEE International Scalable Computing Challenge, co-located with CCGrid'08, 2008.

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. *Scalaris: Reliable Transactional P2P Key/Value Store.* ACM SIGPLAN Erlang Workshop, 2008.

Structured Overlay Networks and Routing

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. 11 on page 77). Our implementation of the routing algorithms is described in more detail in Sect. 9.3 on page 60 and the actual implementation is in `src/rt_chord.erl`. We also implemented Flexible Routing Tables according to [6] which can be found in `src/rt_frtchord.erl` and src/rt_gfrtchord.erl.

Publications of the Scalaris team

Magnus Müller. *Flexible Routing Tables in a Distributed Key-Value Store.* Diploma thesis, HU-Berlin, 2013.

Mikael Högqvist. *Consistent Key-Based Routing in Decentralized and Reconfigurable Data Services.* Doctoral thesis, HU-Berlin, 2012.

Philipp Borgers. *Erweiterung eines verteilten Key-Value-Stores (Riak) um einen räumlichen Index.* Bachelor thesis, FU-Berlin, 2012.

Thorsten Schütt. *Range queries in distributed hash tables.* Doctoral thesis, 2010.

Christian von Prollius. *Ein Peer-to-Peer System mit Bereichsabfragen in PlanetLab.* Diploma thesis, FU-Berlin, 2008.

Jeroen Vlek. *Reducing latency: Log b routing for Chord$^{\#}$.* Bachelor thesis, Uni Amsterdam, 2008.

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. *Range Queries on structured overlay networks.* Computer Communications, 31(2), pp. 280-291, 2008.

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. *A Structured Overlay for Multi-dimensional Range Queries.* Euro-Par Conference, Luc Anne-Marie Kermarrec (Ed.)pp. 503-513, Vol.4641, LNCS, 2007.

Alexander Reinefeld, Florian Schintke, Thorsten Schütt. *P2P Routing of Range Queries in Skewed Multidimensional Data Sets.* ZIB report ZR-07-23, 2007.

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. *Structured Overlay without Consistent Hashing.* Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06) at Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), 16-19 May 2006, Singapore, p. 8, 2006.

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. *Chord$^{\#}$: Structured Overlay Network for Non-Uniform Load-Distribution.* ZIB report ZR-05-40, 2005.

Related work

[6] Hiroya Nagao, Kazuyuki Shudo. *Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set.* In: Peer-to-Peer Computing, IEEE, 2011.

P. Ganesan, B. Yang, H. Garcia-Molina. *One torus to rule them all: Multi- dimensional queries in P2P systems.* In: WebDB2004, 2004.

Luc Onana Alima, Sameh El-Ansary, Per Brand and Seif Haridi. *DKS(N, k, f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications.* The 3rd International workshop on Global and P2P Computing on Large Scale Distributed Systems, (CCGRID 2003), May 2003.

[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`

Transactions

The most interesting part is probably the transaction algorithms. The last description of the algorithms and background is in [7].

The implementation consists of the Paxos algorithm in `src/paxos` and the transaction algorithms itself in `src/transactions` (see also Chap. 10 on page 76).

Publications of the Scalaris team

[7] Florian Schintke, Alexander Reinefeld, Seif Haridi, Thorsten Schütt. *Enhanced Paxos Commit for Transactions on DHTs.* CCGRID, pp. 448-454, 2010.

Florian Schintke. *Management verteilter Daten in Grid- und Peer-to-Peer-Systemen.* Doctoral thesis, HU-Berlin, 2010.

Monika Moser, Seif Haridi, Tallat Shafaat, Thorsten Schütt, Mikael Högqvist, Alexander Reinefeld. *Transactional DHT Algorithms.* ZIB report ZR-09-34, 2009.

Stefan Plantikow, Alexander Reinefeld, Florian Schintke. *Transactions and Concurrency Control for Peer-to-Peer-Wikis.* In: Making Grids Work, Marco Danelutto, Paraskevi Fragopoulo, Vladimir Getov (Eds.)pp. 337-349, 2008.

B. Mejías, M. Högqvist, P. Van Roy. *Visualizing Transactional Algorithms for DHTs.* IEEE P2P Conference, 2008.

Monika Moser, Seif Haridi. *Atomic Commitment in Transactional DHTs.* Proceedings of the CoreGRID Symposium, 2007.

S. Plantikow, A. Reinefeld, F. Schintke. *Distributed Wikis on Structured Overlays.* CoreGrid Workshop on Grid Programming Models, Grid and P2P System Architecture, Grid Systems, Tools and Environments, 2007.

S. Plantikow, A. Reinefeld, F. Schintke. *Transactions for Distributed Wikis on Structured Overlays.* DSOM, Alexander Clemm, Lisandro Granville, Rolf Stadler (Eds.)pp. 256-267, Vol.4785, LNCS, 2007.

Stefan Plantikow. *Transaktionen für verteilte Wikis auf strukturierten Overlay-Netzwerken.* Diploma thesis, HU-Berlin, 2007.

Related work

Björn Kolbeck, Mikael Högqvist, Jan Stender, Felix Hupfeld. *Flease – Lease Coordination Without a Lock Server.* Intl. Parallel and Distributed Processing Symposium, pp. 978-988, 2011.

J. Gray, L. Lamport. *Consensus on transaction commit.* ACM Trans. Database Syst., 31(1):133–160, 2006.

L. Lamport. *Fast Paxos.* Distributed Computing, 19(2):79–103, 2006.

L. Lamport. *Paxos Made Simple.* SIGACT News, 32(4):51–58, December 2001.

L. Lamport. *The Part-Time Parliament.* ACM Trans. Comput. Syst., 16(2):133–169, 1998.

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 [8].

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

Publications of the Scalaris team

Paolo Costa, Guillaume Pierre, Alexander Reinefeld, Thorsten Schütt, Maarten van Steen. *Sloppy Management of Structured P2P Services.* Proceedings of the 3^{rd} International Workshop on Hot Topics in Autonomic Computing (HotAC III), co-located with IEEE ICAC'08, 2008.

Related work

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

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

Gossiping and Topology Inference

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`.

For some algorithms, we use estimates of global information. These estimates are aggregated with the help of gossiping techniques [9].

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

Publications of the Scalaris team

Jens V. Fischer. *A Gossiping Framework for Scalaris.* Bachelor thesis, FU-Berlin, 2014.

Marie Hoffmann. *Approximate Algorithms for Distributed Systems.* Master thesis, FU-Berlin, 2012.

Thorsten Schütt, Alexander Reinefeld, Florian Schintke, Marie Hoffmann. *Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers.* Peer-to-Peer Computing, pp. 147-150, 2009.

Related work

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

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

Load-Balancing

Publications of the Scalaris team

Maximilian Michels. *Request-Based Load Balancing in Distributed Hash Tables.* Master thesis, FU-Berlin, 2014.

Mikael Högqvist, Nico Kruber. *Passive/Active Load Balancing with Informed Node Placement in DHTs.* IWSOS, Thrasyvoulos Spyropoulos, Karin Hummel (Eds.)pp. 101-112, Vol.5918, Lecture Notes in Computer Science, 2009.

Nico Kruber. *DHT Load Balancing with Estimated Global Information.* Diploma thesis, HU-Berlin, 2009.

Mikael Högqvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, Thorsten Schütt. *Using Global Information for Load Balancing in DHTs.* Workshop on Decentralized Self Management for Grids, P2P, and User Communities, 2008.

Simon Rieche. *Lastbalancierung in Peer-to-Peer Systemen.* Diploma thesis, FU-Berlin, 2003.

Related work

David R. Karger, Matthias Ruhl. *Simple efficient load-balancing algorithms for peer-to-peer systems.* Theory of Computing Systems, 39(6):787–804, November 2006.

Ashwin R. Bharambe, Mukesh Agrawal, Srinivasan Seshan. *Mercury: support- ing scalable multi-attribute range queries.* SIGCOMM Comput. Commun. Rev., 34(4):353–366, 2004.

Self-Management

Publications of the Scalaris team

T. Schütt, A. Reinefeld, F. Schintke, C. Hennig. *Self-Adaptation in Large-Scale Systems.* Architectures and Languages for Self-Managing Distributed Systems (SelfMan@SASO), 2009.

P. Van Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, T. Coupaye. *Self Management for Large-Scale Distributed Systems.* Formal Methods for Components and Objects 2007 (FMCO 2007), 2008.

P. Van Roy, A. Ghodsi, S. Haridi, J.-B. Stefani, T. Coupaye, A. Reinefeld, E. Winter, R. Yap. *Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components*, 2005.

Other Topics

Publications of the Scalaris team

Data Placement

M. Högqvist, S. Plantikow. *Towards Explicit Data Placement in Scalable Key/Value Stores.* Architectures and Languages for Self-Managing Distributed Systems (SelfMan@SASO), 2009.

Consistency

Tallat Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, Alexander Reinefeld. *Key-Based Consistency and Availability in Structured Overlay Networks.* International ICST Conference on Scalable Information Systems, 2008.

Tallat Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks.* Coregrid Integration Workshop, 2008.

Snapshots

Stefan Keidel. *Snapshots in Scalaris.* Diploma thesis, HU-Berlin, 2012.

Replication and Replica Repair

Maik Lange. *Redundanzverwaltung in konsistenten verteilten Datenbanken.* Diploma thesis, HU-Berlin, 2012.

Part I.

Users Guide

2. Download and Installation

2.1. Requirements

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

- Erlang R13B01 or newer
- OpenSSL (required by Erlang's crypto module)
- GNU-like Make and autoconf (not required on Windows)

To build the Java API (and its command-line client) the following programs are also required:

- 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.

To build the Python API (and its command-line client) the following programs are also required:

- Python >= 2.6

2.2. Download

The sources can be obtained from https://github.com/scalaris-team/scalaris. RPM and DEB packages are available from http://download.opensuse.org/repositories/home:/scalaris/ for various Linux distributions.

2.2.1. Development Branch

You find the latest development version in the git repository:

```
git clone https://github.com/scalaris-team/scalaris.git scalaris
```

2.2.2. Releases

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

2.3. Build

2.3.1. 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.

2.3.2. 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 `firstnode.bat`, `joining_node.bat`
- run `firstnode.bat` or one of the other start scripts in the cmd window

`build.bat` will generate a `Emakefile` if there is none yet. On certain older Erlang versions, you will need to adapt the `Emakefile`. Please refer to the `build.bat` and `configure.ac` for the available configuration parameters and their meaning.

For the most recent description please see the FAQ at http://scalaris.zib.de/faq.html.

2.3.3. 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
```

2.3.4. Python-API

The Python API for Python 2.* (at least 2.6) is located in the `python-api` directory. Files for Python 3.* can be created using `2to3` from the files in `python-api`. The following command will use `2to3` to convert the modules and place them in `python3-api`.

```
%> make python3
```

Both versions of python will compile required modules on demand when executing the scripts for the first time. However, pre-compiled modules can be created with:

```
%> make python
%> make python3
```

2.3.5. Ruby-API

The Ruby API for Ruby >= 1.8 is located in the `ruby-api` directory. Compilation is not necessary.

2.4. 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`, by default. But it is more convenient to build an RPM and install it. On openSUSE, for example, do the following:

```
export SCALARIS_GIT=https://raw.githubusercontent.com/scalaris-team/scalaris/master
for package in main bindings; do
  mkdir -p ${package}
  cd ${package}
  wget ${SCALARIS_GIT}/contrib/packages/${package}/checkout.sh
  ./checkout.sh
  cp * /usr/src/packages/SOURCES/
  rpmbuild -ba scalaris*.spec
  cd ..
done
```

If any additional packages are required in order to build an RPM, `rpmbuild` will print an error.

Your source and binary RPMs will be generated in `/usr/src/packages/SRPMS` and `RPMS`.

We build RPM and DEB packages for the newest stable Scalaris version as well as snapshots of the git master branch and provide them using the Open Build Service. The latest stable version is available at `http://download.opensuse.org/repositories/home:/scalaris/`. The latest git snapshot is available at `http://download.opensuse.org/repositories/home:/scalaris:/svn`.

For those distributions which provide a recent-enough Erlang version, we build the packages using their Erlang package and recommend using the same version that came with the distribution. In this case we do not provide Erlang packages in our repository.

Exceptions are made for (old) openSUSE-based and RHEL-based distributions:

- For older openSUSE or SLE distributions, we provide Erlang R14B04.
- For RHEL-based distributions (CentOS 5,6,7, RHEL 5,6,7) we included the Erlang package from the EPEL repository of RHEL 6 and RHEL 7, respectively.

2.5. Testing the Installation

After installing Scalaris you can check your installation and perform some basic tests using

```
%> scalarisctl checkinstallation
```

For further details on `scalarisctl` see Section 3.3 on page 18.

3. Setting up Scalaris

3.1. Runtime 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` and needs to be altered for a distributed setup (see Section 3.2.2 on page 17). A third way to alter the configuration of Scalaris, e.g. port numbers, is to use parameters for the `scalarisctl` script (ref. Section 3.3 on page 18. The following example changes the port to 14195 and the YAWS port to 8080:

```
%> ./bin/scalarisctl -p 14194 -y 8080
```

The configuration precedence is as follows:

1. configuration parameters of `scalarisctl`
2. `bin/scalaris.local.cfg`
3. `bin/scalaris.cfg`

3.1.1. 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` 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 11 on page 77, 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}.
```

3.2. Running Scalaris

A Scalaris deployment can have a *management server* as well as *regular nodes*. The management server is optional and provides a global view on all nodes of a Scalaris deployment which contact this server, i.e. have its address specified in the `mgmt_server` configuration setting. A regular node is either the first node in a system or joins an existing system deployment.

3.2.1. Running on a local machine

Open at least two shells. In the first, inside the Scalaris directory, start the first node (`firstnode.bat` on Windows):

```
%> ./bin/firstnode.sh
```

This will start a new Scalaris deployment with a single node, including a management server. On success http://localhost:8000 should point to the management interface page of the management server. The main page will show you the number of nodes currently in the system. A first Scalaris node should have started and the number should show 1 node. 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 Section 4.1 on page 19 for application APIs.

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

```
%> ./bin/joining_node.sh
```

The second node will read the configuration file and use this information to contact a number of known nodes (set by the `known_hosts` configuration setting) and join the ring. It will also register itself with the management server. 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/joining_node.sh 2
```

In a fourth shell:

```
%> ./bin/joining_node.sh 3
```

This will add two further nodes to the deployment. The `./bin/joining_node.sh` script accepts a number as its parameter which will be added to the started node's name, i.e. `1` will lead to a node named `node1`. The web pages at http://localhost:8000 should show the additional nodes.

3.2.2. Running distributed

Scalaris can be installed on other machines in the same way as described in Section 2.4 on page 14. In the default configuration, nodes will look for the management server on `127.0.0.1` on port 14195. To run Scalaris distributed over several nodes, each node requires a `bin/scalaris.local.cfg` pointing to the node running the management server (if available) and containing a list of known nodes. Without a list of known nodes, a joining node will not know where to join.

In the following example, the `mgmt_server`'s location is defined as an IP address plus a TCP port and its Erlang-internal process name. If the deployment should not use a management server, replace the setting with an invalid address, e.g. ' null '.

File `scalaris.local.cfg`:

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

% IP Address, Port, and label of a node which is already in the system
{known_hosts, [{{127,0,0,1}, 14195, service_per_vm}]}.
```

If you are starting the management server using `firstnode.sh`, it will listen on port 14195 and you only have to change the the IP address in the configuration file. Otherwise the other nodes will not find the management server. Calling `./bin/joining_node.sh` on a remote machine will start the node and automatically contact the configured management server.

3.3. Custom startup using `scalarisctl`

On Linux you can also use the `scalarisctl` script to start a management server and 'regular' nodes directly.

```
%> ./bin/scalarisctl -h
```

```
usage: scalarisctl [options] <cmd>
 options:
    -h           - print this help message
    -d           - daemonize
    --screen     - if daemonized, put an interactive session into screen
    -e <params> - pass additional parameters to erl
    -n <name>    - Erlang process name (default 'node')
    -c <cookie> - Erlang cookie to use (for distributed Erlang)
                   (default 'chocolate chip cookie')
    -p <port>    - TCP port for the Scalaris node
    -y <port>    - TCP port for the built-in webserver (YAWS)
    -k <key>     - join at the given key
    -j <list>    - join at the given list of keys
    -v           - verbose
    -l <dir>     - use this logdir base directory (will create a sub-folder
                   per node)
    --dist-erl-port <port>
                 - (single) port distributed erlang listens on
    --nodes-per-vm <number>
                 - number of Scalaris nodes to start inside the VM
    -t <stype>   - select start type: first|joining|quorum|recover|nostart|first_nostart
    -m           - start global Scalaris management server
 <cmd>:
    checkinstallation
                 - test installation
    start        - start services (see -m and -t)
    stop         - stop a scalaris process defined by its name (see -n)
    restart      - restart a scalaris process by its name (see -n)

    list         - list locally running Erlang VMs
    debug        - connect to a running node via an Erlang shell
    dbg-check-ring <ring-size> <attempts>
                 - checks (up to) <attempts> times whether Scalaris has
                   <ring-size> nodes and the ring maintenance has settled
                   (requires a mgmt_server)
```

4. Using the system

Scalaris can be used with one of the provided command line interfaces or by using one of the APIs in a custom program. The following sections will describe the APIs in general, each API in more detail and the use of our command line interfaces.

4.1. Application Programming Interfaces (APIs)

Currently we offer the following APIs:

- an *Erlang API* running on the node Scalaris is run
  (functions can be called using remote connections with distributed Erlang)
- a *Java API* using Erlang's `JInterface` library
  (connections are established using distributed Erlang)
- a generic *JSON API*
  (offered by an integrated HTTP server running on each Scalaris node)
- a *Python API* for Python >= 2.6 using JSON to talk to Scalaris.
- a *Ruby API* for Ruby >= 1.8 using JSON to talk to Scalaris.

Each API contains methods for accessing functions from the three layers Scalaris is composed of. Table 4.1 shows the modules and classes of Erlang, Java, Python and Ruby and their mapping to these layers. Details about the supported operations and how to access them in each of the APIs are provided in Section 4.1.2 on page 21. A more detailed discussion about the generic JSON API including examples of JSON calls is shown in Section 4.1.3 on page 27.

	Erlang module	Java class in `de.zib.scalaris`	JSON file in `<URL>/api/`	Python / Ruby class in module `scalaris`
Transaction Layer	`api_tx`	`Transaction`, `TransactionSingleOp`	`tx.yaws`	`Transaction`, `TransactionSingleOp`
Replication Layer	`api_rdht`	`ReplicatedDHT`	`rdht.yaws`	`ReplicatedDHT`

P2P Layer	`api_dht`
	`api_dht_raw`
	`api_vm`
	`api_monitor`
		
		`ScalarisVM`
		`Monitor`
			
			`dht_raw.yaws`
			
			`monitor.yaws`	

Table 4.1.: Layered API structure

	Erlang	Java	JSON	Python	Ruby
boolean	`boolean()`	`bool`, `Boolean`	`true`, `false`	`True`, `False`	`true`, `false`

integer	`integer()`	`int`, `Integer`
		`long`, `Long`
		`BigInteger`
			`int`	`int`	`Fixnum`,
					`Bignum`

float	`float()`	`double`, `Double`	`int frac`
			`int exp`
			`int frac exp`
				`float`	`Float`

string	`string()`	`String`	`string`	`str`	`String`
binary	`binary()`	`byte[]`	`string` (base64-encoded)	`bytearray`	`String`
list(type)	`[type()]`	`List<Object>`	`array`	`list`	`Array`
JSON	`json_obj()`*	`Map<String, Object>`	`object`	`dict`	`Hash`
custom	`any()`	`OtpErlangObject`	/	/	/

*
```
json_obj() :: {struct, [Key::atom() | string(), Value::json_val()]}
json_val() :: string() | number() | json_obj() | {array, [any()]} | true | false | null
```

Table 4.2.: Types supported by the Scalaris APIs

4.1.1. Supported Types

Different programming languages have different types. In order for our APIs to be compatible with each other, only a subset of the available types is officially supported.

Keys are always strings. In order to avoid problems with different encodings on different systems, we suggest to only use ASCII characters.

For *values* we distinguish between *native*, *composite* and *custom* types (refer to Table 4.2 for the mapping to the language-specific types of each API).

Native types are

- boolean values
- integer numbers
- floating point numbers
- strings and
- binary objects (a number of bytes).

Composite types are

- lists of the following elements:
  - native types *(except binary objects!)*,
  - composite types
- objects in JavaScript Object Notation (JSON)[1]

Custom types include any Erlang term not covered by the previous types. Special care needs to be taken using custom types as they may not be accessible through every API or may be misinterpreted by an API. The use of them is discouraged.

[1]see http://json.org/

4.1.2. Supported Operations

Most operations are available to all APIs, but some (especially convenience methods) are API- or language-specific. The following paragraphs provide a brief overview of what is available to which API. For a full reference, see the documentation of the specific API.

Transaction Layer

Read Reads the value stored at a given key using quorum read.

Erlang	`api_tx:read(Key)`
Java:	`TransactionSingleOp.read(Key)`
JSON:	`tx.yaws/read(Key)`
Python:	`TransactionSingleOp.read(Key)`
Ruby:	`TransactionSingleOp.read(Key)`

Write Writes a value to a given key.

Erlang	`api_tx:write(Key, Value)`
Java:	`TransactionSingleOp.write(Key, Value)`
JSON:	`tx.yaws/write(Key, Value)`
Python:	`TransactionSingleOp.write(Key, Value)`
Ruby:	`TransactionSingleOp.write(Key, Value)`

"Add to" & "Delete from" List Operations For the list stored at a given key, first add all elements from a given list, then remove all elements from a second given list.

Erlang	`api_tx:add_del_on_list(Key, ToAddList, ToRemoveList)`
Java:	`TransactionSingleOp.addDelOnList(Key, ToAddList, ToRemoveList)`
JSON:	`tx.yaws/add_del_on_list(Key, ToAddList, ToRemoveList)`
Python:	`TransactionSingleOp.add_del_on_list(Key, ToAddList, ToRemoveList)`
Ruby:	`TransactionSingleOp.add_del_on_list(Key, ToAddList, ToRemoveList)`

Add to a number Adds a given number to the number stored at a given key.

Erlang	`api_tx:add_on_nr(Key, ToAddNumber)`
Java:	`TransactionSingleOp.addOnNr(Key, ToAddNumber)`
JSON:	`tx.yaws/add_on_nr(Key, ToAddList, ToAddNumber)`
Python:	`TransactionSingleOp.add_on_nr(Key, ToAddNumber)`
Ruby:	`TransactionSingleOp.add_on_nr(Key, ToAddNumber)`

Atomic Test and Set Writes the given (new) value to a key if the current value is equal to the given old value.

Erlang	`api_tx:test_and_set(Key, OldValue, NewValue)`
Java:	`TransactionSingleOp.testAndSet(Key, OldValue, NewValue)`
JSON:	`tx.yaws/add_on_nr(Key, OldValue, NewValue)`
Python:	`TransactionSingleOp.test_and_set(Key, OldValue, NewValue)`
Ruby:	`TransactionSingleOp.test_and_set(Key, OldValue, NewValue)`

Bulk Operations Executes multiple requests, i.e. operations, where each of them will be committed.

Collecting requests and executing all of them in a single call yields better performance than executing all on their own.

Erlang	`api_tx:req_list_commit_each(RequestList)`
Java:	`TransactionSingleOp.req_list(RequestList)`
JSON:	`tx.yaws/req_list_commit_each(RequestList)`
Python:	`TransactionSingleOp.req_list(RequestList)`
Ruby:	`TransactionSingleOp.req_list(RequestList)`

Transaction Layer (with TLog)

Read (with TLog) Reads the value stored at a given key using quorum read as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*.

Erlang	`api_tx:read(TLog, Key)`
Java:	`Transaction.read(Key)`
JSON:	n/a - use `req_list`
Python:	`Transaction.read(Key)`
Ruby:	`Transaction.read(Key)`

Write (with TLog) Writes a value to a given key as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*.

Erlang	`api_tx:write(TLog, Key, Value)`
Java:	`Transaction.write(Key, Value)`
JSON:	n/a - use `req_list`
Python:	`Transaction.write(Key, Value)`
Ruby:	`Transaction.write(Key, Value)`

"Add to" & "Delete from" List Operations (with TLog) For the list stored at a given key, first add all elements from a given list, then remove all elements from a second given list as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*.

Erlang	`api_tx:add_del_on_list(TLog, Key, ToAddList, ToRemoveList)`
Java:	`Transaction.addDelOnList(Key, ToAddList, ToRemoveList)`
JSON:	n/a - use `req_list`
Python:	`Transaction.add_del_on_list(Key, ToAddList, ToRemoveList)`
Ruby:	`Transaction.add_del_on_list(Key, ToAddList, ToRemoveList)`

Add to a number (with TLog) Adds a given number to the number stored at a given key as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*.

Erlang	`api_tx:add_on_nr(TLog, Key, ToAddNumber)`
Java:	`Transaction.addOnNr(Key, ToAddNumber)`
JSON:	n/a - use `req_list`
Python:	`Transaction.add_on_nr(Key, ToAddNumber)`
Ruby:	`Transaction.add_on_nr(Key, ToAddNumber)`

Atomic Test and Set (with TLog) Writes the given (new) value to a key if the current value is equal to the given old value as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*.

Erlang	`api_tx:test_and_set(TLog, Key, OldValue, NewValue)`
Java:	`Transaction.testAndSet(Key, OldValue, NewValue)`
JSON:	`tx.yaws/test_and_set(Key, OldValue, NewValue)`
Python:	`Transaction.test_and_set(Key, OldValue, NewValue)`
Ruby:	`Transaction.test_and_set(Key, OldValue, NewValue)`

Bulk Operations (with TLog) Executes multiple requests, i.e. operations, as an additional part of a previous transaction or for starting a new one *(no auto-commit!)*. Only one `commit` request is allowed per call!

Collecting requests and executing all of them in a single call yields better performance than executing all on their own.

Erlang	`api_tx:req_list(RequestList)`, `api_tx:req_list(TLog, RequestList)`
Java:	`Transaction.req_list(RequestList)`
JSON:	`tx.yaws/req_list(RequestList)`, `req_list(TLog, RequestList)`
Python:	`Transaction.req_list(RequestList)`
Ruby:	`Transaction.req_list(RequestList)`

Replication Layer

Delete Tries to delete a value at a given key.

Warning: This can only be done outside the transaction layer and is thus not absolutely safe. Refer to the following thread on the mailing list: http://groups.google.com/group/scalaris/browse_thread/thread/ff1d9237e218799.

Erlang	`api_rdht:delete(Key)`, `api_rdht:delete(Key, Timeout)`
Java:	`ReplicatedDHT.delete(Key)`, `ReplicatedDHT.delete(Key, Timeout)`
JSON:	`rdht.yaws/delete(Key)`, `rdht.yaws/delete(Key, Timeout)`
Python:	`ReplicatedDHT.delete(Key)`, `ReplicatedDHT.delete(Key, Timeout)`
Ruby:	`ReplicatedDHT.delete(Key)`, `ReplicatedDHT.delete(Key, Timeout)`

Get Replica Keys Gets the (hashed) keys used for the replicas of a given (user) key (ref. Section P2P Layer).

Erlang	`api_rdht:get_replica_keys(Key)`
Java:	n/a
JSON:	n/a
Python:	n/a
Ruby:	n/a

P2P Layer

Hash Key Generates the hash of a given (user) key.

Erlang	`api_dht:hash_key(Key)`
Java:	n/a
JSON:	n/a
Python:	n/a
Ruby:	n/a

Get Replica Keys Gets the (hashed) keys used for the replicas of a given (hashed) key.

Erlang	`api_dht_raw:get_replica_keys(HashedKey)`
Java:	n/a
JSON:	n/a
Python:	n/a
Ruby:	n/a

Range Read Reads all Key-Value pairs in a given range of (hashed) keys.

Erlang	`api_dht_raw:range_read(StartHashedKey, EndHashedKey)`
Java:	n/a
JSON:	`dht_raw.yaws/range_read(StartHashedKey, EndHashedKey)`
Python:	n/a
Ruby:	n/a

P2P Layer (VM Management)

Get Scalaris Version Gets the version of Scalaris running in the requested Erlang VM.

Erlang	`api_vm:get_version()`
Java:	`ScalarisVM.getVersion()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Get Node Info Gets various information about the requested Erlang VM and the running Scalaris code, e.g. Scalaris version, erlang version, memory use, uptime.

Erlang	`api_vm:get_info()`
Java:	`ScalarisVM.getInfo()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Get Information about Different VMs Get connection info about other Erlang VMs running Scalaris nodes. Note: This info is provided by the `cyclon` service built into Scalaris.

Erlang	`api_vm:get_other_vms(MaxVMs)`
Java:	`ScalarisVM.getOtherVMs(MaxVMs)`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Get Number of Scalaris Nodes in the VM Gets the number of Scalaris nodes running inside the Erlang VM.

Erlang	`api_vm:number_of_nodes()`
Java:	`ScalarisVM.getNumberOfNodes()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Get Scalaris Nodes Gets a list of Scalaris nodes running inside the Erlang VM.

Erlang	`api_vm:get_nodes()`
Java:	`ScalarisVM.getNodes()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Add Scalaris Nodes Starts additional Scalaris nodes inside the Erlang VM.

Erlang	`api_vm:add_nodes(Number)`
Java:	`ScalarisVM.addNodes(Number)`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Shutdown Scalaris Nodes Gracefully kill some Scalaris nodes inside the Erlang VM. This will first move the data from the nodes to other nodes and then shut them down.

Erlang	`api_vm:shutdown_node(Name)`, `api_vm:shutdown_nodes(Count)`, `api_vm:shutdown_nodes_by_name(Names)`
Java:	`ScalarisVM.shutdownNode(Name)`, `ScalarisVM.shutdownNodes(Number)`, `ScalarisVM.shutdownNodesByName(Names)`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Kill Scalaris Nodes Immediately kills some Scalaris nodes inside the Erlang VM.

Erlang	`api_vm:kill_node(Name)`, `api_vm:kill_nodes(Count)`, `api_vm:kill_nodes_by_name(Names)`
Java:	`ScalarisVM.killNode(Name)`, `ScalarisVM.killNodes(Number)`, `ScalarisVM.killNodesByName(Names)`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Shutdown the Erlang VM Gracefully shuts down all Scalaris nodes in the Erlang VM and then exits.

Erlang	`api_vm:shutdown_vm()`
Java:	`ScalarisVM.shutdownVM()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

Kill the Erlang VM Immediately kills all Scalaris nodes in the Erlang VM and then exits.

Erlang	`api_vm:kill_vm()`
Java:	`ScalarisVM.killVM()`
JSON:	n/a
Python:	n/a
Ruby:	n/a

P2P Layer (Monitoring)

Get Node Info Gets some information about the node, e.g. Scalaris version, Erlang version, number of Scalaris nodes in the VM.

Erlang	`api_monitor:get_node_info()`
Java:	`Monitor.getNodeInfo()`
JSON:	`monitor.yaws/get_node_info()`
Python:	n/a
Ruby:	n/a

Get Node Performance Gets some performance information about the node, e.g. the average latency and standard deviation of transactional operations.

Erlang	`api_monitor:get_node_performance()`
Java:	`Monitor.getNodePerformance()`
JSON:	`monitor.yaws/get_node_performance()`
Python:	n/a
Ruby:	n/a

Get Service Info Gets some information about the whole Scalaris ring (may be estimated if no management server is used). Includes the overall load and the total number of nodes in the ring.

Erlang	`api_monitor:get_service_info()`
Java:	`Monitor.getServiceInfo()`
JSON:	`monitor.yaws/get_service_info()`
Python:	n/a
Ruby:	n/a

Get Service Performance Gets some performance information about the whole Scalaris ring, e.g. the average latency and standard deviation of transactional operations. Both are aggregated and may be estimates.

Erlang	`api_monitor:get_service_performance()`
Java:	`Monitor.getServicePerformance()`
JSON:	`monitor.yaws/get_service_performance()`
Python:	n/a
Ruby:	n/a

Convenience Methods / Classes

Connection Pool Implements a thread-safe pool of connections to Scalaris instances. Can be instantiated with a fixed maximum number of connections. Connections are either taken from a pool of available connections or are created on demand. If finished, a connection can be put back into the pool.

Erlang	n/a
Java:	`ConnectionPool`
JSON:	n/a
Python:	`ConnectionPool`
Ruby:	n/a

Connection Policies Defines policies on how to select a node to connect to from a set of possible nodes and whether and how to automatically re-connect.

Erlang	n/a
Java:	`ConnectionPolicy`
JSON:	n/a
Python:	n/a
Ruby:	n/a

4.1.3. 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 contains a list of the results of the requests and - in case of a transaction - a TransLog. 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. Request lists are also supported for single read/write operations, i.e. every single operation is committed on its own.

The JSON-API can be accessed via the Scalaris-Web-Server running on port 8000 by default and pages under `<URL>/api/`. For backwards-compatibility the page `<URL>/jsonrpc.yaws` provides some functions otherwise provided by the different pages under `<URL>/api/` but beware that this may be removed in future. Other examples include `http://localhost:8000/api/tx.yaws`. See Table 4.1 on page 19 for a mapping of the layers to the different pages. Requests are issued by sending a JSON object with header `"Content-type"="application/json"` to this URL. The result will then be returned as a JSON object with the same content type. The following table shows how both objects look like:

Request

```
{
  "jsonrpc": "2.0",
  "method" : "<method>",
  "params" : [<params>],
  "id"     : <number>
}
```

Result

```
{
  "result" : <result_object>,
  "id"     : <number>
}
```

The `id` in the request can be an arbitrary number which identifies the request and is returned in the result. The following operations (shown as `<method>(<params>)`) are currently supported (the given result is the `<result_object>` mentioned above):

generic, e.g. for testing - `<URL>/api/*.yaws`

- `nop(Value)` - no operation, result:

```
"ok"
```

single operations, e.g. read/write - `<URL>/api/tx.yaws`:

- `req_list_commit_each(<req_list_ce>)` - commit each request in the list, result:

```
{[{"status": "ok"} or {"status": "ok", "value": <json_value>} or
  {"status": "fail", "reason": "timeout" or "abort" or "not_found" or
                               "not_a_list" or "not_a_number"} or
  {"status": "fail", "reason": "key_changed", "value": <json_value>}]}
```

- `read(<key>)` - read the value at `key`, result:

```
{"status": "ok", "value", <json_value>} or
{"status": "fail", "reason": "timeout" or "not_found"}
```

- `write(<key>, <json_value>)` - write `value` (inside `json_value`) to `key`, result:

```
{"status": "ok"} or
{"status": "fail", "reason": "timeout" or "abort"}
```

- `add_del_on_list(<key>, ToAdd, ToRemove)` - adding to / removing from a list (for the list at `key` adds all values in the `ToAdd` list and then removes all values in the `ToRemove` list; if there is no value at `key`, uses an empty list - both value lists are `[<value>]`), result:

```
{"status": "ok"} or
{"status": "fail", "reason": "timeout" or "abort" or "not_a_list"}
```

- `add_on_nr(<key>, <value>)` - adding to a number (adds `value` to the number at `key` - both values must be numbers), result:

```
{"status": "ok"} or
{"status": "fail", "reason": "timeout" or "abort" or "not_a_number"}
```

- `test_and_set(<key>, OldValue, NewValue)` - atomic test-and-set (write `NewValue` to `key` if the current value is `OldValue` - both values are `<json_value>`), result:

```
{"status": "ok"} or
{"status": "fail", "reason": "timeout" or "abort" or "not_found"} or
{"status": "fail", "reason": "key_changed", "value": <json_value>}
```

transactions - <URL>/api/tx.yaws:

- req_list(<req_list>) - process a list of requests, result:

```
{"tlog": <tlog>,
 "results": [{"status": "ok"} or {"status": "ok", "value": <json_value>} or
             {"status": "fail", "reason": "timeout" or "abort" or "not_found" or
                                          "not_a_list" or "not_a_number"} or
             {"status": "fail", "reason": "key_changed", "value": <json_value>}]}
```

- req_list(<tlog>, <req_list>) - process a list of requests with a previous translog, result:

```
{"tlog": <tlog>,
 "results": [{"status": "ok"} or {"status": "ok", "value": <json_value>} or
             {"status": "fail", "reason": "timeout" or "abort" or "not_found" or
                                          "not_a_list" or "not_a_number"} or
             {"status": "fail", "reason": "key_changed", "value": <json_value>}]}
```

replication layer functions - <URL>/api/rdht.yaws:

- delete(<key>) - delete the value at key, default timeout 2s, result:

```
{"ok": <number>, "results": ["ok" or "locks_set" or "undef"]} or
{"failure": "timeout", "ok": <number>, "results": ["ok" or "locks_set" or "undef"]}
```

- delete(<key>, Timeout) - delete the value at key with a timeout of Timeout Milliseconds, result:

```
{"ok": <number>, "results": ["ok" or "locks_set" or "undef"]} or
{"failure": "timeout", "ok": <number>, "results": ["ok" or "locks_set" or "undef"]}
```

raw DHT functions - <URL>/api/dht_raw.yaws:

- range_read(From, To) - read a range of (raw) keys, result:

```
{"status": "ok" or "timeout",
 "value": [{"key": <key>, "value": <json_value>, "version": <version>}]}
```

monitor - <URL>/api/monitor.yaws:

- get_node_info() - gets some information about the node, result:

```
{"status": "ok" or "timeout",
 "value": [{"scalaris_version": <version_string>,
            "erlang_version": <version_string>,
            "dht_nodes": <number>}]}
```

- get_node_performance() - gets some performance information about the node, result:

```
{"status": "ok" or "timeout",
 "value": [{"latency_avg": <perf_data>, "latency_stddev": <perf_data>}]}
```

- get_service_info() - gets some information about the Scalaris ring, result:

```
{"status": "ok" or "timeout",
 "value": [{"total_load": <number>, "nodes": <number>}]}
```

- get_service_performance() - gets some performance information about the Scalaris ring, result:

```
{"status": "ok" or "timeout",
 "value": [{"latency_avg": <perf_data>, "latency_stddev": <perf_data>}]}
```

Note:

```
<json_value> = {"type": "as_is" or "as_bin", "value": <value>}
<operation> = {"read": <key>} or {"write", {<key>: <json_value>}} or
              {"add_del_on_list": {"key": <key>, "add": [<value>], "del": [<value>]}} or
              {"add_on_nr": {<key>: <value>}} or
              {"test_and_set": {"key": <key>, "old": <json_value>, "new": <json_value>}}
<req_list_ce> = [<operation>]
<req_list> = [<operation> or {"commit", _}]
<perf_data> = {<number>: <perf_val>,...}
```

The <value> inside <json_value> is either a base64-encoded string representing a binary object (type = "as_bin") or the value itself (type = "as_is").

JSON-Example

The following example illustrates the message flow:

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

```
{"jsonrpc": "2.0",
 "method": "req_list",
 "params": [
  [ { "write": { "keyA": {"type": "as_is", "value": "valueA"} } },
    { "write": { "keyB": {"type": "as_is", "value": "valueB"} } },
    { "commit": "" } ]
 ],
 "id": 0
}
```

← Scalaris sends results back

```
{"error": null,
 "result": {
  "results": [ {"status": "ok"}, {"status": "ok"}, {"status": "ok"} ],
  "tlog": <TLOG> // this is the translog for further operations!
 },
 "id": 0
}
```

In a second transaction: Read the two keys →

```
{"jsonrpc": "2.0",
 "method": "req_list",
 "params": [
  [ { "read": "keyA" },
    { "read": "keyB" } ]
 ],
 "id": 0
}
```

← Scalaris sends results back

```
{"error": null,
 "result": {
  "results": [
   { "status": "ok", "value": {"type": "as_is", "value": "valueA"} },
   { "status": "ok", "value": {"type": "as_is", "value": "valueB"} }
  ],
  "tlog": <TLOG>
 },
 "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 →

```
{"jsonrpc": "2.0",
 "method": "req_list",
 "params": [
  <TLOG>,
  [ { "write": { "keyA": {"type": "as_is", "value": "valueA2"} } },
    { "commit": "" } ]
 ],
 "id": 0
}
```

← Scalaris sends results back

```
{"error": null,
 "result": {
  "results": [ {"status": "ok"}, {"status": "ok"} ],
  "tlog": <TLOG>
 },
 "id": 0
}
```

Examples of how to use the JSON API are the Python and Ruby API which use JSON to communicate with Scalaris.

4.1.4. Java API

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

- `TransactionSingleOp` provides methods for reading and writing values.
- `Transaction` provides methods for reading and writing values in transactions.
- `ReplicatedDHT` provides low-level methods for accessing the replicated DHT of Scalaris.

For details regarding the API we refer the reader to the Javadoc:

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

4.2. Command Line Interfaces

4.2.1. Java command line interface

As mentioned above, the `scalaris.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 --noconfig --help
```

```
../java-api/scalaris [script options] [options]
Script Options:
  --help, -h              print this message and scalaris help
  --noconfig              suppress sourcing of config files in $HOME/.scalaris/
                          and ${prefix}/etc/scalaris/
  --execdebug             print scalaris exec line generated by this
                          launch script
  --noerl                 do not ask erlang for its (local) host name

usage: scalaris [Options]
 -h,--help                                    print this message
 -v,--verbose                                 print verbose information,
                                              e.g. the properties read
 -lh,--localhost                              gets the local host's name as
                                              known to Java (for debugging
                                              purposes)
 -b,--minibench <[ops]> <[tpn]> <[benchs]>   run selected mini
                                              benchmark(s) [1|...|18|all]
                                              (default: all benchmarks, 500
                                              operations, 10 threads per
                                              Scalaris node)
 -m,--monitor <node>                          print monitoring information
 -r,--read <key>                              read an item
 -w,--write <key> <value>                     write an item
    --test-and-set <key> <old> <new>          atomic test and set, i.e.
                                              write <key> to <new> if the
                                              current value is <old>
 -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.
 -jmx,--jmxservice <node>                     starts a service exposing
                                              Scalaris monitoring values
                                              via JMX
```

`read`, `write`, `delete` and similar operations can be used to read, write and delete from/to the overlay, respectively. 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 management 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`.

4.2.2. Python command line interface

```
%> ./python-api/scalaris --help
```

```
usage: ../python-api/scalaris_client.py [Options]
 -r,--read <key>
                           read an item
 -w,--write <key> <value>
                           write an item
 --test-and-set <key> <old_value> <new_value>
                           atomic test and set, i.e. write <key> to
                           <new_value> if the current value is <old_value>
 -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 if an item is re-created, the version
                           before the delete can re-appear.
 -h,--help
                           print this message
 -b,--minibench [<ops> [<threads_per_node> [<benchmarks>]]]
                           run selected mini benchmark(s)
                           [1|...|9|all] (default: all benchmarks, 500
                           operations each, 10 threads per Scalaris node)
```

4.2.3. Ruby command line interface

```
%> ./ruby-api/scalaris --help
```

```
Usage: scalaris_client [options]
    -r, --read KEY                   read key KEY
    -w, --write KEY,VALUE            write key KEY to VALUE
        --test-and-set KEY,OLDVALUE,NEWVALUE
                                     write key KEY to NEWVALUE if the current value is OLDVALUE
        --add-del-on-list KEY,TOADD,TOREMOVE
                                     add and remove elements from the value at key KEY
        --add-on-nr KEY,VALUE        add VALUE to the value at key KEY
    -h, --help                       Show this message
```

4.3. Using Scalaris from Erlang

In this section, we will describe how to use Scalaris with two small examples. After having build Scalaris as described in 2, Scalaris can be run from the source directory directly.

4.3.1. Running a Scalaris Cluster

In this example, we will set up a simple Scalaris cluster consisting of up to five nodes running on a single computer.

Adapt the configuration. The first step is to adapt the configuration to your needs. We use the sample local configuration from 3.1, copy it to `bin/scalaris.local.cfg` and add a number of different known hosts. Note that the management server will run on the same port as the first node started in the example, hence we adapt its port as well.

```
{listen_ip , {127,0,0,1}}.
{mgmt_server , {{127,0,0,1},14195,mgmt_server}}.
{known_hosts , [{{127,0,0,1},14195, service_per_vm},
                {{127,0,0,1},14196, service_per_vm},
                {{127,0,0,1},14197, service_per_vm},
                {{127,0,0,1},14198, service_per_vm}
                % Although we will be using 5 nodes later, only 4 are added as known nodes.
            ]}.
```

Bootstrapping. In a shell (from now on called `S1`), start the first node ("premier"):

```
./bin/scalarisctl -m -n premier@127.0.0.1 -p 14195 -y 8000 -s -f start
```

The `-m` and `-f` options instruct `scalarisctl` to start the management server and the first_node (see Section 3.3 on page 18 for further details on `scalarisctl`). Note that the command above will produce some output about unknown nodes. This is expected, as some nodes defined in the configuration file above are not started yet.

After you run the above command and no further error occurred, you can query the locally available nodes using `scalarisctl`. Enter into a new shell (called `MS`):

```
./bin/scalarisctl list
epmd: up and running on port 4369 with data:
name premier at port 47235
```

Scalaris also contains a webserver. You can access it by pointing your browser to http://127.0.0.1:8000 (or the respective IP address of the node). With the above example, you can see the first node ("premier") and its management role.

Adding Nodes. We will now add four additional nodes to the cluster. Use a new shell (`S2` to `S5`) for each of the following commands. Each newly added node is a "real" Scalaris node and could run on another physical computer than the other nodes.

```
./bin/scalarisctl -n second@127.0.0.1 -p 14196 -y 8001 -s start
./bin/scalarisctl -n n3@127.0.0.1 -p 14197 -y 8002 -s start
./bin/scalarisctl -n n4@127.0.0.1 -p 14198 -y 8003 -s start
./bin/scalarisctl -n n5@127.0.0.1 -p 14199 -y 8004 -s start
```

Note that the last added nodes should not report a node as not reachable.

The management server should now report that the nodes have indeed joined Scalaris successfully. Query `scalarisctl`:

```
./bin/scalarisctl list
epmd: up and running on port 4369 with data:
name n5 at port 47801
name n4 at port 54614
name n3 at port 41710
name second at port 44329
name premier at port 44862
```

The actual output might differ, as the port numbers are assigned by the operating system.

Each node offers a web console. Point your browser to any url for http://127.0.0.1:8001 to http://127.0.0.1:8004. Observe that all nodes claim the cluster ring to consist of 5 nodes.

The web interface of node `premier` differs from the other interfaces. This is due to the fact that the management server is running on this node, adding additional information to the web interface.

Entering Data Using the Web Interface. A node's web interface can be used to query and enter data into Scalaris. To try this, point your browser to http://127.0.0.1:8000 (or any of the other nodes) and use the provided HTML form.

1. Lookup key `hello`. This will return `{fail,not_found}`
2. Add new keys `k1` and `k2` with values `v1` and `v2`, respectively. Then, lookup that key on the current and one of the other nodes. This should return `{ok,"v1"}` and `{ok, "v2"}` on both nodes.
3. Update the key `k1` by adding it on any node with value `v1updated`.
4. Update the key `k2` by adding it on any node with value `v2updated`. Lookup the key again and you should receive `{ok, v2updated}`

Simulating Node Failure. To simulate a node failure, we will simply stop `n4` using `scalarisctl`:

```
./bin/scalarisctl -n n4@127.0.0.1 stop
```

Other nodes will notice the crash of `n4`. By querying the available nodes in the shell MS again, you will now see only 4 nodes.

Although the node `n4` left the system, the data in the system is still consistent. Try to query the keys you added above. You should receive the values for each.

We will start a new node with the name `n4` again:

```
./bin/scalarisctl -n n4@127.0.0.1 -p 14198 -y 8003 -s start
```

The node list (again, query `scalarisctl` in shell MS) will report `n4` as alive again. You can still lookup the keys from above and should also receive the same result for the queries.

After running the above, we went from a five-node cluster to a 4-node cluster and back to a five-node cluster without any data loss due to a leaving node. The system was not unavailable for users and would have served any user requests without violating the data consistency or availability.

Controlling Scalaris Using the Erlang Shell. The calls to `scalarisctl` above which started a new Scalaris node ended within an Erlang shell. Each of those shells can be used to control a local Scalaris node and issue queries to the distributed database. Enter shell `S1` and hit <return> to see the Erlang shell prompt. Now, enter the following commands and check that the output is similar to the one provided here. You can stop the Erlang shell using `quit().`, which then also stops the corresponding Scalaris node.

```
(premier@127.0.0.1)1> api_tx:read("k0").
{fail,not_found}
(premier@127.0.0.1)2> api_tx:read("k1").
{ok,"v1updated"}
(premier@127.0.0.1)3> api_tx:read("k2").
{ok,"v2updated"}
(premier@127.0.0.1)4> api_tx:read(<<"k1">>).
{ok,"v1updated"}
(premier@127.0.0.1)5> api_tx:read(<<"k2">>).
{ok,"v2updated"}
(premier@127.0.0.1)6> api_tx:write(<<"k3">>,<<"v3">>).
{ok}
```

```
(premier@127.0.0.1)7> api_tx:read(<<"k3">>).
{ok,<<"v3">>}
(premier@127.0.0.1)8> api_tx:read("k3").
{ok,<<"v3">>}
(premier@127.0.0.1)9> api_tx:write(<<"k4">>,{1,2,3,four}).
{ok}
(premier@127.0.0.1)10> api_tx:read("k4").
{ok,{1,2,3,four}}
```

Attaching a Client to Scalaris. Now we will connect a true client to our 5 nodes Scalaris cluster. This client will not be a Scalaris node itself and thus represents a user application interacting with Scalaris.

We use a new shell to run an Erlang shell to do remote API calls to the server nodes.

```
erl -name client@127.0.0.1 -hidden -setcookie 'chocolate chip cookie'
```

The requests to Scalaris will be done using `rpc:call/4`. A production system would have some more sophisticated client side module, dispatching requests automatically to server nodes, for example.

```
(client@127.0.0.1)1> net_adm:ping('n3@127.0.0.1').
pong
(client@127.0.0.1)2> rpc:call('n3@127.0.0.1', api_tx, read, [<<"k0">>]).
{fail,not_found}
(client@127.0.0.1)3> rpc:call('n3@127.0.0.1', api_tx, read, [<<"k4">>]).
{ok,{1,2,3,four}}
(client@127.0.0.1)4> rpc:call('n4@127.0.0.1', api_tx, read, [<<"k4">>]).
{ok,{1,2,3,four}}
(client@127.0.0.1)5> rpc:call('n5@127.0.0.1', api_tx, write, [<<"num5">>,55]).
{ok}
(client@127.0.0.1)6> rpc:call('n3@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,55}
(client@127.0.0.1)7> rpc:call('n2@127.0.0.1', api_tx, add_on_nr, [<<"num5">>,2]).
{badrpc,nodedown}
(client@127.0.0.1)8> rpc:call('second@127.0.0.1', api_tx, add_on_nr, [<<"num5">>,2]).
{ok}
(client@127.0.0.1)9> rpc:call('n3@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,57}
(client@127.0.0.1)10> rpc:call('n4@127.0.0.1', api_tx, test_and_set, [<<"num5">>,57,59]).
{ok}
(client@127.0.0.1)11> rpc:call('n5@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,59}
(client@127.0.0.1)12> rpc:call('n4@127.0.0.1', api_tx, test_and_set, [<<"num5">>,57,55]).
{fail,{key_changed,59}}
(client@127.0.0.1)13> rpc:call('n3@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,59}
(client@127.0.0.1)14> rpc:call('n5@127.0.0.1', api_tx, test_and_set,
                               [<<"k2">>,"v2updated",<<"v2updatedTWICE">>]).
{ok}
(client@127.0.0.1)15> rpc:call('n4@127.0.0.1', api_tx, read, [<<"k2">>]).
{ok,<<"v2updatedTWICE">>}
(client@127.0.0.1)16> rpc:call('n3@127.0.0.1', api_tx, add_on_nr, [<<"num5">>,-4]).
{ok}
(client@127.0.0.1)17> rpc:call('n4@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,55}
(client@127.0.0.1)18> q().
ok
```

To show that the above calls actually worked with Scalaris, connect another client to the cluster and read updates made by the first:

```
erl -name clientagain@127.0.0.1 -hidden -setcookie 'chocolate chip cookie'
```

```
(clientagain@127.0.0.1)1> net_adm:ping('n5@127.0.0.1').
pong
(clientagain@127.0.0.1)2> rpc:call('n4@127.0.0.1', api_tx, read, [<<"k0">>]).
{fail,not_found}
(clientagain@127.0.0.1)3> rpc:call('n4@127.0.0.1', api_tx, read, [<<"k1">>]).
{ok,"v1updated"}
(clientagain@127.0.0.1)4> rpc:call('n3@127.0.0.1', api_tx, read, [<<"k2">>]).
{ok,<<"v2updatedTWICE">>}
(clientagain@127.0.0.1)5> rpc:call('second@127.0.0.1', api_tx, read, [<<"num5">>]).
{ok,55}
```

Shutting Down Scalaris. Firstly, we list the available nodes using `scalarisctl` using the shell `MS`.

```
./bin/scalarisctl list
epmd: up and running on port 4369 with data:
name n4 at port 52504
name n5 at port 47801
name n3 at port 41710
name second at port 44329
name premier at port 44862
```

Secondly, we shut down each of the nodes:

```
./bin/scalarisctl -n second@127.0.0.1 stop
'second@127.0.0.1'
./bin/scalarisctl -n n3@127.0.0.1 stop
'n3@127.0.0.1'
./bin/scalarisctl -n n4@127.0.0.1 stop
'n4@127.0.0.1'
./bin/scalarisctl -n n5@127.0.0.1 stop
'n5@127.0.0.1'
```

Only the first node remains:

```
./bin/scalarisctl list
epmd: up and running on port 4369 with data:
name premier at port 44862

./bin/scalarisctl -n premier@127.0.0.1 stop
'premier@127.0.0.1'
./bin/scalarisctl list
epmd: up and running on port 4369 with data:
(nothing)
```

The Scalaris API offers more transactional operations than just single-key read and write. The next part of this section will describe how to build transaction logs for atomic operations and how Scalaris handles conflicts in concurrently running transactions. See the module `api_tx` for more functions to access the data layer of Scalaris.

4.3.2. Transaction

In this section, we will describe how to build transactions using `api_tx:req_list(Tlog, List)` on the client side.

The setup is similar to the five nodes cluster in the previous section. To simplify the example all API calls are typed inside the Erlang shells of nodes `n4` and `n5`.

Consider two concurrent transactions `A` and `B`. `A` is a long-running operation, whereas `B` is only a short transaction. In the example, `A` starts before `B` and `B` ends before `A`. `B` is "timely" nested in `A` and disturbs `A`.

Single Read Operations. We first issue two read operations on nodes `n4, n5` to see that we are working on the same state for key `k1`:

```
(n4@127.0.0.1)10> api_tx:read(<<"k1">>).
{ok,<<"v1">>}
(n5@127.0.0.1)17> api_tx:read(<<"k1">>).
{ok,<<"v1">>}
```

Create Transaction Logs and Add Operations. Now, we create two transaction logs for the transactions and add the operations which are to be run atomically. `A` will be created on node `n5`, `B` on `n4`:

```
(n5@127.0.0.1)18> T5longA0 = api_tx:new_tlog().
[]
(n5@127.0.0.1)19> {T5longA1, R5longA1} = api_tx:req_list(T5longA0, [{read, <<"k1">>}]).
{[{76,<<"k1">>,1,75,'$empty'}],[{ok,<<"v1">>}]}
(n4@127.0.0.1)11> T4shortB0 = api_tx:new_tlog().
[]
(n4@127.0.0.1)12> {T4shortB1, R4shortB1} = api_tx:req_list(T4shortB0, [{read, <<"k1">>}]).
{[{76,<<"k1">>,1,75,'$empty'}],[{ok,<<"v1">>}]}
(n4@127.0.0.1)13> {T4shortB2, R4shortB2} = api_tx:req_list(T4shortB1,
                                            [{write, <<"k1">>, <<"v1Bshort">>}]).
{[{77,<<"k1">>,1,75, <<131,109,0,0,0,8,118,49,66,115,104,111,114,116>>}],
 [{ok}]}
(n4@127.0.0.1)14> {T4shortB3, R4shortB3} = api_tx:req_list(T4shortB2, [{read, <<"k1">>}]).
{[{77,<<"k1">>,1,75, <<131,109,0,0,0,8,118,49,66,115,104,111,114,116>>}],
 [{ok,<<"v1Bshort">>}]}
```

To finish the transaction log for `B`, we add `{commit}`. This operation should return an `ok`:

```
(n4@127.0.0.1)15> {T4shortB4, R4shortB4} = api_tx:req_list(T4shortB3, [{commit}]).
{[],[{ok}]}
(n4@127.0.0.1)16> [R4shortB1,R4shortB2,R4shortB3,R4shortB4].
[[{ok,<<"v1">>}],[{ok}],[{ok,<<"v1Bshort">>}],[{ok}]]
```

This concludes the creation of `B`. Now we will try to commit the long running transaction `A` after reading the key `k1` again. This and further attempts to write the key will fail, as the transaction `B` wrote this key since `A` started.

```
(n5@127.0.0.1)20> {T5longA2, R5longA2} = api_tx:req_list(T5longA1, [{read, <<"k1">>}]).
{[{76,<<"k1">>,2,{fail,abort},'$empty'}],
 [{ok,<<"v1Bshort">>}]}                                  % <-- SEE #### FAIL and ABORT ####
(n5@127.0.0.1)21> {T5longA3, R5longA3} = api_tx:req_list(T5longA2,
                                            [{write, <<"k1">>,<<"v1Along">>}]).
{[{76,<<"k1">>,2,{fail,abort},'$empty'}],[{ok}]}
(n5@127.0.0.1)22> {T5longA4, R5longA4} = api_tx:req_list(T5longA3, [{read, <<"k1">>}]).
{[{76,<<"k1">>,2,{fail,abort},'$empty'}],
 [{ok,<<"v1Bshort">>}]}
(n5@127.0.0.1)23> {T5longA5, R5longA5} = api_tx:req_list(T5longA4, [{commit}]).
{[],[{fail,abort,[<<"k1">>]}]}                           % <-- SEE #### FAIL and ABORT ####
(n4@127.0.0.1)17> api_tx:read(<<"k1">>).
{ok,<<"v1Bshort">>}
(n5@127.0.0.1)24> api_tx:read(<<"k1">>).
{ok,<<"v1Bshort">>}
```

As expected, the first coherent commit `B` constructed on `n4` has won.

Note that in a real system, operations in `api_tx:req_list(Tlog, List)` should be grouped together with a trailing `{commit}` as far as possible. The individual separation of all reads, writes and commits was done here on purpose to study the transactional behaviour.

5. Testing the system

5.1. Erlang unit tests

There are some unit tests in the `test` directory which test Scalaris itself (the Erlang code). 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 10-20 minutes, depending on your machine.

If the test suite is interrupted before finishing, the results may not have been linked into the `index.html` file. They are however stored in the `ct_run.ct@...` directory.

5.2. Java unit tests

The Java unit tests can be run by executing `make java-test` in the main directory. This will start a Scalaris node with the default ports and test all functions of the Java API. A typical run will look like the following:

```
%> make java-test
[...]
tools.test:
    [junit] Running de.zib.tools.PropertyLoaderTest
    [junit] Testsuite: de.zib.tools.PropertyLoaderTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 sec
    [junit] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] Working Directory = <scalarisdir>/java-api/classes
    [junit] ------------- ---------------- ---------------
[...]
scalaris.test:
    [junit] Running de.zib.scalaris.ConnectionTest
    [junit] Testsuite: de.zib.scalaris.ConnectionTest
    [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.303 sec
    [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.303 sec
    [junit]
    [junit] Running de.zib.scalaris.DefaultConnectionPolicyTest
    [junit] Testsuite: de.zib.scalaris.DefaultConnectionPolicyTest
    [junit] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.309 sec
    [junit] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.309 sec
    [junit]
    [junit] Running de.zib.scalaris.ErlangValueTest
    [junit] Testsuite: de.zib.scalaris.ErlangValueTest
```

```
    [junit] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.444 sec
    [junit] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.444 sec
    [junit]
    [junit] Running de.zib.scalaris.MonitorTest
    [junit] Testsuite: de.zib.scalaris.MonitorTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 sec
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 sec
    [junit]
    [junit] Running de.zib.scalaris.PeerNodeTest
    [junit] Testsuite: de.zib.scalaris.PeerNodeTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.066 sec
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.066 sec
    [junit]
    [junit] Running de.zib.scalaris.ReplicatedDHTTest
    [junit] Testsuite: de.zib.scalaris.ReplicatedDHTTest
    [junit] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.723 sec
    [junit] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.723 sec
    [junit]
    [junit] Running de.zib.scalaris.ScalarisTest
    [junit] Testsuite: de.zib.scalaris.ScalarisTest
    [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.063 sec
    [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.063 sec
    [junit]
    [junit] Running de.zib.scalaris.ScalarisVMTest
    [junit] Testsuite: de.zib.scalaris.ScalarisVMTest
    [junit] Tests run: 42, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 0.699 sec
    [junit] Tests run: 42, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 0.699 sec
    [junit]
    [junit] Testcase: testKillVM1(de.zib.scalaris.ScalarisVMTest):SKIPPED: we still need the Scalaris
    [junit] Testcase: testShutdownVM1(de.zib.scalaris.ScalarisVMTest):SKIPPED: we still need the Scal
    [junit] Running de.zib.scalaris.TransactionSingleOpTest
    [junit] Testsuite: de.zib.scalaris.TransactionSingleOpTest
    [junit] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.996 sec
    [junit] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.996 sec
    [junit]
    [junit] Running de.zib.scalaris.TransactionTest
    [junit] Testsuite: de.zib.scalaris.TransactionTest
    [junit] Tests run: 30, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.803 sec
    [junit] Tests run: 30, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.803 sec
    [junit]

test:

BUILD SUCCESSFUL
Total time: 27 seconds
'jtest_boot@csr-pc40.zib.de'
```

5.3. Python2 unit tests

The Python unit tests can be run by executing `make python-test` in the main directory. This will start a Scalaris node with the default ports and test all functions of the Python API. A typical run will look like the following:

```
%> make python-test
[...]
testDelete1 (__main__.TestReplicatedDHT) ... ok
testDelete2 (__main__.TestReplicatedDHT) ... ok
testDelete_notExistingKey (__main__.TestReplicatedDHT) ... ok
testDoubleClose (__main__.TestReplicatedDHT) ... ok
testReplicatedDHT1 (__main__.TestReplicatedDHT) ... ok
testReplicatedDHT2 (__main__.TestReplicatedDHT) ... ok
testAddNodes0 (__main__.TestScalarisVM)
Test method for ScalarisVM.addNodes(0). ... ok
testAddNodes1 (__main__.TestScalarisVM)
Test method for ScalarisVM.addNodes(1). ... ok
testAddNodes3 (__main__.TestScalarisVM)
```

```
Test method for ScalarisVM.addNodes(3). ... ok
testAddNodes_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.addNodes() with a closed connection. ... ok
testDoubleClose (__main__.TestScalarisVM) ... ok
testGetInfo1 (__main__.TestScalarisVM)
Test method for ScalarisVM.getInfo(). ... ok
testGetInfo_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.getInfo() with a closed connection. ... ok
testGetNodes1 (__main__.TestScalarisVM)
Test method for ScalarisVM.getNodes(). ... ok
testGetNodes_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.getNodes() with a closed connection. ... ok
testGetNumberOfNodes1 (__main__.TestScalarisVM)
Test method for ScalarisVM.getVersion(). ... ok
testGetNumberOfNodes_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.getNumberOfNodes() with a closed connection. ... ok
testGetOtherVMs1 (__main__.TestScalarisVM)
Test method for ScalarisVM.getOtherVMs(1). ... ok
testGetOtherVMs2 (__main__.TestScalarisVM)
Test method for ScalarisVM.getOtherVMs(2). ... ok
testGetOtherVMs3 (__main__.TestScalarisVM)
Test method for ScalarisVM.getOtherVMs(3). ... ok
testGetOtherVMs_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.getOtherVMs() with a closed connection. ... ok
testGetVersion1 (__main__.TestScalarisVM)
Test method for ScalarisVM.getVersion(). ... ok
testGetVersion_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.getVersion() with a closed connection. ... ok
testKillNode1 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNode(). ... ok
testKillNode_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.killNode() with a closed connection. ... ok
testKillNodes0 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodes(0). ... ok
testKillNodes1 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodes(1). ... ok
testKillNodes3 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodes(3). ... ok
testKillNodesByName0 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodesByName(0). ... ok
testKillNodesByName1 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodesByName(1). ... ok
testKillNodesByName3 (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodesByName(3). ... ok
testKillNodesByName_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodesByName() with a closed connection. ... ok
testKillNodes_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.killNodes() with a closed connection. ... ok
testScalarisVM1 (__main__.TestScalarisVM) ... ok
testScalarisVM2 (__main__.TestScalarisVM) ... ok
testShutdownNode1 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNode(). ... ok
testShutdownNode_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNode() with a closed connection. ... ok
testShutdownNodes0 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodes(0). ... ok
testShutdownNodes1 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodes(1). ... ok
testShutdownNodes3 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodes(3). ... ok
testShutdownNodesByName0 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodesByName(0). ... ok
testShutdownNodesByName1 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodesByName(1). ... ok
testShutdownNodesByName3 (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodesByName(3). ... ok
testShutdownNodesByName_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodesByName() with a closed connection. ... ok
testShutdownNodes_NotConnected (__main__.TestScalarisVM)
Test method for ScalarisVM.shutdownNodes() with a closed connection. ... ok
testAbort_Empty (__main__.TestTransaction) ... ok
```

```
testAbort_NotConnected (__main__.TestTransaction) ... ok
testCommit_Empty (__main__.TestTransaction) ... ok
testCommit_NotConnected (__main__.TestTransaction) ... ok
testDoubleClose (__main__.TestTransaction) ... ok
testRead_NotConnected (__main__.TestTransaction) ... ok
testRead_NotFound (__main__.TestTransaction) ... ok
testReqList1 (__main__.TestTransaction) ... ok
testReqList_Empty (__main__.TestTransaction) ... ok
testReqTooLarge (__main__.TestTransaction) ... ok
testTransaction1 (__main__.TestTransaction) ... ok
testTransaction3 (__main__.TestTransaction) ... ok
testVarious (__main__.TestTransaction) ... ok
testWriteList1 (__main__.TestTransaction) ... ok
testWriteString (__main__.TestTransaction) ... ok
testWriteString_NotConnected (__main__.TestTransaction) ... ok
testWriteString_NotFound (__main__.TestTransaction) ... ok
testDoubleClose (__main__.TestTransactionSingleOp) ... ok
testRead_NotConnected (__main__.TestTransactionSingleOp) ... ok
testRead_NotFound (__main__.TestTransactionSingleOp) ... ok
testReqList1 (__main__.TestTransactionSingleOp) ... ok
testReqList_Empty (__main__.TestTransactionSingleOp) ... ok
testReqTooLarge (__main__.TestTransactionSingleOp) ... ok
testTestAndSetList1 (__main__.TestTransactionSingleOp) ... ok
testTestAndSetList2 (__main__.TestTransactionSingleOp) ... ok
testTestAndSetList_NotConnected (__main__.TestTransactionSingleOp) ... ok
testTestAndSetList_NotFound (__main__.TestTransactionSingleOp) ... ok
testTestAndSetString1 (__main__.TestTransactionSingleOp) ... ok
testTestAndSetString2 (__main__.TestTransactionSingleOp) ... ok
testTestAndSetString_NotConnected (__main__.TestTransactionSingleOp) ... ok
testTestAndSetString_NotFound (__main__.TestTransactionSingleOp) ... ok
testTransactionSingleOp1 (__main__.TestTransactionSingleOp) ... ok
testTransactionSingleOp2 (__main__.TestTransactionSingleOp) ... ok
testWriteList1 (__main__.TestTransactionSingleOp) ... ok
testWriteList2 (__main__.TestTransactionSingleOp) ... ok
testWriteList_NotConnected (__main__.TestTransactionSingleOp) ... ok
testWriteString1 (__main__.TestTransactionSingleOp) ... ok
testWriteString2 (__main__.TestTransactionSingleOp) ... ok
testWriteString_NotConnected (__main__.TestTransactionSingleOp) ... ok

----------------------------------------------------------------------
Ran 84 tests in 3.565s

OK
'jtest_boot@csr-pc40.zib.de'
```

5.4. Python3 unit tests

The Python 3 tests are similar to the Python 2 tests above and can be run by executing `make python3-test`.

5.5. Ruby unit tests

The Ruby unit tests can be run by executing `make ruby-test` in the main directory. This will start a Scalaris node with the default ports and test all functions of the Ruby API. A typical run will look like the following:

```
%> make ruby-test
[...]
# Running tests:

TestReplicatedDHT#testDelete1 = 0.19 s = .
TestReplicatedDHT#testDelete2 = 0.29 s = .
TestReplicatedDHT#testDelete_notExistingKey = 0.05 s = .
```

```
TestReplicatedDHT#testDoubleClose = 0.00 s = .
TestReplicatedDHT#testReplicatedDHT1 = 0.00 s = .
TestReplicatedDHT#testReplicatedDHT2 = 0.00 s = .
TestTransaction#testAbort_Empty = 0.00 s = .
TestTransaction#testAbort_NotConnected = 0.00 s = .
TestTransaction#testCommit_Empty = 0.00 s = .
TestTransaction#testCommit_NotConnected = 0.00 s = .
TestTransaction#testDoubleClose = 0.00 s = .
TestTransaction#testRead_NotConnected = 0.00 s = .
TestTransaction#testRead_NotFound = 0.00 s = .
TestTransaction#testReqList1 = 0.02 s = .
TestTransaction#testReqList_Empty = 0.00 s = .
TestTransaction#testReqTooLarge = 0.38 s = .
TestTransaction#testTransaction1 = 0.00 s = .
TestTransaction#testTransaction3 = 0.00 s = .
TestTransaction#testVarious = 0.01 s = .
TestTransaction#testWriteList1 = 0.08 s = .
TestTransaction#testWriteString = 0.11 s = .
TestTransaction#testWriteString_NotConnected = 0.00 s = .
TestTransaction#testWriteString_NotFound = 0.00 s = .
TestTransactionSingleOp#testDoubleClose = 0.00 s = .
TestTransactionSingleOp#testRead_NotConnected = 0.00 s = .
TestTransactionSingleOp#testRead_NotFound = 0.00 s = .
TestTransactionSingleOp#testReqList1 = 0.03 s = .
TestTransactionSingleOp#testReqList_Empty = 0.00 s = .
TestTransactionSingleOp#testReqTooLarge = 0.38 s = .
TestTransactionSingleOp#testTestAndSetList1 = 0.07 s = .
TestTransactionSingleOp#testTestAndSetList2 = 0.05 s = .
TestTransactionSingleOp#testTestAndSetList_NotConnected = 0.00 s = .
TestTransactionSingleOp#testTestAndSetList_NotFound = 0.00 s = .
TestTransactionSingleOp#testTestAndSetString1 = 0.06 s = .
TestTransactionSingleOp#testTestAndSetString2 = 0.08 s = .
TestTransactionSingleOp#testTestAndSetString_NotConnected = 0.00 s = .
TestTransactionSingleOp#testTestAndSetString_NotFound = 0.00 s = .
TestTransactionSingleOp#testTransactionSingleOp1 = 0.00 s = .
TestTransactionSingleOp#testTransactionSingleOp2 = 0.00 s = .
TestTransactionSingleOp#testWriteList1 = 0.06 s = .
TestTransactionSingleOp#testWriteList2 = 0.02 s = .
TestTransactionSingleOp#testWriteList_NotConnected = 0.00 s = .
TestTransactionSingleOp#testWriteString1 = 0.08 s = .
TestTransactionSingleOp#testWriteString2 = 0.05 s = .
TestTransactionSingleOp#testWriteString_NotConnected = 0.00 s = .


Finished tests in 2.040348s, 22.0551 tests/s, 675.8650 assertions/s.

45 tests, 1379 assertions, 0 failures, 0 errors, 0 skips

ruby -v: ruby 2.1.3p242 (2014-09-19 revision 47630) [x86_64-linux-gnu]
'jtest_boot@csr-pc40.zib.de'
```

5.6. Interoperability Tests

In order to check whether the common types described in Section 4.1 on page 19 are fully supported by the APIs and yield to the appropriate types in another API, we implemented some interoperability tests. Two make targets exist:

- `make interop-test` verifies compliance in Java, Python2 and Ruby,
- `make interop3-test` verifies compliance in Java, Python2, Python3 and Ruby.

This will start a Scalaris node with the default ports, write test data using the mentioned APIs and let each API read the data it wrote itself as well as the data the other APIs wrote. On success it will print

```
%> make interop3-test
[...]
all tests successful
```

6. Troubleshooting

6.1. Network

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

	HTTP Server	Inter-node communication
default (see `bin/scalaris.cfg`)	8000	14195–14198
first node (`bin/firstnode.sh`)	8000	14195
joining node 1 (`bin/joining_node.sh`)	8001	14196
other joining nodes (`bin/joining_node.sh <ID>`)	8000 + `<ID>`	14195 + `<ID>`
standalone mgmt server (`bin/mgmt-server.sh`)	7999	14194

Please make sure that at least 14195 and 14196 are not blocked by firewalls in order to be able to start at least one first and one joining node on each machine..

6.2. Miscellaneous

For up-to-date information about frequently asked questions and troubleshooting, please refer to our FAQs at http://scalaris.zib.de/faq.html and our mailing list at http://groups.google.com/group/scalaris.

Part II.

Developers Guide

7. General Hints

7.1. 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.

7.2. 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/firstnode.sh` and run the tests by `cd java; ant test`)
- Run the Ruby client by starting Scalaris and running `cd contrib; ./jsonrpc.rb`

7.3. 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
- Trace messages using the `trace_mpath` module
- 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 `admin:number_of_nodes()`.
- use `admin:check_ring()`.

8. System Infrastructure

8.1. 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)

8.2. The Communication Layer `comm`

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

8.3. The `gen_component`

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).

8.3.1. 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:

```
-ifdef(have_callback_support).
-callback init(Args::term()) -> user_state().
-else.
-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, UserState) -> NewUserState | unknown_event | kill
    ];
behaviour_info(_Other) -> undefined.
-endif.
```

This is illustrated by the following example:

File msg_delay.erl:

```
%% initialize: return initial state.
-spec init([]) -> state().
init([]) ->
    ?TRACE("msg_delay:init for pid group ~p~n", [pid_groups:my_groupname()]),
    %% For easier debugging, use a named table (generates an atom)
    %%TableName = erlang:list_to_atom(pid_groups:my_groupname() ++ "_msg_delay"),
    %%TimeTable = pdb:new(TableName, [set, protected, named_table]),
    %% use random table name provided by ets to *not* generate an atom
    TimeTable = pdb:new(?MODULE, [set]),
    comm:send_local(self(), {msg_delay_periodic}),
    _State = {TimeTable, _Round = 0}.

-spec on(message(), state()) -> state().
on({msg_delay_req, Seconds, Dest, Msg, Options} = _FullMsg,
   {TimeTable, Counter} = State) ->
    ?TRACE("msg_delay:on(~.0p, ~.0p)~n", [_FullMsg, State]),
    Future = trunc(Counter + Seconds),
    EMsg = case erlang:get(trace_mpath) of
               undefined -> Msg;
               PState -> trace_mpath:epidemic_reply_msg(PState, comm:this(), Dest, Msg)
           end,
    case pdb:get(Future, TimeTable) of
        undefined ->
            pdb:set({Future, [{Dest, EMsg, Options}]}, TimeTable);
        {_, MsgQueue} ->
            pdb:set({Future, [{Dest, EMsg, Options} | MsgQueue]}, TimeTable)
    end,
    State;

%% periodic trigger
on({msg_delay_periodic} = Trigger, {TimeTable, Counter} = _State) ->
    ?TRACE("msg_delay:on(~.0p, ~.0p)~n", [Trigger, State]),
    % triggers are not allowed to be infected!
    ?DBG_ASSERT2(not trace_mpath:infected(), trigger_infected),
    _ = case pdb:take(Counter, TimeTable) of
        undefined -> ok;
        {_, MsgQueue} ->
```

```
            _ = [ case Msg of
                      {'$gen_component', trace_mpath, PState, _From, _To, OrigMsg} ->
                          case element(2, PState) of %% element 2 is the logger
                              {proto_sched, _} ->
                                  log:log("msg_delay: proto_sched not ready for delayed messages, so
                                  %% these messages should not be
                                  %% accepted to the database of
                                  %% msg_delay anyway, but instead
                                  %% should be immediately delivered
                                  %% to proto_sched for the 'delayed'
                                  %% messages pool (which is not
                                  %% implemented yet) (see send_local,
                                  %% send_local_as_client)
                                  %% erlang:throw(redirect_proto_sched_msgs_at_submission_please)
                                  ok;
                              _ ->
                                  trace_mpath:start(PState),
                                  comm:send_local(Dest, OrigMsg, Options),
                                  trace_mpath:stop()
                          end;
                      _ ->
                          ?DBG_ASSERT2(not trace_mpath:infected(), infected_with_uninfected_msg),
                          comm:send_local(Dest, Msg, Options)
                  end || {Dest, Msg, Options} <- MsgQueue ]
    end,
    _ = comm:send_local_after(1000, self(), Trigger),
    {TimeTable, Counter + 1};

on({web_debug_info, Requestor}, {TimeTable, Counter} = State) ->
    KeyValueList =
        [{"queued messages (in 0-10s, messages):", ""} |
         [begin
              Future = trunc(Counter + Seconds),
              Queue = case pdb:get(Future, TimeTable) of
                          undefined -> none;
                          {_, Q}     -> Q
                      end,
              {webhelpers:safe_html_string("~p", [Seconds]),
               webhelpers:safe_html_string("~p", [Queue])}
          end || Seconds <- lists:seq(0, 10)]],
    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 given to gen_component:start_link/4 as well as gen_component:start/4 or gen_component:start/5. It can be changed by calling gen_component:change_handler/2 (see Section 8.3.7). 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.

8.3.2. How to start a gen_component?

A gen_component can be started using one of:

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

`Handler`: the inital message handler
`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}`.

8.3.3. 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.

8.3.4. How to determine whether a process is a `gen_component`?

A `gen_component` can be detected by:

`gen_component:is_gen_component(Pid)`, which returns a boolean.

8.3.5. 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 8.1) 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.

8.3.6. 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.

8.3.7. 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.

8.3.8. Handling several messages atomically

The message handler is called for each message separately. Such a single call is atomic, i.e. the component does not perform any other action until the called message handler finishes. Sometimes, it is necessary to execute two or more calls to the message handler atomically (without other interleaving messages). For example if a message A contains another message B as payload, it may be necessary to handle A and B directly one after the other without interference of other messages. So, after handling A you want to call your message handler with B.

In most cases, you could just do so by calculating the new state as result of handling message A first and then calling the message handler with message B and the new state by yourself.

It is safer to use gen_component:post_op(2) in such cases: When B contains a special message, which is usually handled by the gen_component module itself (like send_to_group_member, kill, sleep), the direct call to the message handler would not achieve the expected result. By calling gen_component:post_op(B, NewState) to return the new state after handling message A, message B will be handled directly after the current message A.

8.3.9. Halting and pausing a gen_component

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

8.3.10. Integration with pid_groups: Redirecting 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 8.1) 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.

8.3.11. 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.

8.3.12. 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 8.2 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(1..4, 4..16, {pos_integer(), pos_integer(), pos_integer()})
        -> true.
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(comm:make_local(X), ?proposer_initialize, bp)
            ||  X <- Proposers],
    _ = [ gen_component:bp_set(comm:make_local(X), acceptor_initialize, bp)
            ||  X <- Acceptors ],
    _ = [ gen_component:bp_set(comm:make_local(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,
    _ = [ gen_component:kill(comm:make_local(X))
          || X <- lists:flatten([Proposers, Acceptors, Learners])],
    true.

step_until_decide(Processes, PaxId, SumSteps) ->
    %% io:format("Step ~p~n", [SumSteps]),
    Runnable = [ X || X <- Processes, gen_component:runnable(comm:make_local(X)) ],
    case Runnable of
        [] ->
            ct:pal("No runnable processes of ~p~n", [length(Processes)]),
            timer:sleep(5), step_until_decide(Processes, PaxId, SumSteps);
        _ ->
            Num = random:uniform(length(Runnable)),
            _ = gen_component:bp_step(comm:make_local(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
    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), log:pal(X,Y)).         %% output even if called by unittest
%-define(TRACE_BP_STEPS(X,Y), io:format(user,X,Y)). %% clean output even if called by unittest
-define(TRACE_BP_STEPS(X,Y), ok).
```

8.3.13. 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.

8.4. The Process' Database (pdb)

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

8.5. Failure Detectors (fd)

- uses Erlang monitors locally
- is independent of component load
- uses heartbeats between Erlang virtual machines
- uses a single proxy heartbeat server per Erlang virtual machine, which itself uses Erlang monitors to monitor locally

- uses dynamic timeouts to implement an eventually perfect failure detector.

8.6. Monitoring Statistics (`monitor`, `rrd`)

The `monitor` module offers several methods to gather meaningful statistics using the `rrd()` data type defined in `rrd`.

`rrd()` records work with time slots, i.e. a fixed slot length is given at creation and items which should be inserted will be either put into the current slot, or a new slot will be created. Each data item thus needs a time stamp associated with it. It must not be a real time, but can also be a virtual time stamp.

The `rrd` module thus offers two different APIs: one with transparent time handling, e.g. `rrd:-create/3`, `rrd:add_now/2`, and one with manual time handling, e.g. `rrd:create/4`, `rrd:add/3`.

To allow different evaluations of the stored data, the following types of data are supported:

- `gauge`: only stores the newest value of a time slot, e.g. for thermometers,
- `counter`: sums up all values inside a time slot,
- `timing`: records time spans and stores values to easily calculate e.g. the sum, the standard deviation, the number of events, the min and max,
- `timing_with_hist`: similar to `timing` but also records a more detailed (approximated) histogram of the data,
- `event`: records each event (including its time stamp) inside a time slot in a list (this should be rarely used as the amount of data stored may be very big).
- `histogram, N`: records values in an approximative histogram of size N
- `histogram_rt, N, BaseKey`: histogram of size N which operates on the key space of the DHT. BaseKey is the key with the largest distance to all keys in the histogram.

The `monitor` offers functions to conveniently store and retrieve such values. It is also started as a process in each `dht_node` and `basic_services` group as well as inside each `clients_group`. This process ultimately stores the whole rrd() structure. There are three paradigms how values can be stored:

1. Values are gathered in the process that is generating the values. Inside this process, the `rrd()` is stored in the erlang dictionary. Whenever a new time slot is started, the values will be reported to the monitor process of the gathering process' group.
2. Values are gathered in the process that is generating the values. Inside this process, the `rrd()` is handled manually. After changing the `rrd()`, a manual check for reporting needs to be issued using `monitor:check_report/4`.
3. Values are immediately send to the monitor process where it undergoes the same procedures until it is finally stored and available to other processes. This is especially useful if the process generating the values does not live long or does not regularly create new data, e.g. the client.

The following example illustrates the first mode, i.e. gathering data in the generating process. It has been taken from the `cyclon` module which uses a `counter` data type:

```
% initialise the monitor with an empty rrd() using a 60s monitoring interval
monitor:proc_set_value(?MODULE, 'shuffle', rrd:create(60 * 1000000, 3, counter)),
% update the value by adding one
monitor:proc_set_value(?MODULE, 'shuffle', fun(Old) -> rrd:add_now(1, Old) end),
% check regularly whether to report the data to the monitor:
```

```
monitor:proc_check_timeslot(?MODULE, 'shuffle')
```

The first two parameters of monitor:proc_set_value/3 define the name of a monitored value, the module's name and a unique key. The second can be either an rrd() or an update fun. The monitor:proc_check_timeslot/3 function can be used if your module does not regularly create new data. In this case, the monitor process would not have the latest data for others to retrieve. This function forces a check and creates the new time slot if needed (thus reporting the data).

This is how forwarding works (taken from api_tx):

```
monitor:client_monitor_set_value(
  ?MODULE, 'req_list',
  fun(Old) ->
    Old2 = case Old of
             % 10s monitoring interval, only keep newest in the client process
             undefined -> rrd:create(10 * 1000000, 1, {timing, ms});
             _ -> Old
           end,
    rrd:add_now(TimeInUs / 1000, Old2)
  end),
```

As in this case there is no safe way of initialising the value, it is more useful to provide an update fun to monitor:client_monitor_set_value/3. This function is only useful for the client processes as it reports to the monitor in the clients_group (recall that client processes do not belong to any group). All other processes should use monitor:monitor_set_value/3 with the same semantics.

8.7. Writing Unit Tests

8.7.1. Plain Unit Tests

8.7.2. Randomized Testing Using `tester`

8.7.3. Randomized Testing Using `proto_sched`

9. Basic Structured Overlay

9.1. Ring Maintenance

9.2. T-Man

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

File api_dht_raw.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(), noid, 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 config:read(leases) of
        true ->
            lookup_aux_leases(State, Key, Hops, Msg);
        _ ->
            lookup_aux_chord(State, Key, Hops, Msg)
end.

-spec lookup_aux_chord(State::dht_node_state:state(), Key::intervals:key(),
                       Hops::non_neg_integer(), Msg::comm:message()) -> ok.
lookup_aux_chord(State, Key, Hops, Msg) ->
    WrappedMsg = ?RT:wrap_message(Key, Msg, State, Hops),
```

```
    case ?RT:next_hop(State, Key) of
        {succ, P} -> % found node -> terminate
            comm:send(P, {?lookup_fin, Key, ?HOPS_TO_DATA(Hops + 1), WrappedMsg}, [{shepherd, self()}
        {other, P} ->
            comm:send(P, {?lookup_aux, Key, Hops + 1, WrappedMsg}, [{shepherd, self()}])
    end.

-spec lookup_aux_leases(State::dht_node_state:state(), Key::intervals:key(),
                        Hops::non_neg_integer(), Msg::comm:message()) -> ok.
lookup_aux_leases(State, Key, Hops, Msg) ->
    WrappedMsg = ?RT:wrap_message(Key, Msg, State, Hops),
    case leases:is_responsible(State, Key) of
        true ->
            comm:send_local(dht_node_state:get(State, monitor_proc),
                            {lookup_hops, Hops}),
            DHTNode = pid_groups:find_a(dht_node),
            %log:log("aux -> fin: ~p ~p~n", [self(), DHTNode]),
            comm:send_local(DHTNode,
                            {?lookup_fin, Key, ?HOPS_TO_DATA(Hops + 1), WrappedMsg});
        maybe ->
            DHTNode = pid_groups:find_a(dht_node),
            %log:log("aux -> fin: ~p ~p~n", [self(), DHTNode]),
            comm:send_local(DHTNode,
                            {?lookup_fin, Key, ?HOPS_TO_DATA(Hops + 1), WrappedMsg});
        false ->
            MyRange = dht_node_state:get(State, my_range),
            case intervals:in(Key, MyRange) of
                true ->
                    % @doc we are here because leases and rm disagree
                    % over responsibility. One cause for this case can
                    % be join/sliding. Our successor still has the
                    % lease for our range. But rm already believes
                    % that we are responsible for our range. The
                    % solution is to forward the lookup to our
                    % successor instead of asking rt.
                    Succ = node:pidX(dht_node_state:get(State, succ)),
                    comm:send(Succ, {?lookup_aux, Key, Hops + 1, WrappedMsg}, [{shepherd, self()}]);
                false ->
                    P = element(2, ?RT:next_hop(State, Key)),
                    comm:send(P, {?lookup_aux, Key, Hops + 1, WrappedMsg}, [{shepherd, self()}])
            end
    end.

%% @doc Find the node responsible for Key and send him the message Msg.
-spec lookup_fin(State::dht_node_state:state(), Key::intervals:key(),
                 Data::data(), Msg::comm:message()) -> dht_node_state:state().
lookup_fin(State, Key, Hops, Msg) ->
    case config:read(leases) of
        true ->
            lookup_fin_leases(State, Key, Hops, Msg);
        _ ->
            lookup_fin_chord(State, Key, Hops, Msg)
    end.

-spec lookup_fin_chord(State::dht_node_state:state(), Key::intervals:key(),
                       Data::data(), Msg::comm:message()) -> dht_node_state:state().
lookup_fin_chord(State, Key, Data, Msg) ->
    MsgFwd = dht_node_state:get(State, msg_fwd),
    FwdList = [P || {I, P} <- MsgFwd, intervals:in(Key, I)],
    Hops = ?HOPS_FROM_DATA(Data),
    case FwdList of
        []    ->
            case dht_node_state:is_db_responsible__no_msg_fwd_check(Key, State) of
                true ->
                    %comm:send_local(dht_node_state:get(State, monitor_proc),
                    %                {lookup_hops, Hops}),
                    %Unwrap = ?RT:unwrap_message(Msg, State),
                    %gen_component:post_op(Unwrap, State);
                    deliver(State, Msg, false, Hops);
                false ->
                    % do not warn if
```

```
                    % a) received lookup_fin due to a msg_fwd while sliding and
                    %    before the other node removed the message forward or
                    % b) our pred is not be aware of our ID change yet (after
                    %    moving data to our successor) yet
                    SlidePred = dht_node_state:get(State, slide_pred),
                    SlideSucc = dht_node_state:get(State, slide_succ),
                    Neighbors = dht_node_state:get(State, neighbors),
                    InSlideIntervalFun =
                        fun(SlideOp) ->
                                slide_op:is_slide(SlideOp) andalso
                                    slide_op:get_sendORreceive(SlideOp) =:= 'send' andalso
                                    intervals:in(Key, slide_op:get_interval(SlideOp))
                        end,
                    case lists:any(InSlideIntervalFun, [SlidePred, SlideSucc]) orelse
                             intervals:in(Key, nodelist:succ_range(Neighbors)) of
                        true -> ok;
                        false ->
                            DBRange = dht_node_state:get(State, db_range),
                            DBRange2 = [begin
                                            case intervals:is_continuous(Interval) of
                                                true -> {intervals:get_bounds(Interval), Id};
                                                _     -> {Interval, Id}
                                            end
                                        end || {Interval, Id} <- DBRange],
                            log:log(warn,
                                    "[ ~.0p ] Routing is damaged (~p)!! Trying again...~n"
                                    "  myrange:~p~n  db_range:~p~n  msgfwd:~p~n  Key:~p~n"
                                    "  pred: ~.4p~n  node: ~.4p~n  succ: ~.4p",
                                    [self(), Data, intervals:get_bounds(nodelist:node_range(Neighbors
                                     DBRange2, MsgFwd, Key, nodelist:pred(Neighbors),
                                     nodelist:node(Neighbors), nodelist:succ(Neighbors)])
                    end,
                    lookup_aux(State, Key, Hops, Msg),
                    State
            end;
        [Pid] -> comm:send(Pid, {?lookup_fin, Key, ?HOPS_TO_DATA(Hops + 1), Msg}),
                 State
    end.

-spec lookup_fin_leases(State::dht_node_state:state(), Key::intervals:key(),
                        Data::data(), Msg::comm:message()) -> dht_node_state:state().
lookup_fin_leases(State, Key, Data, Msg) ->
    Hops = ?HOPS_FROM_DATA(Data),
    case leases:is_responsible(State, Key) of
        true ->
            deliver(State, Msg, true, Hops);
        maybe ->
            deliver(State, Msg, false, Hops);
        false ->
            log:log("lookup_fin fail: ~p", [self()]),
            lookup_aux(State, Key, Hops, Msg),
            State
    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`:

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

%%Standard Chord routingtable
-define(RT, rt_chord).
% first valid key:
-define(MINUS_INFINITY, 0).
-define(MINUS_INFINITY_TYPE, 0).
% first invalid key:
-define(PLUS_INFINITY, 16#100000000000000000000000000000000).
-define(PLUS_INFINITY_TYPE, 16#100000000000000000000000000000000).

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

%% Flexible Routing Tables
%% Standard flexible routingtable
%-define(RT, rt_frtchord).
%% Grouped Flexible Routing Table
%-define(RT, rt_gfrtchord).
```

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

File `rt_beh.erl`:

```erlang
-ifdef(have_callback_support).
-include("scalaris.hrl").
-include("client_types.hrl").
-type rt() :: term().
-type external_rt() :: term().
-type key() :: term().

-callback empty_ext(nodelist:neighborhood()) -> external_rt().
-callback init(nodelist:neighborhood()) -> rt().
-callback hash_key(client_key() | binary()) -> key().
-callback get_random_node_id() -> key().
-callback next_hop(dht_node_state:state(), key()) -> {succ | other, comm:mypid()}.

-callback init_stabilize(nodelist:neighborhood(), rt()) -> rt().
-callback update(OldRT::rt(), OldNeighbors::nodelist:neighborhood(),
                 NewNeighbors::nodelist:neighborhood())
        -> {trigger_rebuild, rt()} | {ok, rt()}.
-callback filter_dead_node(rt(), DeadPid::comm:mypid(), Reason::fd:reason()) -> rt().

-callback to_pid_list(rt()) -> [comm:mypid()].
-callback get_size(rt() | external_rt()) -> non_neg_integer().
-callback get_replica_keys(key()) -> [key()].
-callback get_key_segment(key()) -> pos_integer().

-callback n() -> number().
-callback get_range(Begin::key(), End::key() | ?PLUS_INFINITY_TYPE) -> number().
-callback get_split_key(Begin::key(), End::key() | ?PLUS_INFINITY_TYPE,
                        SplitFraction::{Num::number(), Denom::pos_integer()})
        -> key() | ?PLUS_INFINITY_TYPE.
-callback get_split_keys(Begin::key(), End::key() | ?PLUS_INFINITY_TYPE,
                         Parts::pos_integer()) -> [key()].
-callback get_random_in_interval(intervals:simple_interval2()) -> key().

-callback dump(RT::rt()) -> KeyValueList::[{Index::string(), Node::string()}].

-callback to_list(dht_node_state:state()) -> nodelist:snodelist().
-callback export_rt_to_dht_node(rt(), Neighbors::nodelist:neighborhood()) -> external_rt().
-callback handle_custom_message(comm:message(), rt_loop:state_active()) -> rt_loop:state_active() | u

-callback check(OldRT::rt(), NewRT::rt(), Neighbors::nodelist:neighborhood(),
            ReportToFD::boolean()) -> ok.
-callback check(OldRT::rt(), NewRT::rt(), OldNeighbors::nodelist:neighborhood(),
            NewNeighbors::nodelist:neighborhood(), ReportToFD::boolean()) -> ok.
```

```

-callback check_config() -> boolean().
-callback wrap_message(Key::key(), Msg::comm:message(), State::dht_node_state:state(), Hops::non_neg_
-callback unwrap_message(Msg::comm:message(), State::dht_node_state:state()) ->
    comm:message().

-else.
-spec behaviour_info(atom()) -> [{atom(), arity()}] | undefined.
behaviour_info(callbacks) ->
    [
     % create a default routing table
     {empty_ext, 1},
     % initialize a routing table
     {init, 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, 3},
     % statistics
     {to_pid_list, 1}, {get_size, 1},
     % gets all (replicated) keys for a given (hashed) key
     % (for symmetric replication)
     {get_replica_keys, 1},
     % get the segment of the ring a key belongs to (1-4)
     {get_key_segment, 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, 3},
     % get a random key wihtin the requested interval
     {get_random_in_interval, 1},
     % 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},
     % wrap and unwrap lookup messages
     {wrap_message, 4},
     {unwrap_message, 2}
    ];
behaviour_info(_Other) ->
    undefined.
-endif.
```

`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/` 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.

`wrap_message/1` wraps a message send via a `dht_node_lookup:lookup_aux/4`.

`unwrap_message/2` unwraps a message send via `dht_node_lookup:lookup_aux/4` previously wrapped by `wrap_message/1`.

9.3.1. 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() :: {Neighbors     :: nodelist:neighborhood(),
                           RTState       :: ?RT:rt()}).
-type(state_inactive() :: {inactive,
                           MessageQueue::msg_queue:msg_queue()}).
%% -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`:

```
% Message handler to manage the trigger
on_active({trigger_rt}, State) ->
```

```
    msg_delay:send_trigger(get_base_interval(), {trigger_rt}),
    gen_component:post_op({periodic_rt_rebuild}, State);

% Actual periodic rebuilding of the RT
on_active({periodic_rt_rebuild}, {Neighbors, OldRT}) ->
    % start periodic stabilization
    % log:log(debug, "[ RT ] stabilize"),
    NewRT = ?RT:init_stabilize(Neighbors, OldRT),
    ?RT:check(OldRT, NewRT, Neighbors, true),
    {Neighbors, NewRT};
```

Periodically (see `pointer_base_stabilization_interval` config parameter) 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, NewNeighbors}, {_Neighbors, OldRT}) ->
    case ?RT:update(OldRT, OldNeighbors, NewNeighbors) of
        {trigger_rebuild, NewRT} ->
            ?RT:check(OldRT, NewRT, OldNeighbors, NewNeighbors, true),
            % trigger immediate rebuild
            gen_component:post_op({periodic_rt_rebuild}, {NewNeighbors, NewRT})
        ;
        {ok, NewRT} ->
            ?RT:check(OldRT, NewRT, OldNeighbors, NewNeighbors, true),
            {NewNeighbors, NewRT}
    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 rebuild. If so, it will stop any waiting trigger and schedule an immideate (periodic) stabilization.

9.3.2. 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() :: 0..16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. % 128 bit numbers
-opaque rt() :: Succ::node:node_type().
-type external_rt() :: Succ::node:node_type(). %% @todo: make opaque
-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.
-spec empty(nodelist:neighborhood()) -> rt().
empty(Neighbors) -> nodelist:succ(Neighbors).
```

File rt_simple.erl:

```
-spec empty_ext(nodelist:neighborhood()) -> external_rt().
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.
-spec get_random_node_id() -> key().
get_random_node_id() ->
    case config:read(key_creator) of
        random -> hash_key(randoms:getRandomString());
        random_with_bit_mask ->
            {Mask1, Mask2} = config:read(key_creator_bitmask),
            (hash_key(randoms:getRandomString()) 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.
-spec next_hop(dht_node_state:state(), key()) -> {succ | other, comm:mypid()}.
next_hop(State, Key) ->
    Neighbors = dht_node_state:get(State, neighbors),
    case intervals:in(Key, nodelist:succ_range(Neighbors)) of
        true -> {succ, node:pidX(nodelist:succ(Neighbors))};
        _    -> {other, node:pidX(nodelist:succ(Neighbors))}
    end.
```

Next hop is always the successor.

File rt_simple.erl:

```
%% @doc Triggered by a new stabilization round, renews the routing table.
-spec init_stabilize(nodelist:neighborhood(), rt()) -> rt().
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).
-spec filter_dead_node(rt(), DeadPid::comm:mypid(), Reason::fd:reason()) -> rt().
filter_dead_node(RT, _DeadPid, _Reason) -> 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.
-spec to_pid_list(rt()) -> [comm:mypid()].
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.
-spec get_size(rt() | external_rt()) -> non_neg_integer().
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.
-spec get_replica_keys(key()) -> [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.
-spec n() -> 16#100000000000000000000000000000000.
n() -> 16#100000000000000000000000000000000.
```

There are 2^{128} available keys.

File rt_simple.erl:

```
%% @doc Dumps the RT state for output in the web interface.
-spec dump(RT::rt()) -> KeyValueList::[{Index::string(), Node::string()}].
dump(Succ) -> [{"0", webhelpers:safe_html_string("~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,...
-spec to_list(dht_node_state:state()) -> nodelist:snodelist().
to_list(State) -> [dht_node_state:get(State, succ)].
```

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.
-spec export_rt_to_dht_node(rt(), Neighbors::nodelist:neighborhood()) -> external_rt().
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).
-spec check(OldRT::rt(), NewRT::rt(), Neighbors::nodelist:neighborhood(),
            ReportToFD::boolean()) -> ok.
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.
-spec check(OldRT::rt(), NewRT::rt(), OldNeighbors::nodelist:neighborhood(),
            NewNeighbors::nodelist:neighborhood(), ReportToFD::boolean()) -> ok.
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(self(), 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).

File rt_simple.hrl:

```
%% @doc Notifies the dht_node and failure detector if the routing table changed.
%%      Provided for convenience (see check/5).
-spec check(OldRT::rt(), NewRT::rt(), Neighbors::nodelist:neighborhood(),
            ReportToFD::boolean()) -> ok.
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.
-spec check(OldRT::rt(), NewRT::rt(), OldNeighbors::nodelist:neighborhood(),
            NewNeighbors::nodelist:neighborhood(), ReportToFD::boolean()) -> ok.
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(self(), OldPids, NewPids);
                _ -> ok
            end
    end.
```

File rt_simple.erl:

```
%% @doc Wrap lookup messages. This is a noop in rt_simple.
-spec wrap_message(Key::key(), Msg::comm:message(), State::dht_node_state:state(),
                   Hops::non_neg_integer()) -> comm:message().
wrap_message(_Key, Msg, _State, _Hops) -> Msg.
```

Wraps a message send via `dht_node_lookup:lookup/4` if needed. This routing algorithm does not need callbacks when finishing the lookup, so it does not need to wrap the message.

File rt_simple.erl:

```
%% @doc Unwrap lookup messages. This is a noop in rt_simple.
-spec unwrap_message(Msg::comm:message(), State::dht_node_state:state()) -> comm:message().
unwrap_message(Msg, _State) -> Msg.
```

Unwraps a message previously wrapped with `rt_simple:wrap_message/1`. As that function does not wrap messages, `rt_simple:unwrap_message/2` doesn't have to do anything as well.

9.3.3. Chord routing table (rt_chord)

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

Data types

File rt_chord.erl:

```
-type key() :: 0..16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. % 128 bit numbers
-type index() :: {pos_integer(), non_neg_integer()}.
-opaque rt() :: gb_trees:tree(index(), Node::node:node_type()).
-type external_rt() :: gb_trees:tree(NodeId::key(), Node::node:node_type()). %% @todo: make opaque
-type 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.
-spec empty(nodelist:neighborhood()) -> rt().
empty(_Neighbors) -> gb_trees:empty().
```

File rt_chord.erl:

```
-spec empty_ext(nodelist:neighborhood()) -> external_rt().
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!
-spec next_hop(dht_node_state:state(), key()) -> {succ | other, comm:mypid()}.
next_hop(State, Id) ->
    Neighbors = dht_node_state:get(State, neighbors),
    case intervals:in(Id, nodelist:succ_range(Neighbors)) of
        true ->
            {succ, node:pidX(nodelist:succ(Neighbors))};
        _ ->
            % 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 ->
                             nodelist:succ(Neighbors);
                         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(Neighbors, Id, NodeRT)
                end,
            {other, 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:

```
%% @doc Starts the stabilization routine.
-spec init_stabilize(nodelist:neighborhood(), rt()) -> rt().
init_stabilize(Neighbors, RT) ->
```

```
    % calculate the longest finger
    case first_index(Neighbors) of
        null -> ok;
        {Key, Index} ->
            % trigger a lookup for Key
            api_dht_raw:unreliable_lookup(
              Key, {?send_to_group_member, routing_table,
                    {rt_get_node, comm:this(), Index}})
    end,
    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`:

```
%% @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() | 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}, ?SEND_OPTIONS),
    State;
handle_custom_message({rt_get_node_response, Index, Node}, State) ->
    OldRT = rt_loop:get_rt(State),
    Neighbors = rt_loop:get_neighb(State),
    case stabilize(Neighbors, OldRT, Index, Node) of
        {NewRT, true} ->
            check_do_update(OldRT, NewRT, rt_loop:get_neighb(State), true);
        {NewRT, false} -> ok
    end,
    rt_loop:set_rt(State, NewRT);
handle_custom_message(_Message, _State) ->
    unknown_event.
```

File `rt_chord.erl`:

```
%% @doc Updates one entry in the routing table and triggers the next update.
%%      Changed indicates whether a new node was inserted (the RT structure may
%%      change independently from this indicator!).
-spec stabilize(Neighbors::nodelist:neighborhood(), OldRT::rt(), Index::index(),
                Node::node:node_type()) -> {NewRT::rt(), Changed::boolean()}.
stabilize(Neighbors, RT, Index, Node) ->
    MyId = nodelist:nodeid(Neighbors),
    Succ = nodelist:succ(Neighbors),
    case (node:id(Succ) =/= node:id(Node))   % reached succ?
        andalso (not intervals:in(           % there should be nothing shorter
                   node:id(Node),            %   than succ
                   nodelist:succ_range(Neighbors))) of
        true ->
            NextIndex = next_index(Index),
            NextKey = calculateKey(MyId, NextIndex),
            CurrentKey = calculateKey(MyId, Index),
            case CurrentKey =/= NextKey of
                true ->
                    Msg = {rt_get_node, comm:this(), NextIndex},
```

```
                     api_dht_raw:unreliable_lookup(
                       NextKey, {?send_to_group_member, routing_table, Msg});
                 _ -> ok
             end,
             Changed = (Index =:= first_index() orelse
                            (gb_trees:lookup(prev_index(Index), RT) =/= {value, Node})),
             {gb_trees:enter(Index, Node, RT), Changed};
         _ ->
             {RT, false}
     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())
        -> {ok | trigger_rebuild, rt()}.
update(OldRT, OldNeighbors, NewNeighbors) ->
    NewPred = nodelist:pred(NewNeighbors),
    OldSucc = nodelist:succ(OldNeighbors),
    NewSucc = nodelist:succ(NewNeighbors),
    NewNodeId = nodelist:nodeid(NewNeighbors),
    % only re-build if a new successor occurs or the new node ID is not between
    % Pred and Succ any more (which should not happen since this must come from
    % a slide!)
    % -> if not rebuilding, update the node IDs though
    case node:same_process(OldSucc, NewSucc) andalso
             intervals:in(NewNodeId, node:mk_interval_between_nodes(NewPred, NewSucc)) of
        true ->
            NewRT = gb_trees:map(
                      fun(_K, N) ->
                              case node:same_process(N, NewPred) of
                                  true  ->
                                      node:newer(N, NewPred);
                                  false ->
                                      case node:same_process(N, NewSucc) of
                                          true  -> node:newer(N, NewSucc);
                                          false -> N
                                      end
                              end
                      end, OldRT),
            {ok, NewRT};
        false ->
            % to be on the safe side ...
            {trigger_rebuild, empty(NewNeighbors)}
    end.
```

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.
-spec filter_dead_node(rt(), DeadPid::comm:mypid(), Reason::fd:reason()) -> rt().
filter_dead_node(RT, DeadPid, _Reason) ->
    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`:

```
-spec export_rt_to_dht_node(rt(), Neighbors::nodelist:neighborhood()) -> external_rt().
export_rt_to_dht_node(RT, Neighbors) ->
    Id = nodelist:nodeid(Neighbors),
    Pred = nodelist:pred(Neighbors),
    Succ = nodelist:succ(Neighbors),
    % always include the pred and succ in the external representation
    % note: we are subscribed at the RM for changes to these nodes
    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).
-spec check(OldRT::rt(), NewRT::rt(), Neighbors::nodelist:neighborhood(),
            ReportToFD::boolean()) -> ok.
check(OldRT, OldRT, _Neighbors, _ReportToFD) ->
    ok;
check(OldRT, NewRT, Neighbors, ReportToFD) ->
    check_do_update(OldRT, NewRT, 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.
-spec check(OldRT::rt(), NewRT::rt(), OldNeighbors::nodelist:neighborhood(),
            NewNeighbors::nodelist:neighborhood(), ReportToFD::boolean()) -> ok.
check(OldRT, NewRT, OldNeighbors, NewNeighbors, ReportToFD) ->
    case nodelist:pred(OldNeighbors) =:= nodelist:pred(NewNeighbors) andalso
             nodelist:succ(OldNeighbors) =:= nodelist:succ(NewNeighbors) andalso
             OldRT =:= NewRT of
        true -> ok;
        _ -> check_do_update(OldRT, NewRT, NewNeighbors, ReportToFD)
    end.

%% @doc Helper for check/4 and check/5.
-spec check_do_update(OldRT::rt(), NewRT::rt(), NewNeighbors::nodelist:neighborhood(),
                      ReportToFD::boolean()) -> ok.
check_do_update(OldRT, NewRT, NewNeighbors, ReportToFD) ->
    Pid = pid_groups:get_my(dht_node),
    RT_ext = export_rt_to_dht_node(NewRT, NewNeighbors),
    case Pid of
        failed -> ok;
        _      -> comm:send_local(Pid, {rt_update, RT_ext})
    end,
    % update failure detector:
    case ReportToFD of
        true ->
            NewPids = to_pid_list(NewRT),
            OldPids = to_pid_list(OldRT),
            fd:update_subscriptions(self(), OldPids, NewPids);
        _ -> ok
    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).

File `rt_chord.erl`:

```
%% @doc Wrap lookup messages. This is a noop in Chord.
-spec wrap_message(Key::key(), Msg::comm:message(), State::dht_node_state:state(),
                   Hops::non_neg_integer()) -> comm:message().
wrap_message(_Key, Msg, _State, _Hops) -> Msg.
```

Wraps a message send via `dht_node_lookup:lookup/4` if needed. This routing algorithm does not need callbacks when finishing the lookup, so it does not need to wrap the message.

File `rt_chord.erl`:

```
%% @doc Unwrap lookup messages. This is a noop in Chord.
-spec unwrap_message(Msg::comm:message(), State::dht_node_state:state()) -> comm:message().
unwrap_message(Msg, _State) -> Msg.
```

Unwraps a message previously wrapped with `rt_chord:wrap_message/1`. As that function does not wrap messages, `rt_chord:unwrap_message/2` doesn't have to do anything as well.

9.4. Local Datastore

9.5. Cyclon

9.6. Vivaldi Coordinates

9.7. Estimated Global Information (Gossiping)

9.8. Load Balancing

9.9. Broadcast Trees

10. Transactions in Scalaris

10.1. The Paxos Module

10.2. Transactions using Paxos Commit

10.3. Applying the Tx-Modules to replicated DHTs

Introduces transaction processing on top of a Overlay

11. How a node joins the system

After starting a new Scalaris-System as described in Section 3.2.1 on page 17, ten additional local nodes can be started by typing `api_vm:add_nodes(10)` in the Erlang-Shell that is opened during startup [1].

```
scalaris> ./bin/firstnode.sh
[...]
(firstnode@csr-pc9)1> api_vm:add_nodes(10)
```

In the following we will trace what this function does in order to add additional nodes to the system. The function `api_vm:add_nodes(pos_integer())` is defined as follows.

File `api_vm.erl`:

```
%% @doc Adds Number Scalaris nodes to this VM.
-spec add_nodes(non_neg_integer()) -> {[pid_groups:groupname()], [{error, term()}]}.
add_nodes(Number) when is_integer(Number) andalso Number >= 0 ->
    Result = {Ok, _Failed} = admin:add_nodes(Number),
    % at least wait for the successful nodes to have joined, i.e. left the join phases
    util:wait_for(
      fun() ->
              DhtModule = config:read(dht_node),
              NotReady = [Name || Name <- Ok,
                                  not DhtModule:is_alive(
                                    gen_component:get_state(
                                      pid_groups:pid_of(Name, dht_node)))],
              [] =:= NotReady
      end),
    Result.
```

It uses the `admin:add_nodes/1` function to actually add the given number of nodes and then waits for all nodes to successfully complete their join phases.

File `admin.erl`:

```
% @doc add new Scalaris nodes on the local node
-spec add_node_at_id(?RT:key()) -> pid_groups:groupname() | {error, term()}.
add_node_at_id(Id) ->
    add_node([{{dht_node, id}, Id}, {skip_psv_lb}, {add_node}]).

-spec add_node([tuple()]) -> pid_groups:groupname() | {error, term()}.
add_node(Options) ->
    DhtNodeId = randoms:getRandomString(),
    Group = pid_groups:new("dht_node_"),
    Desc = sup:supervisor_desc(
             DhtNodeId, sup_dht_node, start_link,
             [{Group,
               [{my_sup_dht_node_id, DhtNodeId}, {add_node} | Options]}]),
    Sup = erlang:whereis(main_sup),
    case sup:start_sup_as_child(["+"], Sup, Desc) of
        {ok, _Child, Group}           ->
            DhtNodePid = pid_groups:pid_of(Group, dht_node),
            comm:send_local(DhtNodePid, {join, start}),
            Group;
        {error, already_present}      -> add_node(Options); % try again, different Id
```

[1] Increase the log level to `info` to get more detailed startup logs. See Section 3.1.1 on page 16

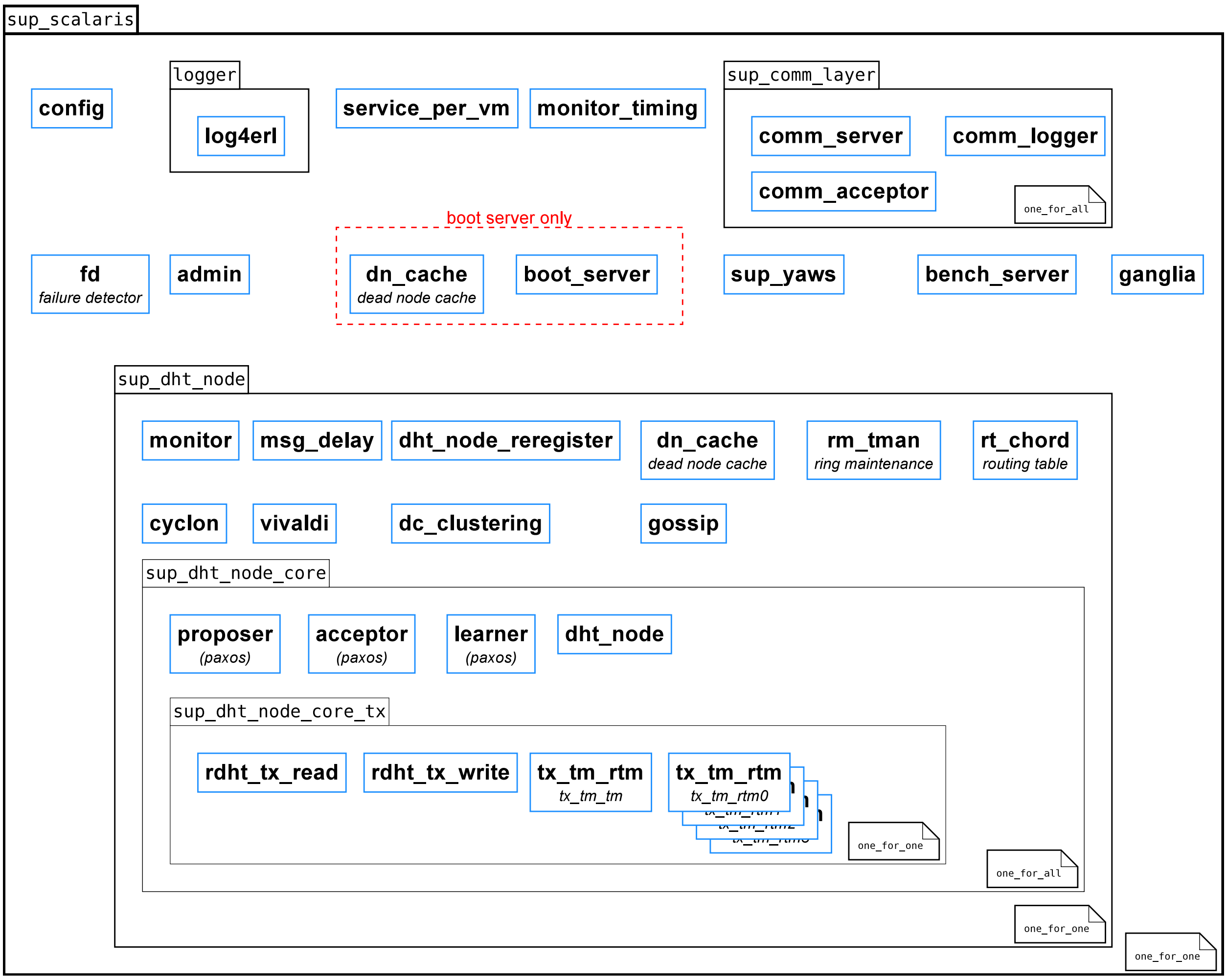
```
        {error, {already_started, _}} -> add_node(Options); % try again, different Id
        {error, _Error} = X           -> X
    end.

-spec add_nodes(non_neg_integer()) -> {[pid_groups:groupname()], [{error, term()}]}.
add_nodes(0) -> {[], []};
add_nodes(Count) ->
    Results = [add_node([]) || _X <- lists:seq(1, Count)],
    lists:partition(fun(E) -> not is_tuple(E) end, Results).
```

The function `admin:add_nodes/1` 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.

11.1. 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.

11.2. 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([{pid_groups:groupname(), [tuple()]}])
        -> {ok, {{one_for_one, MaxRetries::pos_integer(),
                  PeriodInSeconds::pos_integer()}, []}}.
init([{DHTNodeGroup, _Options}] = X) ->
    pid_groups:join_as(DHTNodeGroup, ?MODULE),
    supspec(X).
```

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: Monitor, Delayer, Reregister, DeadNodeCache, RingMaintenance, RoutingTable, Cyclon, Vivaldi, DC_Clustering, Gossip and a SupDHTNodeCore_AND 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.

11.3. 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::supervisor:child_spec()]}}.
init([DHTNodeGroup, _Options] = X) ->
    pid_groups:join_as(DHTNodeGroup, ?MODULE),
    supspec(X).
```

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, fun ?MODULE:on/2, Options,
                             [{pid_groups_join_as, DHTNodeGroup, dht_node},
                              {wait_for_init},
                              {spawn_opts, [{fullsweep_after, 0},
                                            {min_heap_size, 131071}]}]).
```

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`).

11.4. 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()])
        -> dht_node_state:state() |
           {'$gen_component', [{on_handler, Handler::gen_component:handler()}], State::dht_node_join:
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),
    % start trigger here to prevent infection when tracing e.g. node joins
    % (otherwise the trigger would be started at the end of the join and thus
    % be infected forever)
    % NOTE: any trigger started here, needs an exception for queuing messages
    %       in dht_node_join to prevent infection with msg_queue:send/1!
    rm_loop:init_first(),
    dht_node_move:send_trigger(),

    Recover = config:read(start_type) =:= recover,
    case {is_first(Options), config:read(leases), Recover, is_add_nodes(Options)} of
        {_   , true, true, false} ->
            % we are recovering
            dht_node_join_recover:join(Options);
        {true, true, false, _} ->
            msg_delay:send_trigger(1, {l_on_cseq, renew_leases}),
            Id = l_on_cseq:id(intervals:all()),
            TmpState = dht_node_join:join_as_first(Id, 0, Options),
            %% we have to inject the first lease by hand, as otherwise
            %% no routing will work.
            l_on_cseq:add_first_lease_to_db(Id, TmpState);
        {false, true, _, true} ->
            msg_delay:send_trigger(1, {l_on_cseq, renew_leases}),
            % get my ID (if set, otherwise chose a random ID):
            Id = case lists:keyfind({dht_node, id}, 1, Options) of
```

```
                     {{dht_node, id}, IdX} -> IdX;
                     _ -> ?RT:get_random_node_id()
                 end,
            dht_node_join:join_as_other(Id, 0, Options);
        {IsFirst, _, _, _} ->
            % get my ID (if set, otherwise chose a random ID):
            Id = case lists:keyfind({dht_node, id}, 1, Options) of
                     {{dht_node, id}, IdX} -> IdX;
                     _ -> ?RT:get_random_node_id()
                 end,
            if IsFirst -> dht_node_join:join_as_first(Id, 0, Options);
               true    -> dht_node_join:join_as_other(Id, 0, Options)
            end
    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 an ID was given to dht_node:init/1 function as a {{dht_node, id}, KEY} tuple, the given Id will be used. Otherwise a random key is generated. Depending on whether the node is the first inside a VM marked as first or not, the according function in dht_node_join is called. Also the pid of the node's supervisor is kept for future reference.

11.5. 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_state(Message, State). On the existing node, join messages will be processed by dht_node_join:process_join_msg(Message, State).

11.5.1. A single node joining an empty ring

File dht_node_join.erl:

```
-spec join_as_first(Id::?RT:key(), IdVersion::non_neg_integer(), Options::[tuple()])
        -> dht_node_state:state().
join_as_first(Id, IdVersion, _Options) ->
    log:log(info, "[ Node ~w ] joining as first: (~.0p, ~.0p)",
            [self(), Id, IdVersion]),
    Me = node:new(comm:this(), Id, IdVersion),
    % join complete, State is the first "State"
    finish_join(Me, Me, Me, db_dht:new(db_dht), msg_queue:new(), []).
```

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. It will trigger all known nodes to initialize the comm layer and then finish the join. 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(),
                rm_state   = ?required(state, rm_state)   :: rm_loop:state(),
                join_time  = ?required(state, join_time)  :: erlang:timestamp(),
                db         = ?required(state, db)         :: db_dht:db(),
                tx_tp_db   = ?required(state, tx_tp_db)   :: any(),
```

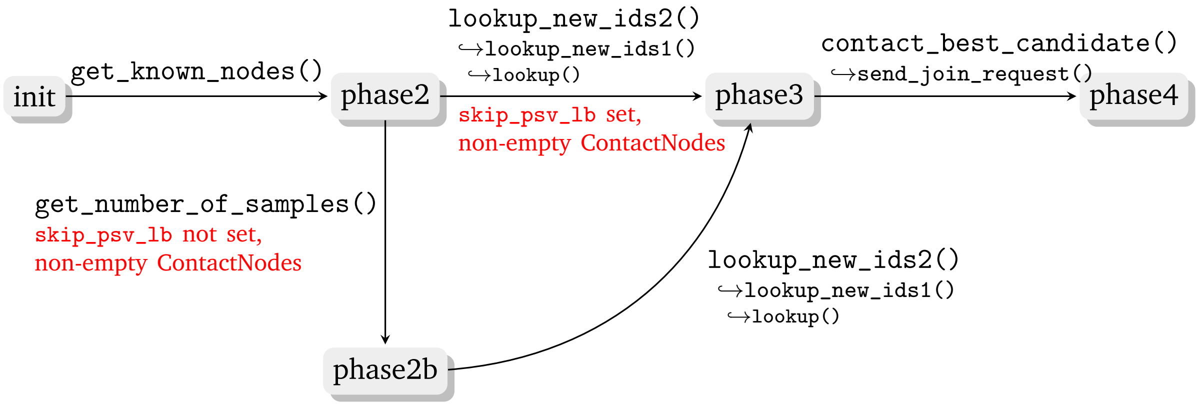
```
                proposer   = ?required(state, proposer)  :: pid(),
                % slide with pred (must not overlap with 'slide with succ'!):
                slide_pred              = null :: slide_op:slide_op() | null,
                % slide with succ (must not overlap with 'slide with pred'!):
                slide_succ              = null :: slide_op:slide_op() | null,
                % additional range to respond to during a move:
                db_range   = []   :: [{intervals:interval(), slide_op:id()}],
                monitor_proc            = ?required(state, monitor_proc) :: pid(),
                prbr_kv_db = ?required(state, prbr_kv_db) :: prbr:state(),
                txid_db1 = ?required(state, txid_db1) :: prbr:state(),
                txid_db2 = ?required(state, txid_db2) :: prbr:state(),
                txid_db3 = ?required(state, txid_db3) :: prbr:state(),
                txid_db4 = ?required(state, txid_db4) :: prbr:state(),
                lease_db1 = ?required(state, lease_db1) :: prbr:state(),
                lease_db2 = ?required(state, lease_db2) :: prbr:state(),
                lease_db3 = ?required(state, lease_db3) :: prbr:state(),
                lease_db4 = ?required(state, lease_db4) :: prbr:state(),
                lease_list = ?required(state, lease_list) :: lease_list:lease_list(),
                snapshot_state   = null :: snapshot_state:snapshot_state() | null,
                mr_state   = ?required(state, mr_state)  :: orddict:orddict(),
                mr_master_state   = ?required(state, mr_master_state)  :: orddict:orddict()
               }).
-opaque state() :: #state{}.
```

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

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

- **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
- **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.



At first all nodes set in the `known_hosts` configuration parameter are contacted. Their responses are then handled in phase 2. In order to separate the join state from the ordinary `dht_node` state, the `gen_component` is instructed to use the `dht_node:on_join/2` message handler which delegates every message to `dht_node_join:process_join_state/2`.

File `dht_node_join.erl`:

```
-spec join_as_other(Id::?RT:key(), IdVersion::non_neg_integer(), Options::[tuple()])
        -> {'$gen_component', [{on_handler, Handler::gen_component:handler()}],
            State::{join, phase2(), msg_queue:msg_queue()}}.
```

```
join_as_other(Id, IdVersion, Options) ->
    log:log(info,"[ Node ~w ] joining, trying ID: (~.0p, ~.0p)",
            [self(), Id, IdVersion]),
    JoinUUID = uid:get_pids_uid(),
    gen_component:change_handler(
      {join, {phase1, JoinUUID, Options, IdVersion, [], [Id], []},
       msg_queue:new()},
      fun ?MODULE:process_join_state/2).
```

Phase 2 and 2b

Phase 2 collects all `dht_node` processes inside the contacted VMs. It therefore mainly processes `get_dht_nodes_response` messages and integrates all received nodes into the list of available connections. 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 only use the ID that has been initially passed to `dht_node_join:join_as_other/3`, 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 connections. These phases' operations will not be interrupted and nothing else is changed though.

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} = _Msg,
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 ->
    ?TRACE_JOIN1(_Msg, JoinState),
    JoinOptions = get_join_options(JoinState),
    %% additional nodes required when firstnode jumps and he's the only known host
    DhtNodes = Nodes ++ proplists:get_value(bootstrap_nodes, JoinOptions, []),
    Connections = [{null, Node} || Node <- DhtNodes, Node =/= comm:this()],
    JoinState1 = add_connections(Connections, JoinState, back),
    NewJoinState = phase2_next_step(JoinState1, Connections),
    ?TRACE_JOIN_STATE(NewJoinState),
    {join, NewJoinState, QueuedMessages};

% in all other phases, just add the provided nodes:
process_join_state({get_dht_nodes_response, Nodes} = _Msg,
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2b orelse
           element(1, JoinState) =:= phase3 orelse
           element(1, JoinState) =:= phase4 ->
    ?TRACE_JOIN1(_Msg, JoinState),
    Connections = [{null, Node} || Node <- Nodes, Node =/= comm:this()],
    JoinState1 = add_connections(Connections, JoinState, back),
    ?TRACE_JOIN_STATE(JoinState1),
    {join, JoinState1, 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 phase2b, also accept message in
```

```
% phase2, e.g. messages arriving from previous calls
process_join_state({join, get_number_of_samples, Samples, Conn} = _Msg,
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 orelse
           element(1, JoinState) =:= phase2b ->
    ?TRACE_JOIN1(_Msg, JoinState),
    % prefer node that send get_number_of_samples as first contact node
    JoinState1 = reset_connection(Conn, JoinState),
    % (re-)issue lookups for all existing IDs and
    % create additional samples, if required
    NewJoinState = lookup_new_ids2(Samples, JoinState1),
    ?TRACE_JOIN_STATE(NewJoinState),
    {join, NewJoinState, QueuedMessages};

% ignore message arriving in other phases:
process_join_state({join, get_number_of_samples, _Samples, Conn} = _Msg,
                   {join, JoinState, QueuedMessages}) ->
    ?TRACE_JOIN1(_Msg, JoinState),
    NewJoinState = reset_connection(Conn, JoinState),
    ?TRACE_JOIN_STATE(NewJoinState),
    {join, NewJoinState, QueuedMessages};
```

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 would 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, Conn} = _Msg, State) ->
    ?TRACE1(_Msg, State),
    call_lb_psv(LbPsv, create_join, [State, Key, Source_PID, Conn]);
```

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, Conn} = _Msg,
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase3 ->
    ?TRACE_JOIN1(_Msg, JoinState),
    JoinState0 = reset_connection(Conn, JoinState),
    JoinState1 = remove_join_id(OrigJoinId, JoinState0),
    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,
    ?TRACE_JOIN_STATE(NewJoinState),
    {join, NewJoinState, QueuedMessages};

% In phase 2 or 2b, also add the candidate but do not continue.
% In phase 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, Conn} = _Msg,
                   {join, JoinState, QueuedMessages})
  when element(1, JoinState) =:= phase2 orelse
           element(1, JoinState) =:= phase2b orelse
           element(1, JoinState) =:= phase4 ->
    ?TRACE_JOIN1(_Msg, JoinState),
    JoinState0 = reset_connection(Conn, JoinState),
    JoinState1 = remove_join_id(OrigJoinId, JoinState0),
    JoinState2 = case get_phase(JoinState1) of
                     phase4 -> integrate_candidate(Candidate, JoinState1, back);
                     _      -> integrate_candidate(Candidate, JoinState1, front)
                 end,
    ?TRACE_JOIN_STATE(JoinState2),
    {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, send a `join_request` message and continue with 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() | phase4().
contact_best_candidate(JoinState) ->
    JoinState1 = sort_candidates(JoinState),
    send_join_request(JoinState1, 0).
```

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 (can happen, e.g. when join candidates are busy):
            start_over(JoinState);
        [BestCand | _] ->
            Id = node_details:get(lb_op:get(BestCand, n1_new), new_key),
            IdVersion = get_id_version(JoinState),
            NewSucc = node_details:get(lb_op:get(BestCand, n1succ_new), node),
            Me = node:new(comm:this(), Id, IdVersion),
            CandId = lb_op:get(BestCand, id),
            MyMTE = case dht_node_move:use_incremental_slides() of
                        true -> dht_node_move:get_max_transport_entries();
                        false -> unknown
                    end,
            Msg = {join, join_request, Me, CandId, MyMTE},
            ?TRACE_SEND(node:pidX(NewSucc), Msg),
            comm:send(node:pidX(NewSucc), Msg),
            msg_delay:send_local(
              get_join_request_timeout() div 1000, self(),
              {join, join_request_timeout, Timeouts, CandId, get_join_uuid(JoinState)}),
            set_phase(phase4, JoinState)
    end.
```

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

```
process_join_msg({join, join_request, NewPred, CandId, MaxTransportEntries} = _Msg, State)
  when (not is_atom(NewPred)) -> % avoid confusion with not_responsible message
    ?TRACE1(_Msg, State),
    TargetId = node:id(NewPred),
    JoinType = {join, 'send'},
    MyNode = dht_node_state:get(State, node),
    Command = dht_node_move:check_setup_slide_not_found(
                State, JoinType, MyNode, NewPred, TargetId),
    case Command of
        {ok, JoinType} ->
            MoveFullId = uid:get_global_uid(),
            State1 = dht_node_move:exec_setup_slide_not_found(
                       Command, State, MoveFullId, NewPred, TargetId, join,
                       MaxTransportEntries, null, nomsg, {none}, false),
            % set up slide, now send join_response:
            MyOldPred = dht_node_state:get(State1, pred),
            % 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),
            SlideOp = dht_node_state:get(State1, slide_pred),
            Msg = {join, join_response, MyNode, MyOldPred, MoveFullId, CandId,
                   slide_op:get_target_id(SlideOp), slide_op:get_next_op(SlideOp)},
            dht_node_move:send(node:pidX(NewPred), Msg, MoveFullId),
            State1;
        {abort, ongoing_slide, JoinType} ->
            ?TRACE("[ ~.0p ]~n  rejecting join_request from ~.0p due to a running slide~n",
                   [self(), NewPred]),
            ?TRACE_SEND(node:pidX(NewPred), {join, join_response, busy, CandId}),
            comm:send(node:pidX(NewPred), {join, join_response, busy, CandId}),
            State;
        {abort, _Reason, JoinType} -> % all other errors:
            ?TRACE("~p", [Command]),
            ?TRACE_SEND(node:pidX(NewPred),
                        {join, join_response, not_responsible, CandId}),
            comm:send(node:pidX(NewPred),
                      {join, join_response, not_responsible, CandId}),
            State
    end;
```

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, Reason, CandId} = _Msg,
                   {join, JoinState, QueuedMessages} = State)
  when element(1, JoinState) =:= phase4 andalso
```

```
            (Reason =:= not_responsible orelse Reason =:= busy) ->
    ?TRACE_JOIN1(_Msg, JoinState),
    % the node we contacted is not responsible for the selected 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()]),
            NewJoinState = start_over(JoinState),
            ?TRACE_JOIN_STATE(NewJoinState),
            {join, NewJoinState, QueuedMessages};
        [Candidate | _Rest] ->
            case lb_op:get(Candidate, id) =:= CandId of
                false -> State; % unrelated/old message
                _ ->
                    if Reason =:= not_responsible ->
                            log:log(info,
                                    "[ Node ~w ] node contacted for join is not "
                                        "responsible for the selected ID (anymore), "
                                        "trying next candidate",
                                    [self()]);
                       Reason =:= busy ->
                            log:log(info,
                                    "[ Node ~w ] node contacted for join is busy, "
                                        "trying next candidate",
                                    [self()])
                    end,
                    NewJoinState = try_next_candidate(JoinState),
                    ?TRACE_JOIN_STATE(NewJoinState),
                    {join, NewJoinState, QueuedMessages}
            end
    end;

% in other phases remove the candidate from the list (if it still exists):
process_join_state({join, join_response, Reason, CandId} = _Msg,
                   {join, JoinState, QueuedMessages})
  when (Reason =:= not_responsible orelse Reason =:= busy) ->
    ?TRACE_JOIN1(_Msg, JoinState),
    {join, remove_candidate(CandId, JoinState), QueuedMessages};

% note: accept (delayed) join_response messages in any phase
process_join_state({join, join_response, Succ, Pred, MoveId, CandId, TargetId, NextOp} = _Msg,
                   {join, JoinState, QueuedMessages} = State) ->
    ?TRACE_JOIN1(_Msg, JoinState),
    % only act on related messages, i.e. messages from the current candidate
    Phase = get_phase(JoinState),
    State1 = 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()]),
            reject_join_response(Succ, Pred, MoveId, CandId),
            NewJoinState = start_over(JoinState),
            ?TRACE_JOIN_STATE(NewJoinState),
            {join, NewJoinState, QueuedMessages};
        [] ->
            % in all other phases, ignore the delayed join_response if no
            % candidates exist
            reject_join_response(Succ, Pred, MoveId, CandId),
            State;
        [Candidate | _Rest] ->
            CandidateNode = node_details:get(lb_op:get(Candidate, n1succ_new), node),
            CandidateNodeSame = node:same_process(CandidateNode, Succ),
            case lb_op:get(Candidate, id) =:= CandId of
                false ->
                    % ignore old/unrelated message
                    log:log(warn, "[ Node ~w ] ignoring old or unrelated "
                                  "join_response message", [self()]),
                    reject_join_response(Succ, Pred, MoveId, CandId),
                    State;
                _ when not CandidateNodeSame ->
```

```
                        % id is correct but the node is not (should never happen!)
                        log:log(error, "[ Node ~w ] got join_response but the node "
                                      "changed, trying next candidate", [self()]),
                        reject_join_response(Succ, Pred, MoveId, CandId),
                        NewJoinState = try_next_candidate(JoinState),
                        ?TRACE_JOIN_STATE(NewJoinState),
                        {join, NewJoinState, QueuedMessages};
                    _ ->
                        MyId = TargetId,
                        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()]),
                                reject_join_response(Succ, Pred, MoveId, CandId),
                                % note: can not keep Id, even if skip_psv_lb is set
                                JoinState1 = remove_candidate_front(JoinState),
                                NewJoinState = contact_best_candidate(JoinState1),
                                ?TRACE_JOIN_STATE(NewJoinState),
                                {join, NewJoinState, QueuedMessages};
                            _ ->
                                ?TRACE("[ ~.0p ]~n  joined MyId:~.0p, MyIdVersion:~.0p~n  "
                                           "Succ: ~.0p~n  Pred: ~.0p~n",
                                           [self(), MyId, MyIdVersion, Succ, Pred]),
                                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),

                                JoinOptions = get_join_options(JoinState),

                                finish_join_and_slide(Me, Pred, Succ, db_dht:new(db_dht),
                                                      QueuedMessages, MoveId, NextOp, JoinOptions)
                        end
                end
        end,
    State1;
```

File dht_node_join.erl:

```
%% @doc Finishes the join and sends all queued messages.
-spec finish_join(Me::node:node_type(), Pred::node:node_type(),
                  Succ::node:node_type(), DB::db_dht:db(),
                  QueuedMessages::msg_queue:msg_queue(),
                  JoinOptions::[tuple()])
        -> dht_node_state:state().
finish_join(Me, Pred, Succ, DB, QueuedMessages, JoinOptions) ->
    %% get old rt loop subscribtion table (if available)
    MoveState = proplists:get_value(move_state, JoinOptions, []),
    OldSubscrTable = proplists:get_value(subscr_table, MoveState, null),
    RMState = rm_loop:init(Me, Pred, Succ, OldSubscrTable),
    Neighbors = rm_loop:get_neighbors(RMState),
    % wait for the ring maintenance to initialize and tell us its table ID
    rt_loop:activate(Neighbors),
    if MoveState =:= [] ->
           dc_clustering:activate(),
           gossip:activate(Neighbors);
       true -> ok
    end,
    dht_node_reregister:activate(),
    msg_queue:send(QueuedMessages),
    NewRT_ext = ?RT:empty_ext(Neighbors),
    service_per_vm:register_dht_node(node:pidX(Me)),
    dht_node_state:new(NewRT_ext, RMState, DB).

-spec reject_join_response(Succ::node:node_type(), Pred::node:node_type(),
                           MoveFullId::slide_op:id(), CandId::lb_op:id()) -> ok.
reject_join_response(Succ, _Pred, MoveId, _CandId) ->
    % similar to dht_node_move:abort_slide/9 - keep message in sync!
```

```
    Msg = {move, slide_abort, pred, MoveId, ongoing_slide},
    ?TRACE_SEND(node:pidX(Succ), Msg),
    dht_node_move:send_no_slide(node:pidX(Succ), Msg, 0).

%% @doc Finishes the join by setting up a slide operation to get the data from
%%      the other node and sends all queued messages.
-spec finish_join_and_slide(Me::node:node_type(), Pred::node:node_type(),
                            Succ::node:node_type(), DB::db_dht:db(),
                            QueuedMessages::msg_queue:msg_queue(),
                            MoveId::slide_op:id(), NextOp::slide_op:next_op(),
                            JoinOptions::[tuple()])
        -> {'$gen_component', [{on_handler, Handler::gen_component:handler()}],
            State::dht_node_state:state()}.
finish_join_and_slide(Me, Pred, Succ, DB, QueuedMessages, MoveId, NextOp, JoinOptions) ->
    State = finish_join(Me, Pred, Succ, DB, QueuedMessages, JoinOptions),
    {SourcePid, Tag} =
        case lists:keyfind(jump, 1, JoinOptions) of
            {jump, JumpTag, Pid} -> {Pid, JumpTag};
            _ -> {null, join}
        end,
    State1 = dht_node_move:exec_setup_slide_not_found(
               {ok, {join, 'rcv'}}, State, MoveId, Succ, node:id(Me), Tag,
               unknown, SourcePid, nomsg, NextOp, false),
    gen_component:change_handler(State1, fun dht_node:on/2).
```

The macro `?RT` maps to the configured routing algorithm. It is defined in `include/scalaris.hrl`. For further details on the routing see Chapter 9.3 on page 60.

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><code>known_hosts_timeout</code></th><th><code>get_number_of_samples_timeout</code></th><th><code>lookup_timeout</code></th><th><code>join_request_timeout</code></th><th><code>timeout</code></th></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

12. How data is transferred (atomically)

A data transfer from a node to one of its (two) neighbours is also called a *slide*. A slide operation is defined in the `slide_op` module, the protocol is mainly implemented in `dht_node_move`. Parts of the slide are dependent on the ring maintenance implementation and are split off into modules implementing the `slide_beh` behaviour.

Though the protocols are mainly symmetric, we distinguish between sending data to the predecessor and sending data to the successor, respectively. In the following protocol visualisations, arrows denote message exchanges, pseudo-code for operations that are being executed is put at the side of each time bar. Functions in green are those implemented in the `slide_beh` behaviour, if annotated with an arrow pointing to itself, this callback is asynchronous. During the protocol, the slide operation goes through several phases which are show in black boxes.

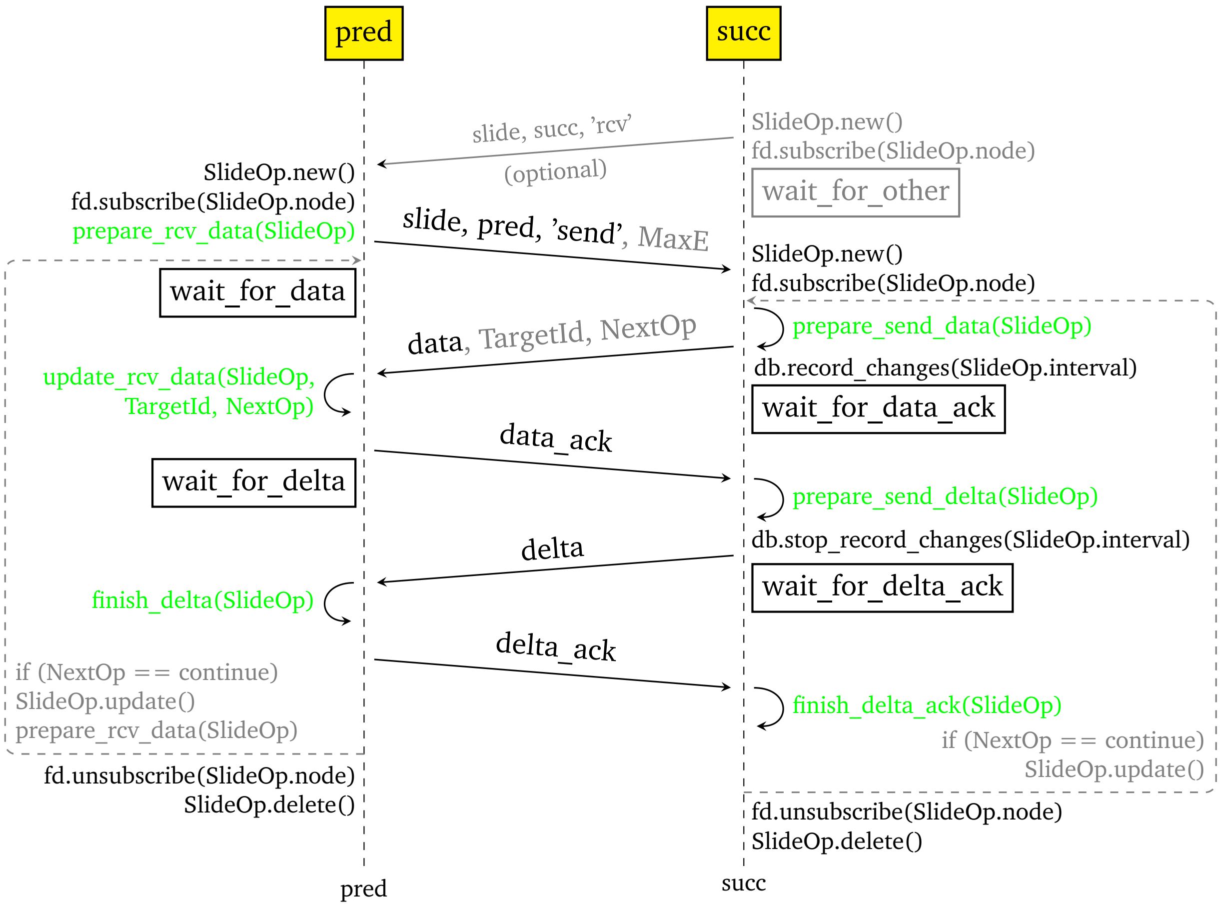
In general, a slide consists of three steps:

1. set up slide
2. send data & start recording changes, i.e. delta
3. send delta & transfer responsibility

The latter two may be repeated to execute incremental slides which further reduce periods of unavailability. During this period, no node is responsible for the range to transfer and messages are thus delayed until the receiving node gains responsibility.

12.1. Sending data to the predecessor

12.1.1. Protocol

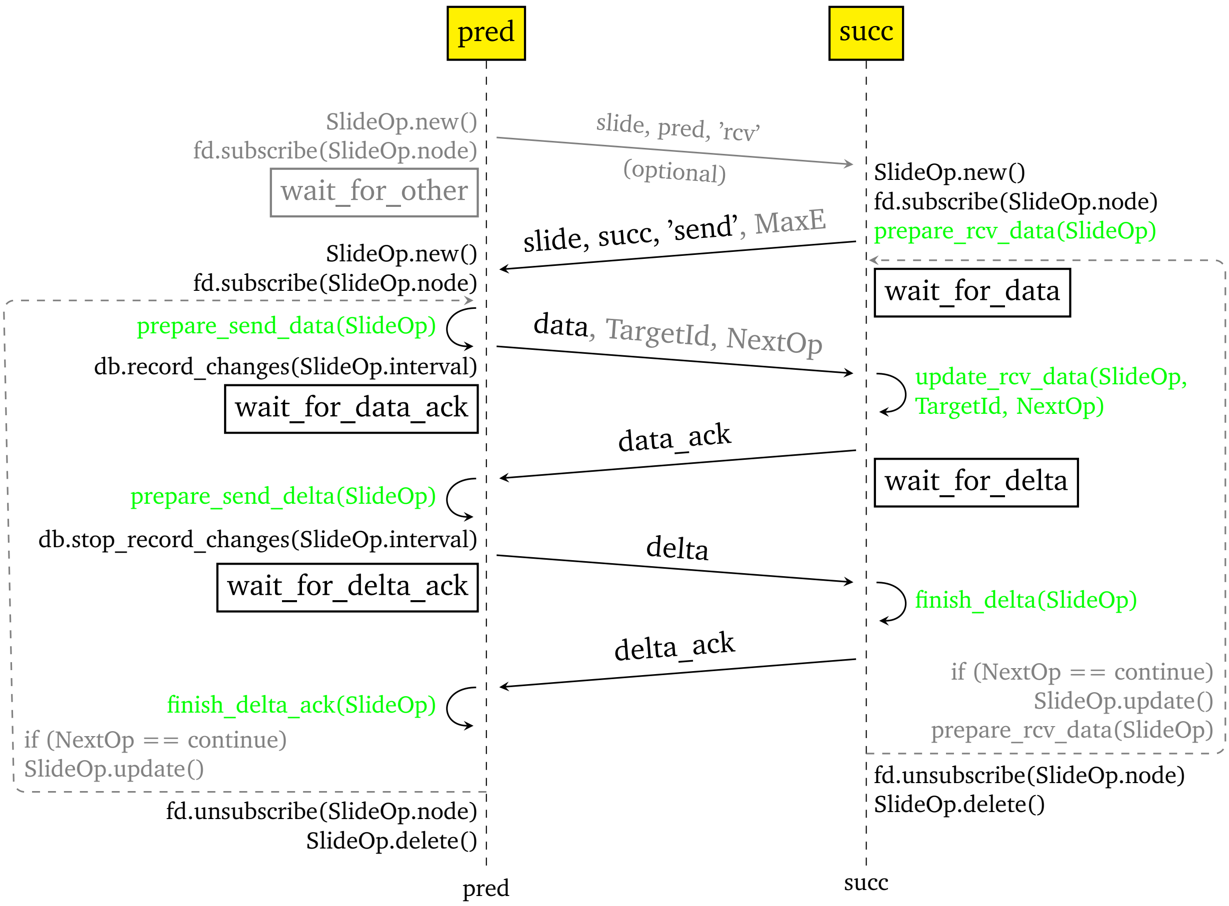


12.1.2. Callbacks

	slide_chord	slide_leases
← **prepare_rcv_data**	*nothing to do*	*nothing to do*
→ **prepare_send_data**	add DB range	*nothing to do*
← **update_rcv_data**	set MSG forward, change my ID	*nothing to do*
→ **prepare_send_delta**	wait until pred up-to-date, then: remove DB range	split own lease into two ranges, locally disable lease sent to pred
← **finish_delta**	remove MSG forward	*nothing to do*
→ **finish_delta_ack**	*nothing to do*	hand over the lease to pred, notify pred of owner change

12.2. Sending data to the successor

12.2.1. Protocol



12.2.2. Callbacks

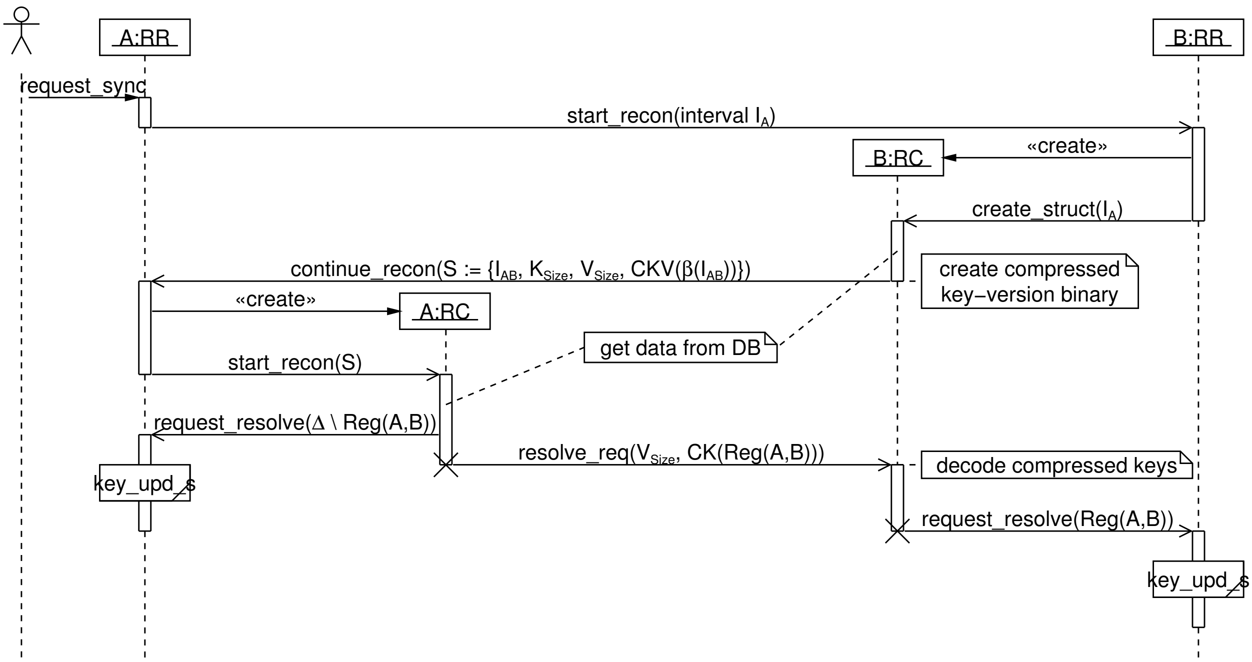
	`slide_chord`	`slide_leases`
→ **prepare_rcv_data**	set MSG forward	*nothing to do*
← **prepare_send_data**	add DB range, change my ID	*nothing to do*
→ **update_rcv_data**	*nothing to do*	*nothing to do*
← **prepare_send_delta**	remove DB range	split own lease into two ranges, locally disable lease sent to succ
→ **finish_delta**	remove MSG forward, add DB range, wait until pred up-to-date then: remove DB range	*nothing to do*
← **finish_delta_ack**	*nothing to do*	hand over the lease to succ, notify succ of owner change

13. Replica Repair

13.1. Replica Reconciliation - `rr_recon`

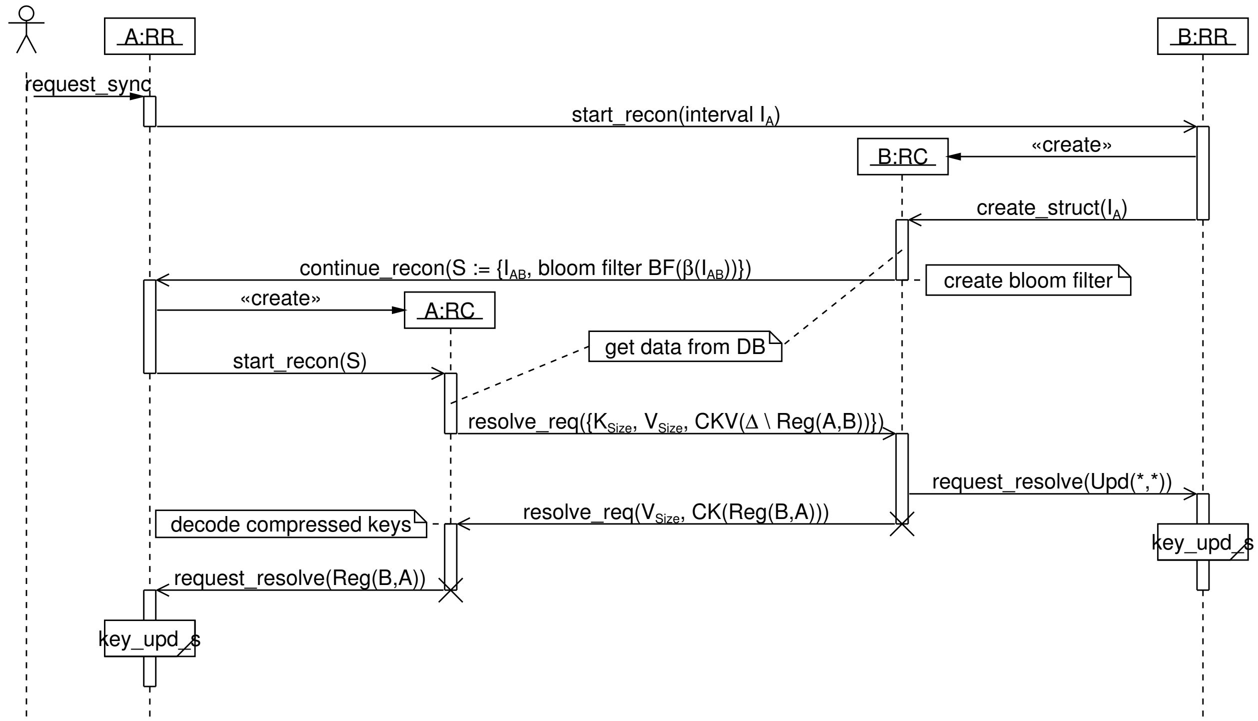
13.1.1. Trivial Replica Repair

Protocol



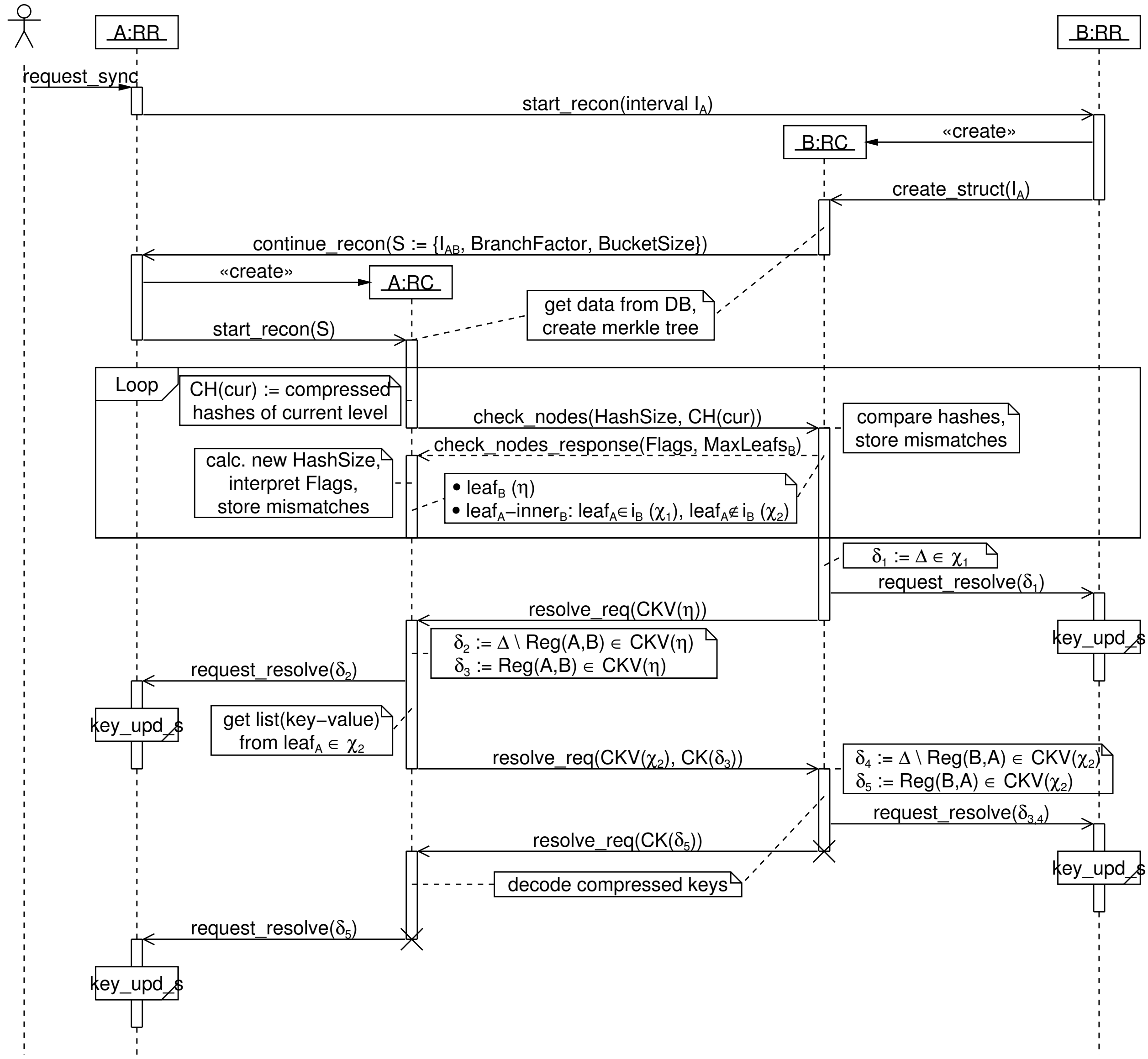
13.1.2. Replica Repair with Bloom Filters

Protocol



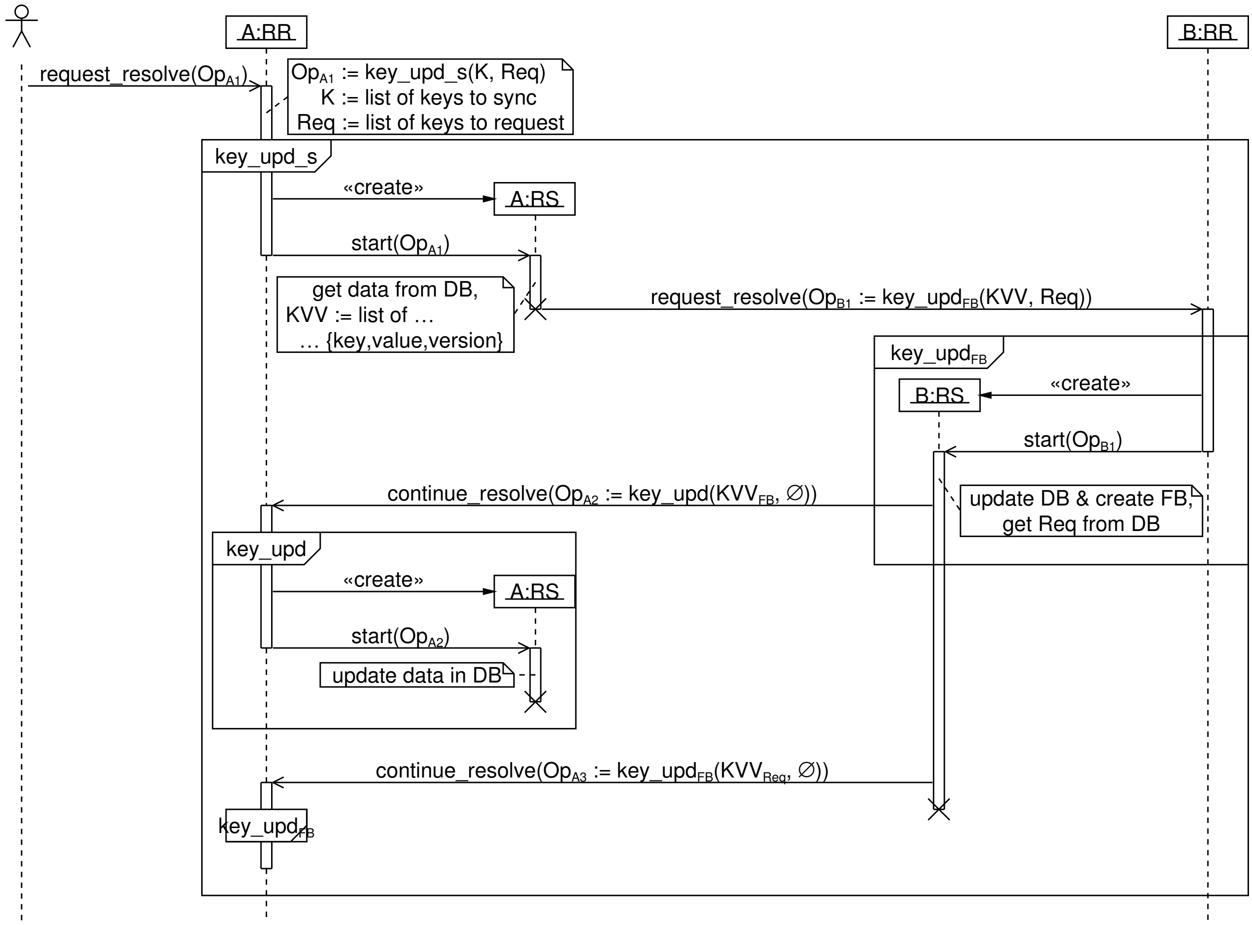
13.1.3. Replica Repair with Merkle Trees

Protocol



13.2. Resolve Replicas - `rr_resolve`

13.2.1. Updating a list of keys - `key_upd`



14. 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 management server, 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
`python-api`	a Python 2 API to Scalaris
`python3-api`	a Python 3 API to Scalaris
`ruby-api`	a Ruby API to Scalaris
`log`	log files
`src`	contains the Scalaris source code
`include`	contains macros for the source code
`test`	unit tests for Scalaris
`user-dev-guide`	contains the sources for this document

15. 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).

The API is divided into four classes:

- `de.zib.scalaris.Transaction` for (multiple) operations inside a transaction
- `de.zib.scalaris.TransactionSingleOp` for single transactional operations
- `de.zib.scalaris.ReplicatedDHT` for non-transactional (inconsistent) access to the replicated DHT items, e.g. deleting items

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

[6] Hiroya Nagao, Kazuyuki Shudo. *Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set.* In: Peer-to-Peer Computing, IEEE, 2011.

[7] 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.

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

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

Index