in `java-api/scalaris.properties`.

4.2.2. Python command line interface

```
%> ./python-api/scalaris_client.py --help
```

```
usage: ../python3-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.
 -p,--publish <topic> <message>
                            publish a new message for the given topic
 -s,--subscribe <topic> <url>
                            subscribe to a topic
 -g,--getsubscribers <topic>
                            get subscribers of a topic
 -u,--unsubscribe <topic> <url>
                            unsubscribe from a topic
 -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_client.rb --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. Note that the command above will produce some output about unknown nodes. This is expected, as some nodes defined in the configuration file above don't exist 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 by pointing your browser to `http://127.0.0.1:8000` (or the respective IP adress 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 restart 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.

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()..

```
(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 atomical 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 on the client side using `api_tx:req_list(Tlog, List)`.

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 a `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}`. The individual separation of all reads, writes and commits was done here on purpose to study the transactional behaviour.