Implementation document

Version 1.2

May 4, 2001

RTD-Log

Dan Patrick Doepel
Matti Johansson
Marko Moisio
Juha Saarelainen
Minna Vasankari

University of Helsinki
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 UNIVERSITY OF HELSINKI

Contents

1 Introduction

1.1 Background and purpose of the system

The system implemented by the RTD-Log project is a new log handling subsystem for the distributed Rodain 2000 Real-Time Database System that is being studied and developed by the RTD-Pilot project. The aim in implementig this system was simply make the recovery more efficient than before in case of other unit fails. This has been achieved by using index that keeps up information on each log record written to SSS.

1.2 Terminology

The abbreviations and terms used in this document has been listed in the Table 1.

1.3 Overview of the document

This document gives a description about how the Log system for Rodain 2000 Real-Time Database Prototype is implemented. Detailed planning documentations for implenting the Log system can be found from the Design document.

Implementation solutions of the Log system has been described in Section 2. Section 3 outlines the module testing that has been done for the different modules of the Log system. Section 4 describes testing that has been done for the whole Log system. Future development of the system has been explained in Section 5.

2 Implementation solutions

This section gives an overview of the implementation of the Log system. First is discussed how and when the system is started, stopped and controlled and what kind of errors can stop the system. The used programming languages and tools are reported then and after that the file structure of the system is described with instructions about compiling. The modules' internal design has been described in the design document, but the implementation matters may force some minor modifications to the design. They are collected to the

SSS	Secondary Storage Subsystem, the disk-based permanent data storage
PU	Primary Unit, the active database unit that processes the clients' requests
MU	Mirror Unit, the secondary unit that maintains a copy of the main memory database and stores the log to the SSS
TU	Transient Unit, the remaining unit after either PU or MU has failed
RU	Recovering Unit, the failed unit that is recovering to become a Mirror Unit
Log mediator	The part of the already implemented RODAIN 2000 Database Prototype that transmits log records to Log system
Log system	The system this project is about to implement
Log manager	The Log system excluding SSS
RM	Recovering Module, the software module in the Log manager that is responsible of the recovering
SM	Storage Module, the software module in the Log manager that is responsible of the communication to SSS
CM	Control Module, the software module that is responsible of the general logic of the Log Manager
IM	Index Module, the software module that maintains the index structure
ISM	Image Saving Module, the software module that is responsible of saving the image of the Mirror Unit's database to SSS
LSM	Log Saving Module, the software module that is responsible of saving the log records to SSS
DB	Database, the main memory database inside the unit
DBMS	Database Management Subsystem, the main subsystem on Primary and Transient Unit which maintains the database, real-time constraints of transactions and concurrency control
FTRS	Fault-Tolerance and Recovery Subsystem, the system in the current Rodain Database Prototype that process the log records and handles most of the operations in the role switching

Table 1: The abbreviations and terms used in this document

final subsection.

2.1 High-level functionality of the system

Log system is controlled by the Fault-Tolerance and Recovery Subsystem (FTRS). FTRS starts the Log system by creating a thread for it and calling the *start_log_manager* function in the creation. Log system runs then until FTRS shuts it down by sending a signal (SIGUSR2), a fatal error occurs or power fails.

FTRS and Log system share a variable which contains the unit's state. The state is either RECOVERY or TRANSIENT when Log system is started. If the state is RECOVERY, Log system starts recovering. It sets the state to INDEX_LOADED when the index has been retrieved from the SSS and to INDEX_READY when the index has been updated to the situation it was in when the failure occurred. After the state has been set to INDEX_LOADED FTRS can turn the log flow and switch the state to MIRROR or TRANSIENT. If it hasn't changed the state yet when recovery becomes complete, Log system updates the state to DB_READY. If the state is TRANSIENT when Log system starts, it just runs until FTRS tells it to shut down by sending a signal.

There are three kinds of fatal errors that cause the Log system to stop and exit. TCP communication errors like failure in opening a socket form the first category. If connection to the SSS fails, it is considered a fatal error.

Another category of fatal errors are the ones caused by failed system calls: file handling, memory allocation, signalling etc. Rest of the errors are internal errors that should never occur. An example is that Control Module would call the Log Saving Module with *log_write_count* parameter of -1 which makes no sense.

2.2 Used programming languages and environments

The system is written in C. It is compiled and tested in the Computer Science Department's Linux system using gcc compiler.

2.3 File structure of the system

This section describes the directory and file structure of the system at the moment the system is handed over to the customer. The directory structure is depicted in Figure 1 and after that the purpose of each file is explained shortly.

All the project documentation and source codes are stored to CVS. The project name in the CVS is *rtd-log* and the CVSROOT environment variable has to be set to /home/group/rtd-log/cvsroot.

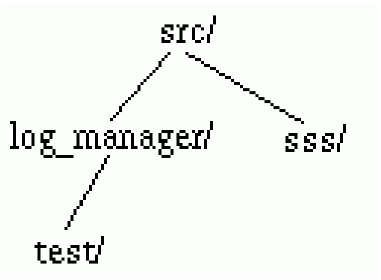

Figure 1: The directory structure of the Log system

src/

LS_globals.h	Global definitions and structures of the Log system
Makefile	Makefile of the Log system

log_manager/

CM.h, CM.c	The introduction and implementation of the Control Module
IM.h, IM.c	The introduction and implementation of the Index Module
ISM.h, ISM.c	The introduction and implementation of the Image Saving Module
LM_dprint.h	DPRINT message printing macro of the Log manager
LM_globals.h	Global definitions and structures of the Log manager
LSM.h, LSM.c	The introduction and implementation of the Log Saving Module
LogManager.h	The introduction of the Log manager
Makefile	Makefile of the rtdtest test program
RM.h, RM.c	The introduction and implementation of the Recovering Module
SM.h, SM.c	The introduction and implementation of the Storage Module

STA.h, STA.c	The introduction and implementation of the module which controls the usage of the system's state variable between Log manager and FTRS
rtdtest.c	The implementation of the test program which is used to test the Log manager
rtdtest.conf	The configuration file for the rtdtest program
rtdtest.h	The introduction of the database and log buffer test interfaces implemented in rtdtest.c.
utils.h, utils.c	The introduction and implementation of some utility functions
test/	
Makefile	Makefile for the SM test programs
SM_test_client.c	The client side of the test program that is used to test the Storage Module
SM_test_server.c	The server side of the test program that is used to test the Storage Module
lsmtest.h, lsmtest.c	The introduction and implementation of the test stub functions used to test the ISM and LSM.
sss/	
Makefile	Makefile for the SSS
SSS_dprint.h	DPRINT message printing macro for the SSS
init_sss.c	Implementation of a program to initialize the SSS.
show_index.c	Program for viewing contents of the active index file.
show_active_index.c	Program that tells which index file is active.
show_logs.c	Program for viewing contents of the log files (test purpose mainly)
sss.h, sss.c	The introduction and implementation of the SSS

2.4 Compile and execution directions

This subsection gives instructions about how the system should be compiled.

2.4.1 Log Manager

The Makefile in the folder /src/log_manager creates rtdtest program. This program is used to test the Log manager. rtdtest program emulates FTRS. It contains the database and log buffer and it creates the thread where the Log manager is executed. More information about rtdtest program can be found in the section 3.1.1.

By default the current Log manager continues an image write in the Transient state, if it was started in the Mirror state. This feature can be disabled by adding a DO_NOT_FINISH_IMAGE_SAVING compile flag in the Makefile.

Debug print macros are included only if program is compiled with DEBUG flag.

2.4.2 SSS

The SSS (/src/sss folder) is compiled with command make. If TEST_COMPILE is defined in the code (in beginning of sss.c or in sss.h), some test functions will be added during compilation.

All .c -files (sss.c, ini_sss.c, show_logs.c, show_index.c) are compiled when using make command. The "make clean" command will remove all object files (.o -files), executable files, and all .rtd -files (log-, index-, and other needed files used by the SSS).

Before the SSS can be started the init_sss must be run. This program will create needed files and initialize the file status information. Init_sss will destroy any existing data! This implementation solution was chosen beacause if a file is missing or file status information is corrupted the database would, during recovery, build a corrupted database.

After the init_sss is run, the SSS can be started with command: sss.

If TEST_COMPILE was defined before compilation, the test mode is available. The SSS will still run normally if started with command: sss . If "test" is given as program parameter (: sss test), the SSS is started in test mode lacking any tcp communication. All input (output) will then be read (written) from (to) the console.

Debug print macros in the sss program are included only if the program is compiled with DEBUG flag.

2.5 Adjustments to the module design

DB_ prefix was added in the return codes of Database functions. LB_ prefix was added in the return codes of Log buffer functions.

The interface of function DB_image_read was modified. It is now as follows:

```
DB_image_read{
  trid      OUT
  oper      OUT
  toid      OUT
  istype    OUT
  oid       IN
  len       IN/OUT  (This was changed from IN to IN/OUT)
  target    OUT
}
```

New variable (*second_recovery_offset_set*) was added to LM_info structure defined in /src/log_manager/LM_globals.h.

3 Produced executables

In this section has been documented all the executables this project has produced.

3.1 Test stubs

3.1.1 rtdtest

rtdtest program contains the log buffer. It initializes it and generates new log events along the way. It also contains the database and if the program is started in the Transient state, it also initializes the database.

Log records are very simple. The values of OIDs are between one and the size of the database. Transaction Identifiers (TRIDs) are used as time stamps. The program saves the value of next trid in the file next_trid so the time stamp values in the new records are allways greater than in the earlier ones even if the rtdtest program is restarted.

After rtdtest has initialized the log buffer and possibly the database it creates a thread where the Log manager is executed. After that rtdttest follows the global state variable (used with function STA_set_state and STA_get_state) and acts acording to that state. When rtdtest wants to stop the execution of the Log manager it sends signal SIGUSR2 to Log manager. After rtdtest notices that Log manager has exited it also exits.

The behaviour of rtdtest program can be configured with configuration file rtdtest.conf. All the configuration options has been commented in the file.

3.1.2 lsmtest and ismtest

The lsmtest.c test module was used in module testing of the Image Saving Module and Log Saving Module. It contains test stub functions that have the same interface as the real functions that ISM and LSM call. But the test stubs simply return the same value every time they are called. Default return values are SUCCESS or otherwise normal values. They can be changed by giving function names and the new values as command line parameters. For example *lsmtest BUFREAD=LB_EMPTY* runs the LSM so that the buffer is empty at the first attempt to read it. lsmtest and ismtest are compiled from the lsmtest.c module with the command *make lsmtest* or *make ismtest*.

3.1.3 sm_test_client

This test stub was used to test the Storage Module. It has no configuration options. It just sends different kinds of messages to sm_test_server and receives an answer to every request it has sent. It prints to stdout information how the test is proceeding.

3.1.4 sm_test_server

This test stub was used to test the Storage Module. It has no configuration options. It just receives different kinds of messages from sm_test_client and sends an answer to every request it has received. It prints to stdout information how the test is proceeding.

3.2 Utilities

Following utility programs mainly used in testing are located in the src/sss/ folder and compiled with the make command executed in same folder (see compile instructions).

show_logs program, which prints the contents of the log files, started with command: show_logs.

show_index program, which prints the contents of the active index file, is started with command: show_index.

4 Module testing

4.1 Control Module

4.1.1 Testing method

Control Module was tested only in the integration test phase and was not tested separately. It uses all the other modules so it would have been very difficult to create a test stub that would have emulated all the modules in a right way. And because Control Module contains very little inner logic but uses most of the time calling other modules the testing was decided to do in the integration test phase. More information about the integration testing can be found from the section 5.

4.1.2 Testing material and test stubs

See section 5.

4.1.3 Detected and fixed errors

See section 5.

4.2 Index Module

4.2.1 Testing method

Because the Index Module has such a central role in Log Manager, it is important to make sure that all the interface functions and internal data structures of the module are working as expected. If the Index Module isn't working correctly, it will certainly generate a lot of errors during the integration testing.

On the other hand, this central role will also guarantee that the Index Module will be tested thoroughly during the integration testing. Since the index is used by all the other modules of the Log Manager and it's used practically in all situations (except in transient mode), it's clear that even the smallest error in Index Module should be spotted easily.

4.2.2 Testing material and test stubs

A small stub-program which offers the functionality of SM was created for testing purposes. This stub offers SM_index_write and SM_index_read functions similar to those offered by the Storage Module, with the exception that the functions of the stub operate directly with the local disk. This stub was needed to test writing and reading the whole index-structure.

Internal functions as well as interface functions of the Index Module were tested simply by creating a main program that mimics the actions of the Log Manager in different situations (different interface functions were called sequentally from the main program). There is a very useful function called IM_show_index that was added to the Index Module for test and debugging. When called, the IM_show_index will print the contents of the index structure. This function was used to monitor the state of the index in every step of the module testing. This way it was easy to verify that the index structure was working correctly, and that calling the interface functions had the desired effects.

Internal data structures and data integrity were given a special attention. For this purpose, a test case was set up where the index was shot with 500 million consecutive write-operations with random numbers (within a range of 1 million). This sort of test produces a comprehensive results on data integrity and memory management of the index structure.

4.2.3 Detected and fixed errors

While testing the data integrity it was noticed that there were duplicates of some oid's in the index structure. This is clearly an error and it was soon found that each time the index structure was reallocated (to make more room for overflow-list), the first record of the newly acquired space was lost. That record was correctly filled with data concerning the oid that was currently being processed, but the next operation on the same oid caused another insert operation insted of update.

The cause of the error was incorrect setting of internal pointer in reallocate_index function. The error was fixed and the data integrity test was repeated, with successful results.

No other errors were found.

4.3 Log Saving Module

4.3.1 Testing method

The white-box testing was carried out as basic path testing. First such testing material was figured out that would reach every part of the code. Then the tests were run and the results were compared to expected results. This module mostly calls other modules, so the testing is carried out by using test stub functions in place of the real functions. These test functions then return values determined in advance. LSM prints messages as it proceeds and from those messages can be seen if right actions are taken and correct return values are returned.

4.3.2 Testing material and test stubs

The test stub functions are collected to lsmtest module. This test stub can be compiled with the Makefile in the *src/log_manager/test* directory. The module contains the main function that starts the LSM in place of the CM. By default the test stub functions return SUCCESS or otherwise normal values. The return values can be changed by giving the new values as command line parameters. The format is:

```
lsmtest FUNCTIONNAME=VALUE...
```

Tested situations:

- LSM was called with *log_write_count* = -1. LSM returns INVALID_PARAMETER.
- LSM was called with *log_write_count* = 0. LSM returns INVALID_PARAMETER.

- LSM was called with *log_write_count* = 1; other values normal. In rest of the tests LSM is called with *log_write_count* = 1 if not otherwise mentioned.
- LSM was called with *blocks* = 0 or 1, other values normal.
- BUF_read returns LB_BUF_EMPTY when LSM tries to read the first log record. LSM returns SUCCESS.
- BUF_read returns LB_OVERFLOW when LSM tries to read the first log record. LSM returns ERROR.
- STA_get_state returns INDEX_LOADED and LM_info.records_for_image is 0. This means the unit is recovering and the first log record has arrived. Now LSM must get the fresh file_status information and update the second_recovery_offset before continuing.
- STA_get_state returns INDEX_LOADED and LM_info.records_for_image is 1. Normal log handling continues.
- STA_get_state returns INDEX_READY and LM_info.records_for_image is 0. File status information is fetched and second_recovery_offset updated.
- STA_get_state returns INDEX_LOADED and LM_info.records_for_image is 0, but SM_file_status_retrieve returns ERROR. LSM returns ERROR.
- STA_get_state returns INDEX_LOADED and LM_info.records_for_image is 0, but SM_file_status_retrieve returns SM_TCP_ERROR. LSM returns SM_TCP_ERROR.
- STA_get_state returns TRANSIENT, other values normal. Normal log handling continues, but database is not updated. In rest of the tests STA_GET_STATE returns MIRROR.
- DB_log_write returns DB_INVALID_OID. Log record is just discarded and new one is read. LSM is called with *log_write_count* = 2.
- DB_log_write returns DB_INVALID_PARAMETER. Log record is just discarded and new one is read. LSM is called with *log_write_count* = 2.
- DB_log_write returns DB_OID_EXISTS. Log record is just discarded and new one is read. LSM is called with *log_write_count* = 2.

- DB_log_write returns DB_OID_NOEXISTS. Log record is just discarded and new one is read. LSM is called with *log_write_count* = 2.
- SM_log_write returns ERROR. LSM returns the same.
- SM_log_write returns SM_TCP_ERROR. LSM returns the same.
- LSM is called so that index_present is 0. Index is not updated. In rest of the tests index_present is 1.
- LSM deletes a record from the index and IM_delete_index succeeds.
- LSM deletes a record from the index and IM_delete_index returns INVALID_PARAMETER. LSM continues.
- LSM deletes a record from the index and IM_delete_index returns ERROR. LSM returns the same.
- LSM updates a record in the index and IM_write_index returns INVALID_PARAMETER. LSM continues.
- LSM updates a record in the index and IM_write_index returns ERROR. LSM returns the same.
- BUF_clear returns INVALID_PARAMETER when LSM tries to clear a record. LSM continues.
- For the following two tests LSM was called with *log_write_count* = 2. The stub function was temporarily modified to return SUCCESS at the first time it is called and the defined error value at the second time.
  - BUF_read returns LB_BUF_EMPTY when LSM tries to read the next log record. LSM returns SUCCESS.
  - BUF_read returns LB_OVERFLOW when LSM tries to read the next log record. LSM returns ERROR.
- LSM called 5 times; checked that 5 log records are handled and offsets, records_for_image and records_for_index are updated correctly.

4.3.3 Detected and fixed errors

LSM asked for a record from the buffer once although the *log_write_count* had already been reached. This was fixed by making LSM check if the limit was reached before asking the next record.

LSM stopped and returned INVALID_PARAMETER if IM_delete_index or IM_write_index returned that value. Anyway this is not a fatal error so now LSM just prints an error message and continues normally.

4.4 Image Saving Module

4.4.1 Testing method

ISM was tested the same way as the LSM. Basic path testing was done by using test stub functions and printed messages were compared to expected results.

4.4.2 Testing material and test stubs

The test stub functions are collected to ismtest module. This test stub can be compiled with the Makefile in the *src/log_manager/test* directory. The module contains the main function that starts the ISM in place of the CM. By default the test stub functions return SUCCESS or otherwise normal values. The return values can be changed by giving the new values as command line parameters. The format is:

```
ismtest FUNCTIONNAME=VALUE...
```

Tested situations:

- ISM was called with *image_write_count* = -1. ISM returns INVALID_PARAMETER.
- ISM was called with *image_write_count* = 0. ISM returns INVALID_PARAMETER.
- ISM was called with *image_write_count* = 1; other values normal. In rest of the tests ISM is called with *image_write_count* = 1 if not otherwise mentioned.
- *image_complete* = 1; other return values normal. ISM switches active and passive files and sends an IMAGE_START record to SM.
- *image_complete* = 1; SM_image_write returns ERROR when ISM tries to write the IMAGE_START record. ISM returns ERROR too.

- *image_complete* = 1; SM_image_write returns SM_TCP_ERROR when ISM tries to write the IMAGE_START record. ISM returns SM_TCP_ERROR too.
- *image_complete* = 0; other values normal.
- IM_read_nextindex_ISM returns ERROR when ISM tries to get the first oid. ISM returns the same.
- IM_read_nextindex_ISM returns IM_END when ISM tries to get the first oid. ISM returns ISM_IMAGE_COMPLETED.
- DB_image_read returns DB_INVALID_OID when ISM tries to read an image record. ISM continues by asking another oid.
- DB_image_read returns DB_OID_NOEXISTS when ISM tries to read an image record. ISM continues by asking another oid.
- DB_image_read returns DB_OVERFLOW when ISM tries to read an image record. ISM continues by asking another oid.
- SM_image_write returns ERROR when ISM tries to write an image record. ISM returns the same.
- SM_image_write returns SM_TCP_ERROR when ISM tries to write an image record. ISM returns the same.
- IM_write_index returns ERROR when ISM tries to update a record. ISM returns the same.
- IM_write_index returns INVALID_PARAMETER when ISM tries to update a record. ISM continues.
- For the following two tests ISM was called with *image_write_count* = 2. The stub function was temporarily modified to return SUCCESS at the first time it is called and the defined error value at the second time.
  - IM_read_nextindex_ISM returns ERROR when ISM tries to get the next oid. ISM returns the same.
  - IM_read_nextindex_ISM returns IM_END when ISM tries to get the next oid. ISM returns ISM_IMAGE_COMPLETED.
- SM_image_write returns ERROR when ISM tries to write an IMAGE_END record. ISM returns the same.

- SM_image_write returns SM_TCP_ERROR when ISM tries to write an IMAGE_END record. ISM returns the same.
- ISM called 5 times; checked that 5 image records are handled and offsets are updated correctly.

4.4.3 Detected and fixed errors

ISM asked for an oid from the index once although the *image_write_count* had already been reached. This was fixed by making ISM check if the limit was reached before asking the next oid.

ISM stopped and returned INVALID_PARAMETER if IM_write_index returned that value. Anyway this is not a fatal error so now ISM just prints an error message and continues normally.

4.5 Recovery Module

4.5.1 Testing method

Recovery Module was tested only in the integration test phase and was not tested separately. It uses a number of other modules so it would have been very difficult to create a test stub that would have emulated all the modules in a right way. More information about the integration testing can be found from the section 5.

4.5.2 Testing material and test stubs

See section 5.

4.5.3 Detected and fixed errors

See section 5.

4.6 Storage Module

4.6.1 Testing method

The goal of the Storage Module's testing was to make sure that it implements the communication protocol correctly and that all data is delivered correctly to SSS. Every API function of the Storage Module and every protocol message was tested during this test.

4.6.2 Testing material and test stubs

Two test stubs were used in this test. The first one used the Strorage Module and the other one was emulating the SSS.

The client stub sent different kinds of request and messages to server stub and the server stub then send an answer to every request it received. Here is listed the output of the client and server test stubs.

Output of the client test stub:

```
Initializing Storage Module
Storage Module initialized succesfully
```

The creation of TCP connection was succesfull.

```
Retrieving file status information:

  FILE STATUS INFORMATION
  index         : 1
  active log    : 2
  passive log   : 3
  file status   : 4
  image complete: 1
  index stored  : 0
  active size   : 100
  passive size  : 200
  index size    : 300
```

The content of the file status information was correct.

```
Sending log record
Log record sent succesfully

Sending index structure
Index structure sent succesfully
```

Succesfully sent an index of size 1000000 bytes.

```
Retrieving index structure
Index structure received
Content of index structure was valid
```

Succesfully received index of size 1000000 bytes. The content of the index structure was correct.

```
Send image record: IMAGE_START
Image record sent succesfully

Send image record: IMAGE_RECORD
Image record sent succesfully

Send image record: IMAGE_END
Image record sent succesfully

Send log read request
Log record read request sent succesfully

Reading log record:

  LOG RECORD
  log trid    : 1
  log oper    : 2
  log oid     : 3
  log toid    : 4
  log istype  : 5
  log data    : Log record
```

The content of the log record was correct.

```
Finishing Storage Module
```

Succefully closed the connection to test server.

Output of the server test stub:

```
NEW CONNECTION

Message received

action      : SM_FILE_STATUS
file id     : 0
data length : 0
offset      : 0


Message received

action      : SM_LOG_WRITE
file id     : 10
data length : 280
offset      : 0
log trid    : 1
log oper    : 2
log oid     : 3
log toid    : 4
log istype  : 5
log data    : Log record


Message received

action      : SM_INDEX_WRITE
file id     : 20
data length : 1000000
offset      : 0
Content of index structure was valid


Message received

action      : SM_INDEX_READ
file id     : 30
data length : 0
offset      : 0
```

```
Message received

action      : SM_IMAGE_START
file id     : 40
data length : 0
offset      : 0


Message received

action      : SM_LOG_WRITE
file id     : 50
data length : 280
offset      : 0
log trid    : 1
log oper    : 2
log oid     : 3
log toid    : 4
log istype  : 5
log data    : Log record


Message received

action      : SM_IMAGE_END
file id     : 60
data length : 0
offset      : 0


Message received

action      : SM_LOG_READ
file id     : 70
data length : 280
offset      : 1000
```

Every message was received correctly.

```
Storage Module closed connection
```

All the messages tested was sent and received correctly.

The test stubs can be found in the CVS under rtd-log/src/log_manager/test. Makefile creates two executables, sm_test_client and sm_test_server, that has been used to do these tests. More information about the test stubs can be found in the section 3.1.

4.6.3 Detected and fixed errors

No errors were found.

4.7 Secondary Storage Subsystem (SSS)

4.7.1 Testing method

The testing of the SSS was carried out during the whole implementation phase. The SSS was implemented by gradually adding more functionality in form of new functions to carry out a specific task. These added functions where then tested by invoking the part of the SSS that uses the function. It was then easy to see if the added function behaved correctly because the rest of the implemented software already had been tested and new possible errors where therefore considered to be caused by the added function.

Testing of tcp-communication with SM was carried out in the integration phase. Because the logic of the SSS had already been tested, the testing of the tcp-communication was restricted to the creation of a listening socket and sending and receiving data correctly.

4.7.2 Testing material

The functionality of the SSS was carried out by gradually adding functionality and testing it, as mentioned in previous subsection.

The main function to handle the logic on the SSS was implemented basically as a switch statement, examing the action given as input. The test input to the SSS was (in the beginning) supplied as program parameters, except the data which was set to a default value.

Execution of SSS look something like this in the beginning: sss x y

where

- x stands for action:

  - file status information request
  - log read
  - log write
  - index read
  - index write
  - image start
  - image end
  - default (if action not recognized)

- y stands for file identifier:
  - 0 - for index file
  - 1 - for active log file
  - 2 - for passive log file
  - 3 - for file status information file (later not used)

The values for action are defined in sss.h.

The log-, index- and file status files could be examined with a normal editor to ensure that data was correctly saved.

The SSS module now contains possibility to start the SSS in a test mode. By defining TEST_COMPILE the test functions will be included in the compilation. The test mode can then be started by giving "test" as program parameter (prompt$ sss test), and the logic of the SSS can be tested manually.

When the logic of the SSS had been tested completely using the method described above, the next stage was integration testing to validate the functionality of the tcp-communication. This stage was quite straight forward, requiring only some fine tuning to receive and send data correctly.

4.7.3 Detected and fixed errors

The SSS was continuesly tested which reduced the errors found. Some errors where found during implementation. The errors where though minimal and easy to correct. The most important found errors are listed below:

- Log-, index- and file status files must exist before starting the SSS

  - A small program, init_sss, was implemented and should be run before starting the sss. The init_sss program creates needed file (truncates them if they exist), and sets default values to the file status.

- The SSS run out of memory if running for a long time

  - Freeing allocated memory had been forgotten in some places. This was corrected by freeing the allocated memory after use.

5 Integration and validation testing

5.1 Test cases

```
Test environment:
Database size: 51
OID range: 1 - 51
delete operations included
```

```
TEST CASE 1:
LM starts in Transient state
SSS is empty
```

The LM is run for a predefined time (5 seconds). The state of the database and the log files are then examined. This way can be ensured that the program is functioning correctly in Transient state.

Test passed OK.

```
TEST CASE 2:
LM starts in Transient state
SSS is not empty
```

Same procedures as in test case 1.

Test passed OK.

```
TEST CASE 3:
LM starts in Recovery state
State after recovery is Mirror
SSS is empty
```

This test is carried out to ensure that the LM works correctly if SSS is empty (initial startup).

> *This test passed OK. The database seemed to match the log files on SSS. Deleted record was deleted from the database and index too. If a record was written to the active log file as a normal log record, it was not written again as an image record.*

```
TEST CASE 4:
LM starts in Recovery state
State after recovery is Mirror
SSS is not empty
```

Test case 3 can be used to brought the SSS to a suitable state where image and index has been stored. This test should ensure that the database and index are brought up to date in recovery phase using the stored index.

> *Test passed OK.*

```
TEST CASE 5:
LM starts in Recovery state
State after recovery is Mirror
SSS is not empty
```

Test case 3 can be used to brought the SSS to a suitable state where image but not index has been stored. This test should ensure that the database and index are brought up to date in recovery phase using only the stored image in the active log file.

> *Test passed OK.*

```
TEST CASE 6:
LM starts in Recovery state
State after recovery is Transient
SSS is not empty
```

Same as test case 4, except state after recovery is now Transient.

> *Test passed OK.*

```
TEST CASE 7:
LM starts in Recovery state
State after recovery is Mirror
State is changed instantly when index is up to date
SSS is not empty
```

Same as in test case 4, but the state is changed to Mirror when index is up to date (not when database is up to date).

Test passed OK.

```
TEST CASE 8:
LM starts in Recovery state
State after recovery is Mirror
SSS is not empty
```

In this test the active log file should not contain complete image when starting the LM. It this test it can be checked that in the recovery the LM uses the passive log file when it notices that the active file doesn't contain complete image. This test should also ensure that the LM is able to continue incomplete image saving. Test situation can be created by exiting the LM during image save in test case 3.

Test passed OK.

```
TEST CASE 9:
LM starts in Recovery state
SSS is not empty
```

During the recovery state another LM generates log records to the SSS while the LM being tested is recovering. The purpose of this test is to ensure that the SSS can handle concurrent connections, and that RM can handle the situation when new log records arrive from another LM during recovery. The situation can be forced by adding a sleep in the tested LM during recovery, while another LM is run in transient state.

Test passed OK.

```
TEST CASE 10: ENDURANCE TEST
Test environment:
Index size: 10001
Database size: 12000
OID range: 1 - 12000
LM starts in Recovery state
State after recovery is Mirror
SSS is empty
```

The purpose of the endurance test was to make sure that the system will run smoothly over extended period of time. First this test was set up to last 48 hours, but the system crashed after only 8 hours. This was caused by a memory leak in SM.c.

Test 1. failed.

After the memory leak was fixed, the endurance test was repeated but this time we were aiming for 96 hours. This time everything went fine.

Test passed OK.

Finally the whole system (rtdtest test stub and sss) where checked with memory debugger (dmalloc). No memory leaks were found.

5.2 Detected and fixed errors

Writing the INDEX_STORED record did not update the offset of the active file so the offset was more and more behind the actual offset every time index was saved. This was fixed.

When image records were stored the length of the data part was set to zero. This caused sometimes problems in the recovery and because of that the data part of the record was lost. This was fixed.

A memory leak bug was found in the Storage Module (SM.c) as a result of the endurance test. This bug was fixed.

6 Future development

6.1 Instructions for further development

All the features that were discussed in the design document were implemented. The handling of delete records were also implemented even though the current RODAIN prototype doesn't generate those. In the next section has been summarized those features that could be added later on to this Log system.

6.1.1 Not implemented features

Although the handling of the delete records was implemented in RTD-Log, there is one area of development on delete's: Current implementation doesn't remove deleted records from the index. When a record is deleted it is marked as deleted in the index, but the entry for the deleted record will remain. This was considered to be sufficient approach, because deletes are really not supported in the current RODAIN prototype. To complete the handling of delete records, a sort of clean-up routine should be implemented for the index.

One feature that was considered in the planning phase but was left out was On Demand Loading. On Demand Loading meens a feature that enables a retrieval of a specific after image of a database object from the SSS. This can be done in the Recovering Unit when the Index is up-to-date but the database object in question has not yet been fetched from the SSS by recovering mechanism. The location of a specific object on SSS can be get from the Index and with that information the specific database object can be fetched from the SSS.

There should be an outside thread that uses the Index and Storage Module to fetch objects from the SSS. Log manager and modules inside it has been planned so that only one thread uses them at a time. That's why mutexes should be added to those modules that are used by the outside thread.