Design document

Version 2.0

March 30, 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 Overview of the document

This document gives a detailed description about how the Log system for Rodain 2000 Real-Time Database Prototype will be implemented. High-level description of the Log system and the requirements set by the Rodain Database Prototype can be found from the Requirement specification document.

The architecture of the system in general has been described in Section 2. Section 3 outlines the data structures used by the Log system. Section 4 specifies the external interfaces of the Log system. The logic used in recovery has been explained in Section 5. Every software module to be implemented has been described in detail in Section 6. Section 7 presents how all the modules and the whole system should be tested. Section 8 defines the implementation restrictions.

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

2 Architecture plan

This section gives an overall picture of the Log system. The first subsection describes how the system relates to its environment. The second subsection shows how the system decomposes into two physically separated parts and presents the internal architecture of the system by listing the modules it consists of.

2.1 General overview of the system

The Log system is a new log-handling subsystem for the Rodain 2000 Real-Time Database Prototype. This is a distributed main-memory database. Each database node consists of two units: Primary Unit and Mirror Unit. Primary Unit serves clients and executes their transactions. It sends the generated log records to the Mirror Unit. MU then updates its own copy of the database and takes care that the log records are also stored to a stable storage on disk. If either one of the units fails, the remaining one becomes Transient Unit. TU continues serving clients. After the failed unit has recovered it becomes Mirror Unit and TU becomes Primary. The Log

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

system runs on Mirror Unit, Transient Unit and Recovering Unit. It is not needed in Primary Unit, because the only thing PU does with the log records is sending them to the Mirror Unit.

Figure 1 lists all the external objects that the Log system deals with. The Log mediator signals the Log system with a semaphore when new log records have arrived in the buffer (arrow 1). The Log system fetches the records from the buffer (arrow 2) and updates the database (arrow 3). After sending the log record to disk it signals the Log mediator (arrow 4), which can then tell Primary or in TU the DBMS of the unit to clear the corresponding log records from its buffer. Recovering Unit updates the database with the log and image records delivered from the disk.

In Mirror Unit the Log system periodically writes an image of the database to the disk. In order to do that it needs to get the data records from the database (arrow 5). Writing the database image is never started in Transient or Recovering Unit.

If the unit becomes Transient while it is still recovering, the DBMS may ask for a data record to be put into the database immediately, because some client's transaction needed it (arrow 6). This feature will not be implemented in this project, but we should design the system in a way that it is easy to add that feature later on.

2.2 Structure of the system

2.2.1 General structure

Figure 2 shows how the Log system consists of two parts: Log manager and SSS. Log manager runs on the database unit machine and SSS runs on a separate Linux machine. They are connected via IP-network and they use TCP to communicate. Later the Linux machine is planned to be replaced by a dumb disk and therefore the SSS part is designed to contain only minimal logic.

2.2.2 Module hierarchy

Figure 3 shows the internal structure of the Log manager. The Control Module knows the state of the system and calls the other modules to perform tasks. The arrows represent all the possible calls between modules. Each arrow points from the calling module to the called module. The modules are discussed in detail in chapter 6.

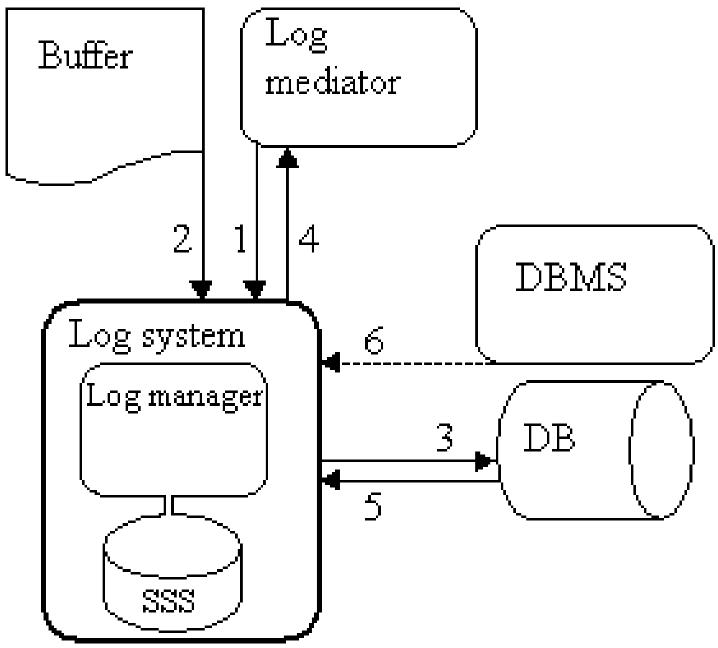

Figure 1: The environment of the Log system

3 Data plan

3.1 Global data structures

The Log system maintains both in the Log manager and in SSS information about the files on the SSS. This information includes references to active and passive log files and to index file (the files on the SSS have been described in sections 3.2.3 and 3.2.4). Information whether the last saved image was succesfully completed and whether the index has been stored after the latest image are also maintained. This information is stored to disk at the SSS, making it available after a system failure during the recovery stage. The Log manager does not explicitly ask the SSS to update this information to disk. Instead the SSS automatically updates these variables to disk when the information changes, and the Log manager updates its own copies while functioning normally.

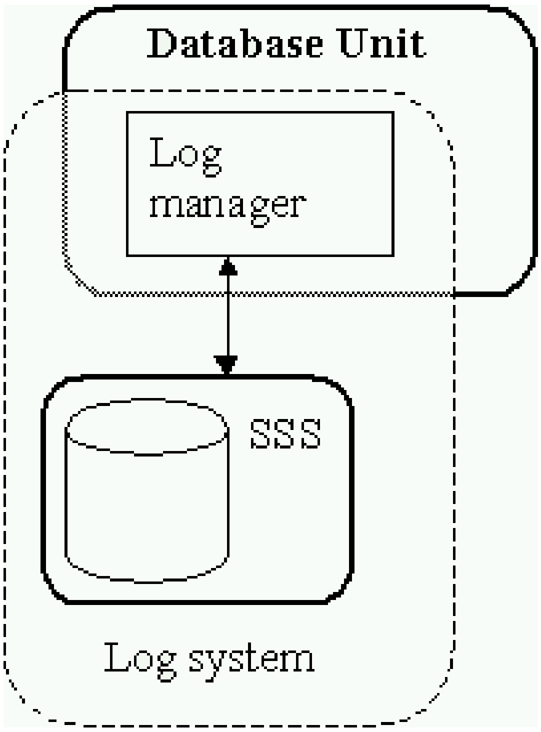

Figure 2: Structure of the Log system

3.1.1 File status information

The file status information is a structure that contains the information described in the previous section. The variables in the structure and their explanations are listed next.

index_file	contains reference to the index file
active_log_file	contains reference to the active log file
passive_log_file	contains reference to the passive log file
image_complete	boolean variable - true if the image in active log file is complete, false otherwise
index_stored	boolean variable - true if index has been stored after the latest image
size_of_active_log_file	contains the size of the active log file
size_of_passive_log_file	contains the size of the passive log file
size_of_index_file	contains the size of the index file

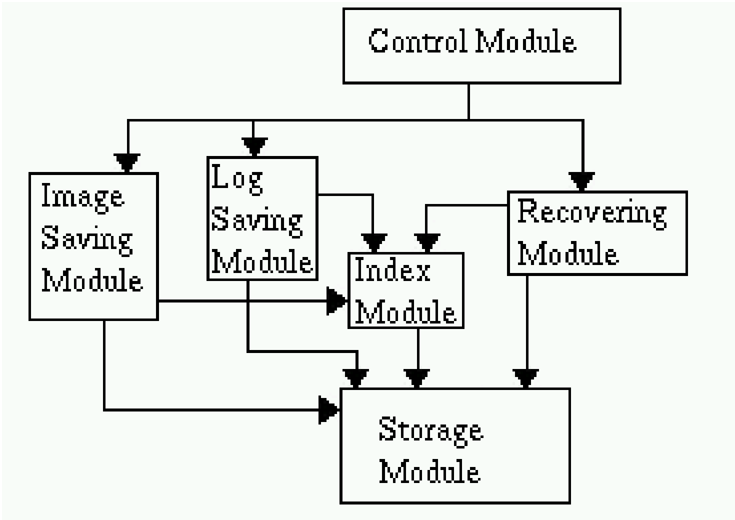

Figure 3: Internal structure of the Log manager

Log manager retrieves this information from the SSS when the Recovering Unit or the Transient Unit needs to know the file status of the SSS. Log manager needs to have this information before it can write or read anything on the SSS.

After Log manager has retrieved this information both SSS and Log manager updates it separately. SSS updates the variables *(size_of_active_log_file)*, *(size_of_passive_log_file)* and *(size_of_index_file)* only when the Log manager sends the file status read request. SSS stores the file status information into a file every time one of the other variables is changed.

When the Log manager refers to a file on the SSS, it uses the file identifiers in the file status information. This identifier is actually an integer. For example the identifier of the index file could be zero (0), active log file one (1) and passive log file two (2). The Log manager uses those values when it wants to access these files. The SSS maps all the file identifiers to real files on the disk. Now the Log manager doesn't need to know the actual names or file descriptors of the files on SSS. When the Log manger starts to write new image both Log manager and SSS switch the file identifiers of the active and passive log files, so in this example the identifier of the active file would then be two (2) and passive one (1). After this switch the passive log file refers to prior active log file and the active log file refers to prior passive log file which is then overwrited with the new image and log records.

3.2 Internal data structures

3.2.1 Log manager

The Log manager maintains an index. The purpose of the index is to make the recovery process faster and to make the Recovering Unit able to serve as a Transient Unit (when Primary Unit fails while the other unit is recovering) in an early stage. The index makes it possible to retrieve the latest image of a database record directly from the SSS, instead of searching the whole log file for the corresponding record.

The Log manager contains also a global structure *LM_info* where is collected all the global variables needed in the Log manager. These variables are:

records_for_image	counter of processed log records: This variable is incremented by the Log Saving Module after every processed log record. Control Module uses this variable to decide when to start storing image to SSS.
records_for_index	counter of processed log records: This variable is incremented by the Log Saving Module after every processed log record. Control Module uses this variable to decide when to store index to SSS.
file_status	file status information structure: The Log manager uses this information when storing data to SSS or when retrieving it.
index_present	true if index is present: This variable is used in a Transient Unit to indicate if it has an index structure. If Transient was previously Mirror it has it but if it was a Primary it don't.
first_recovery_offset	When the recovery starts the size of the log record is stored into this variable. Recovery Module uses this information.
second_recovery_offset	When the first log record is received from the Log mediator the size of the active log file is stored into this variable. Recovery Module uses this information.

3.2.2 SSS

The Secondary Storage Subsystem (SSS) is logically just a disk unit, while in reality the SSS is currently a Linux machine. The RTD-Log project will implement some logic to the SSS, still keeping it simple. Replacing the Linux machine with a real disk should have as little affect as possible to the rest of the implemented software.

The SSS receives log records from the Mirror Unit or from the Transient Unit, and takes the responsibility of saving these log records to disk. The SSS also stores the image and the index to disk when Mirror Unit so decides. To be able to keep track of which files are active, or information about the latest image or index storage, the SSS maintains the file status information, where such information is stored and therefore available after a system failure.

All in all the SSS maintains two log files, an index file, and the file containing the file status information.

3.2.3 Index file

The SSS contains one index file. When a new index is stored, the SSS creates a new file, writes the new index into this file, and if completed succesfully replaces the old index file with the new one. If an error occures during the storage of index, the new file is deleted leaving the old (complete) index file as before.

3.2.4 Active and passive log files

SSS maintains two files for log records, active log file and passive log file. New records are always written into the active log file. When Mirror Unit starts writing the image SSS switches the active and passive file so that the passive file becomes the prior active file and vice verca. After that SSS writes all the image and log records into the new active log file. After the passive log file is changed to active log file SSS overwrites the file with the new records. Log manager must also switch the references of the active and passive log file in its file status information when it starts to write new image.

The active log file has to contain one complete image and at least one index (stored after the image), before a new image can be stored and the file switch can be done. With this procedure the passive log file will allways contain a full image and a quite recent index.

The passive log file is needed because the Mirror Unit, which is writing the image into the active log file, may fail and there may be some after images of database objects that has no representation in the active log file but only in the passive log file that contains the latest and only complete image. Therefore the passive log file is needed until a complete image has been stored into the active log file.

SSS and Log manager maintains the *image_complete* and *index_stored* variables. If *image_complete* is true the active log file contains a full image. If it is false the image in the active log file is not complete. If *index_stored* is true then the active log file contains a complete image and index has been stored after that. Index is never stored during image saving. When the Log manager starts to store image these both variables are set to false both in the SSS and in the Log manager.

3.2.5 The log record

The Log system gets a log record from the buffer. The log record structure has been defined as follows in the existing system:

```
trid    transaction id (not relevant to Log system)
oper    database operation caused by this log entry
oid     id of object
toid    id of object's type (not relevant to Log system)
istype  flag for indicating if object itself is a type
        (not relevant to Log system)
len     length of object's data
data    data of object
```

The trid and oper fields have no meaning in an image record. They are still needed, because the image record structure is made identical with the log record structure so that log records and image records can be handled as if there was no difference. The data field also has a structure of its own, but the only information Log system needs from it is the timestamp of the object. It can be used in resolving which one of two records represents newer version of the object.

3.3 Communication protocol

This section defines the communication protocol used by Log manager and SSS. Log manager sends messages to SSS which can be for example a storing message of a log record or read request of the latest index structure on SSS. SSS on the other hand sends a reply to every read request it receives from the Log manager.

3.3.1 Messages sent by the Log manager

When the Log manager starts up it first asks from the SSS the file status information. After that the unit retrieves the latest index structure. When the unit starts to build up the database the Log manager sends read requests on the database objects it doesn't have yet. The whole time the unit is recovering and also after that the Log manager sends to SSS all the log records it receives. After the unit has recovered and it is running as Mirror Unit it periodically stores the image of the database and the index structure to SSS.

The message send by Log manager has a structure which is illustrated in the Figure 4. The message has a fixed size (6 bytes) header. The header contains three fields.

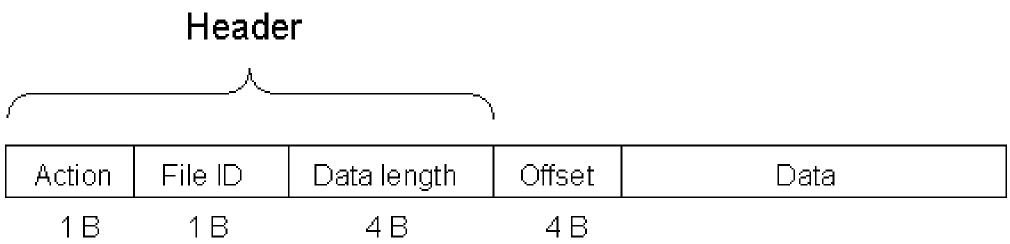

Figure 4: Structure of the message send by Log manager

The first field is an action field (1 byte) which tells to SSS the purpose of the message. The value of the field is one of the followings:

SM_FILE_STATUS	read request of the file status information
SM_LOG_WRITE	message contains a log record to be stored
SM_LOG_READ	read request of a stored log record
SM_INDEX_WRITE	message contains a index structure to be stored
SM_INDEX_READ	read request of the latest stored index structure
SM_IMAGE_START	message contains the first record of the new image
SM_IMAGE_END	message contains the last record of the new image

Second field is the file ID (1 byte). The file ID identifies the file on the SSS the Log manager wants to read or write.

Third field (data_length, 4 bytes) contains the length of the data part of the message.

If the message is a read request of a stored log record (action field is SM_LOG_READ) then the message also contains an offset field (4 bytes). The offset specifies the location of the log record in the file in question.

When the Log manager has some data for the SSS it's in the data part of the message.

3.3.2 Messages sent by the SSS

The messages sent by the SSS contains a 5 byte header which contains a 1 byte *action* field which can be one of the following:

SSS_FILE_STATUS	message contains the values of the FileManagers variables
SSS_INDEX	message contains an index
SSS_LOG	message contains a log record

The following 4 bytes in the header contains the *length* of the data that follows in the variable length *data* part.

The structure of the message sent by the SSS is shown in Figure 5.

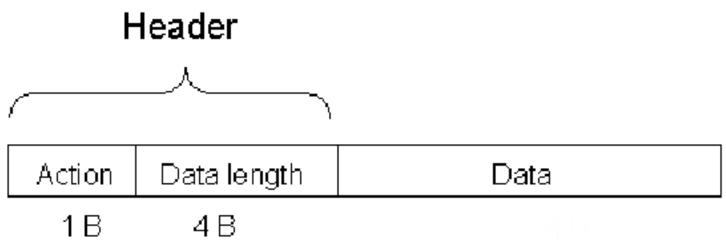

Figure 5: Structure of the message sent by the SSS

3.4 Adjustable parameters

This subsection summarizes the defined values that can easily be changed. They are located in the header files of the modules.

Control Module:

LOG_WRITE_COUNT	The maximum amount of log records written during one call to Log Saving Module.
IMAGE_WRITE_COUNT	The maximum amount of image records written during one call to Image Saving Module.

IMAGE_SAVE_LIMIT	The number of log records processed between two image saves.
INDEX_SAVE_LIMIT	The number of log records processed between two index saves.
Index Module:	
INDEX_SIZE	The maximum size for index.
REALLOC_SIZE	The amount of space that's allocated at once when reallocating space for overflow-list.

4 Interface plan

4.1 Specification of external interfaces

The external interfaces the Log System uses are described here. There are three of them:

1. The state variable interface between the Log manager and the Log mediator
2. The buffer interface between the Log manager and the Log mediator
3. The interface between the Log manager and the database

4.1.1 The state variable interface

The Log manager and the Log mediator share a variable that indicates the state of the unit. Both of them read and update the variable through functions STA_get_state and STA_set_state. The variable is set before starting up the Log system so that the Log system knows if it should start recovery (in RU) or just handle the log flow (in TU). The Log system updates the variable as the recovering proceeds. Then the Log mediator can decide when to turn the log flow and change the unit state to Mirror or Transient.

The Log Mediator sets the values MIRROR, RECOVERY and TRANSIENT. The Log system sets the values INDEX_LOADED, INDEX_READY and DB_READY according to the level of recovery the unit has reached. The PRIMARY value is actually not needed because the Log system does not run on Primary Unit. The value is still included for perfection. The DB_READY or INDEX_READY values are set only if the Log Mediator hasn't already changed the state to TRANSIENT or MIRROR. This function makes that check and returns the value of the variable after the set operation.

The function interfaces are defined as follows:

```
STA_get_state(
)

Parameters:
    -

Return values:

    PRIMARY       in Primary Unit
    MIRROR        in Mirror Unit
    TRANSIENT     in Transient Unit
    RECOVERY      in Recovering Unit; recovery begins
    INDEX_LOADED  in Recovering Unit; uncomplete index
                  loaded from the SSS
    INDEX_READY   in Recovering Unit; index is complete
    DB_READY      in Recovering Unit; database is complete


STA_set_state(
    state
)

Parameters:
    state   IN    the new state of the system; possible values
                  listed in the return values list

Return values:
    PRIMARY
    MIRROR
    TRANSIENT
    RECOVERY
    INDEX_LOADED
```

```
INDEX_READY
DB_READY
```

4.1.2 The buffer interface

The interface between the buffer and the Log manager is implemented in functions BUF_read and BUF_clear. The buffer is guarded by a semaphore whose value these functions read and decrease. The Log mediator increases the value as it adds log records to the buffer.

BUF_read function reads a log record from the buffer. It takes a parameter which tells if the semaphore check should be blocking or not. The function first checks the semaphore to see if there is anything to read in the buffer. If the buffer is empty and the check is blocking, the function waits until a record arrives to the buffer. If the buffer is empty and the check is non-blocking, the function returns BUF_EMPTY. If the buffer is not empty, the function returns a log record.

The semaphore check should be blocking in normal log handling in both MU and TU. This is because the Log system has nothing else to do and would only poll the buffer in a loop. It is better to make it wait. But sometimes in MU the Log system may be saving the image in turns with handling the log records. Then it is better that the semaphore check is non-blocking so that the image saving can be continued if the buffer is empty. The same applies to the situation in RU where the Log system loads missing objects into the database while it is not handling the log flow.

The function interface is defined as follows:

```
BUF_read(
    len
    target
    blocks
)

Parameters:
    blocks  IN    should the buffer semaphore check be
                  blocking or not
    len     IN    length of the area reserved for the log record
    target  OUT   pointer to the area the log record is copied into
```

```
Return values:

    SUCCESS      function succeeded
    BUF_EMPTY    buffer is empty, no record is returned
    OVERFLOW     the log record does not fit into the
                 area reserved for it; no record is returned
```

BUF_clear function is called after the log record is sent to the SSS. It decreases the buffer semaphore value and tells the Log mediator that the log record with the specified oid and trid is handled.

The function interface is defined as follows:

```
BUF_clear(
    oid
    trid
)

Parameters:

    oid     IN     oid of the object
    trid    IN     transaction id

Return values:

    SUCCESS            function succeeded
    INVALID_PARAMETER  some of the parameters was invalid
```

4.1.3 The database interface

The interface between Mirror Unit's database and Log manager is implemented in functions DB_image_read, DB_image_write and DB_log_write.

DB_image_read function is used for reading image records from the database when the Log system wants to store the database image to the SSS. The function gets an oid as a parameter and returns the object holding the specified oid. If the oid is not valid, some of the error codes is returned.

The function interface is defined as follows:

```
DB_image_read(
   trid
```

```
    oper
    toid
    istype
    oid
    len
    target
)
```

Parameters:

trid	OUT	any value; needed only for making image record structure identical with log record structure
oper	OUT	must be "insert" operation; image records only add an object to the database, never delete or update anything
toid	OUT	id of object's type
istype	OUT	flag for indicating if object itself is a type
oid	IN	oid of the object to be read
len	IN	length of the area reserved for the object's data
target	OUT	pointer to the area the object's data is copied into

Return values:

SUCCESS	function succeeded
INVALID_OID	the oid is invalid
OID_NOEXISTS	the database does not contain an object with this oid
OVERFLOW	the object's data does not fit into the reserved area; no record is returned

DB_image_write function is used by the Recovering Unit when the Log system loads image and log records from the SSS and puts the objects to the database. One image or log record is handled at one call. If the database already contains an object with the given oid, an error code is returned and the existing object is left untouched. This way we avoid replacing an object with its older version.

When recovering, the image and log records are read from the disk newest first. After the log flow has been turned, also the Log Saving Module puts missing objects into the database and updates existing ones. So if the database already contains a particular object, it is always newer version than the one Recovery Module has fetched from the disk. The Recovering Module first checks from the index if the database already contains the object and does not try to add existing objects to it. Hence the DB_image_write function should never need to return the OID_EXISTS value. It is included anyway to be on the safe side.

The function interface is defined as follows:

```
DB_image_write(
   trid
   oper
   oid
   toid
   istype
   len
   source
)

Parameters:

   trid    IN   transaction id
   oper    IN   database operation caused by this log entry
                (always "insert")
   oid     IN   id of object
   toid    IN   id of object's type
   istype  IN   flag for indicating if object itself is a type
   len     IN   length of object's data
   source  IN   data of object


Return values:

   SUCCESS            function succeeded
   INVALID_OID        the oid is invalid
   INVALID_PARAMETER  some of the other parameters was invalid
   OID_EXISTS         the database already contains an object with
                      this oid
```

The Log Saving Module uses a different database write function DB_log_write. It is used in normal log handling when the Log system updates the database with the log records. This means deleting, inserting or updating objects. One log record is handled at one call.

The function interface is defined as follows:

```
DB_log_write(
   trid
   oper
   oid
   toid
   istype
   len
   source
)

Parameters:

   trid    IN   transaction id
   oper    IN   database operation caused by this log entry
   oid     IN   id of modified object
   toid    IN   id of object's type
   istype  IN   flag for indicating if object itself is a type
   len     IN   length of object's data
   source  IN   data of object

Return values:

   SUCCESS            function succeeded
   INVALID_OID        the oid is invalid
   INVALID_PARAMETER  some of the other parameters was invalid
   OID_EXISTS         the database already contains an object with
                      this oid (insert operation)
   OID_NOEXISTS       the database does not contain an object
                      with this oid (delete or update operations)
```

5 Recovery logic

When a unit starts up, the Fault-Tolerance and Recovery Subsystem (FTRS) creates a thread where the Log manager will be executed. Log manager contains function *start_log_manager* that is called in the thread creation. (FTRS refers to same module than the Log mediator. FTRS is the name for that module in the RODAIN project and the name Log mediator is commonly used in this project (RTD-Log). But basically they mean the same thing.)

Log manager and FTRS shares a global variable where is the information about the state of the unit. The Log manager can access this variable using functions *STA_set_state* and *STA_get_state* defined on page 13. When the Log manager is started the state can be either RECOVERY or TRANSIENT. If it is RECOVERY it means that the unit was just started. If the state is TRANSIENT then the unit had been Primary Unit and because a failure in the Mirror Unit it has become a Transient Unit. In the next section has been described what the Log manager does when it is started in the Recovery State and in the section after that what it does when it is started in the Transient State.

5.1 Log manager in the Recovery State

First Log manager retrieves the file status information from the SSS because it knows nothing about the files on the SSS. After that it retrieves the latest index stored on SSS.

When the index has been received Log manager starts to read from the active log file those log records that has been saved after the latest index. This has to be done because the Log manager needs to have an index that contains information about the latest disk representations of every database object on the SSS. There is only one problem. There migth be a Transient Unit in the same node that is storing new log records to SSS at the same time. That is why Recovering Unit needs to inform the Transient Unit that it has to stop sending the records directly to SSS and start sending them through the Recovering Unit.

When the Recovering Unit has fetched the latest index from the SSS it sets the global state variable to indicate that index has been received and it can start to process the log records from the Transient Unit. The FTRS notices the new value (INDEX_LOADED) in the state variable and informs the Transient Unit that it can start to send the log records to the Recovering Unit. The FTRS in Transient Unit sends a signal to the Log manager to indicate that the Log manager should stop processing the log records, clean all internal structures and kill its thread of execution. After that the Transient Unit switches its state to Primary Unit and starts sending the log records to Recovering Unit.

During the time the Transient Unit changes its state to Primary and turns the log flow the Recovering Unit reads the log records from the SSS. It does that until it reads a record that indicates that the latest index structure was saved at this point. If the active log file doesn't contain this kind of record but the *image_complete* variable in the file status information indicates that the active log file contains full image then the Log manager needs to read the whole file. If the active log file doesn't contain full image then the Log manager needs to read also the passive log file until it encounters a log record that indicates when the latest index was stored.

The reading of the log records is suspended when the Log manager in Recovering Unit receives the first log record from the Primary Unit. At this point the Log manager can be sure that nobody else is storing any log records to SSS. Now the Recovering Unit retrieves the file status information again from the SSS so that it gets the information how many log records the Transient Unit stored to SSS after the Log manager in Recovering Unit retrieved the file status information in first place.

After that the Log manager knows all the log records it has to read to get its index up-to-date. At this point it also can start to process the log records received from the Primary Unit. Because all the log records contains a time stamp the Log manager is able to check that if a newer after image of a database object has been received from the Primary Unit the Log manager doesn't overwrite it with an older one read from the SSS during recovery.

Recovering Unit's index is up-to-date when the Log manager has read all the log records from the SSS that were stored after the latest index or when a complete image has been read. After that the Log manager changes the shared state variable to indicate the FTRS that index is up-to-date (INDEX_READY). Now there may be some database objetcs in the SSS that has no representation in the Recovering Unit's database. The Log manager still needs to retrieve those records until it is fully recovered. After this is done the Log manager changes the state variable to indicate that the database is ready and the recovery is done (DB_READY). Now the FTRS can change the unit's role to Transient or Mirror. If there is a Primary Unit in the same node then the Recovering Unit switches its state to Mirror Unit, otherwise it will become a Transient Unit. The FTRS is able to change the unit's state to Transient or Mirror already when the Log manager has indicated that the index is ready (INDEX_READY). If the FTRS has done this the Log manager doesn't set the shared status variable to DB_READY when the recovery is done.

5.2 Log manager in the Transient State

When the Log manager is started the unit's state can also be Transient. This happens when Mirror Unit fails in normally operational node. When the Primary Unit notices this it switches its state to Transient and starts the Log manager. First in the Transient Unit the Log manager retrieves the file status information from the SSS. After that it is ready to process the log records. In this situation the Log manager doesn't maintain index structure nor store images to SSS. If Mirror Unit becomes a Transient Unit it is optional (compiler option) whether the Unit maintains the index or not. If Mirror was storing image when Primary failed it is optional whether it finishes it in Transient Unit.

6 Module plan

This section describes in detail every module in the Log system. The interface of every module has been defined as well as the internal data structures. The implementation in pseudocode level has been outlined and the error handling has been described.

6.1 Control Module

Control Module is the heart of the Log manager. When the Log manager starts up, the Control Module starts running. The Control Module then calls the other modules in the Log manager which do the actual work.

Log manager shares a global variable with the FTRS where the state of the unit is stored. Unit can be either Recovering, Mirror, Transient or Primary Unit. Log manager is not executed in the Primary Unit because Primary Unit sends log records directly to Mirror Unit so there is nothing to do for the Log manager.

When the Control Module first starts running, it initializes all the global structures and all the other necessary modules by calling their initialize functions. After that it starts running in an infinite loop. The state of the unit is checked within every loop, and depending on the current state it calls other appropriate modules.

6.1.1 Interface

Control Module doesn't have a real interface. The whole module consist of one function, *start_log_manager*, which is called by FTRS when it creates the thread where the Log manager is executed.

6.1.2 Referenced modules

In Recovering Unit the Control Module calls alternately Recovery Module and Log Saving Module. When Recovery Module is called it does a little bit recovering work and returns. After that the Control Module calls Log Saving Module which process at most certain number of log records and then returns (Control Module gives this upper limit as a parameter when it calls the Log Saving Module). Then the Control Module calls the Recovering Module again. When the unit has entirely recovered Recoverig Module returns value DB_OK so the Control Module knows that it doesn't need to call it anymore.

In the Mirror Unit the Control Module uses most of the time calling just the Log Saving Module. From time to time Log manager needs to store Mirror Unit's copy of the database (image) to SSS. This is done after some defined number of log records has been processed and stored to SSS. Log manager contains a shared variable *records_for_image* which is increased by Log Saving Module after every log record it process. When this variable gets to a certain value (IMAGE_SAVE_LIMIT) Control Module sets it to zero and calls then Image Saving Module which starts to storing image to SSS. Image Saving Module stores only few image records to SSS at a time so that the Log Saving Module can also be executed periodically (this image record limit is also given as a parameter). Image Saving Module returns value IM_IMAGE_COMPLETED when the whole image has been saved to SSS.

From time to time the Log manager also needs to store the index structure to SSS. This is done when *records_for_index* variable gets to a certain value (INDEX_SAVE_LIMIT) and also after image has been stored to SSS. *records_for_index* variable is also incremeted by the Log Saving Module and set to zero by the Control Module. Control Module calls the Index Module to store the index structure. Index is not stored when image saving is unfinished.

When the Log manager is started and it has received the file status information from the SSS, the Control Module calculates the initial values for *records_for_image* and *records_for_index* using the current size of the active log file and the fixed size of one log record.

In the Transient Unit the Control Module acts like in the Mirror Unit but it doesn't store index to SSS nor start storing image. But if the unit was previously Mirror Unit and it was storing image but wasn't able to finish it before the unit became Transient, then it is optional whether the Log manager continues the image storing in the Transient Unit. This functionality can be enabled with a compiler option.

6.1.3 Internal data structures

Control Module doesn't have any internal data structures. It only maintains few internal state variables.

6.1.4 Description

Here is the implementation of the Control Module in pseudocode level. The functionality of the Control Module is based on the global state variable. The description and possible values of this variable can be found from section 4.1.1 on page 12. More information about the usage of this variable is given in the section 5 on page 18.

```
start_log_manager() {

    initialize global structures;
    initialize other modules;

    retrieve file status information;
    IF global_state == RECOVERY THEN
        call Index to retrieve latest index structure;
        set global state to INDEX_LOADED;

    while(FOREVER) {
        switch(global_state) {

        case INDEX_LOADED:
            call Recovery Module;
            call Log Saving Module(non-block);

            WHEN index is ready THEN
```

```
        set global variable to INDEX_READY;

case INDEX_READY:
    call Recovery Module;
    call Log Saving Module(non-block);

    WHEN recovery is done THEN
        set global state to DB_READY;

case DB_READY:
    call Log Saving Module(block);
    break;

case MIRROR:
    IF recovery is not ready THEN
        call Recovery Module;
        /* This can happen when FTRS changes
           the state to MIRROR_UNIT after
           Control Module has changed the
           state to INDEX_READY. */

    IF it's time to store index THEN
        call Index Module to store index;

    IF it's time to store image OR
       image saving is unfinished THEN
        call Image Saving Module to store image;

    IF recovering OR storing image THEN
        call Log Saving Module(non-block);
    ELSE
        call Log Saving Module(block);

    break;

case TRANSIENT;
    IF recovery is not finished THEN
        call Recovery Module;
        /* This can happen when FTRS changes
           the state to TRANSIENT_UNIT after
           Control Module has changed the
```

```
                  state to INDEX_READY. In this case
                  the unit is the only operational unit
                  in the node so it has recovered to
                  Transient Unit */

           IF image saving has been started but not finished THEN
               call Image Saving Module to store image;
               /* If this unit is a prior Mirror Unit and it has
                  started image saving, it can be finished in the
                  Transient Unit */

           IF recovering OR storing image THEN
               call Log Saving Module(non-block);
           ELSE
               call Log Saving Module(block);

           break;
       }
   }
}
```

6.1.5 Error handling

If Control Module receives from other module an error code that indicates a fatal error (ERROR) or if some kind of fatal error occurs inside the Control Module, like failure in memory allocation, then the Control Module just kills the Log manager's thread of execution.

Image and Log Saving Modules return error code INVALID_PARAMETER if the image or log record limit given as parameter is invalid. This should never happen in this implementation because those values will be hard coded using definitions but if for some reason Control Module receives this kind of error code it will be treated as fatal error.

Storage module and some modules which use Storage Module may return error code SM_TCP_ERROR. This error is also treated as fatal.

6.1.6 Alternative design solutions

No other design solutions for Control Module was considered.

6.1.7 Size estimate

The whole size of the Control Module will be approximately 200 lines of C-code.

6.2 Index Module

The purpose of the Index is to speed up the recovery process by making it possible to access the latest value of any given data object directly from the SSS. The Index module is also responsible of storing the whole Index to the SSS and reading it back from the SSS when needed.

6.2.1 Interface

There are several functions that make up the actual interface to the Index module. These functions are briefly outlined here and more detailed decriptions are given in the next section.

IM_read_index function is for reading information regarding a particular data object. The function returns (oid), fileid, offset and length of the data.

```
int IM_read_index(
    oid,
    fileid,
    offset,
    length
)

Parameters:
    oid      IN    object identifier
    fileid   OUT   identifier of the target file on SSS
    offset   OUT   the position of the record in the file
    length   OUT   length of the data

Return value:
    SUCCESS      if function succeeded
    IM_NOTFOUND  if no entry was found
    ERROR        if function failed
```

IM_read_nextindex_ISM is for reading index entries in order. This function is used only by Image Saving Module. There is similar function for Recovery Module, called IM_read_nextindex_RM. Although these two fuctions are closely related, their operational logic is a bit different. For increased clarity and modularity, it's better to use two different functions.

When IM_read_nextindex_ISM is called for the first time, an internal pointer is initialized to mark which entries are already returned. Every subsequent call to this function will return the next index entry that hasn't already been returned or updated since the image saving started (no need to return updated entries, because they are already written to the SSS through the Log Saving Module). A file identifier is used to decide which entries are to be returned (this also makes it possible to resume incomplete image saving). When there are no more entries to return the internal pointer is cleared and a special return value is used to signal that the image saving is completed. Next call to this function will be regarded as a first call (i.e. it starts a new image saving procedure).

```
int IM_read_nextindex_ISM(
    oid
)

Parameters:
    oid      OUT   object identifier

Return value:
    SUCCESS    if function succeeded
    ERROR      if function failed
    IM_END     if no more entries to return
```

IM_read_nextindex_RM does basically the same thing that IM_read_nextindex_ISM function does; it returns all the index entries in order (excluding those that are already updated through the Log Saving Module).

When this function is called for the first time, a special "dirty"-flag is raised for all entries and internal pointer is initialized to mark which entries are already returned. Every subsequent call to this function returns the next "dirty" entry. "Dirty"-flag is cleared when the entry is returned or updated through the Log Saving Module (IM_write_index used by LSM clears the "dirty"-flags). When there are no more entries to return the internal pointer is cleared and a special return value is used to signal that the recovery is completed.

```
int IM_read_nextindex_RM(
    oid,
    fileid,
    offset,
    length
)

Parameters:
    oid      OUT   object identifier
    fileid   OUT   identifier of the target file on SSS
    offset   OUT   the position of the record in the file
    length   OUT   length of the data

Return value:
    SUCCESS    if function succeeded
    ERROR      if function failed
    IM_END     if no more enries to return
```

IM_check_index is needed when the Recovery Module has received log entries from the SM, and is about to update received data to the database. Before the database is updated, it's necessary to call this fuction to check if the oid at hand has already been updated through the LSM, while RM was waiting for the data. If already updated, there's no need to update the database.

```
int IM_check_index(
    oid,
    timestamp
)

Parameters:
    oid        IN   object identifier
    timestamp  IN   timestamp to compare

Return values:
    SUCCESS    if update is needed
    ERROR      if no update is needed
```

IM_write_index is for writing a new entry to the Index. If the there already is an entry for the same oid, its information is updated.

```
int IM_write_index(
    oid,
    fileid,
    offset,
    length,
    timestamp
)

Parameters:
    oid        IN   object identifier
    fileid     IN   identifier of the target file on SSS
    offset     IN   the position of the record in the file
    length     IN   length of the data
    timestamp  IN   timestamp of the object

Return value:
    SUCCESS            if function succeeded
    INVALID_PARAMETER  if there was invalid parameter
    ERROR              fatal error
```

IM_delete_index is for removing an index entry regarding a particular data object.

```
int IM_delete_index(
    oid
)

Parameters:
    oid       IN    object identifier

Return value:
    SUCCESS            if funcion succeeded
    INVALID_PARAMETER  if there was invalid parameter
    ERROR              if function failed
```

IM_get_index is for reading the latest full index back from the SSS. After the index is read a special "dirty"-flag is raised on all entries. This flag indicates that the entry is from the latest index-file, and its value is not necessarily up-to-date.

```
int IM_get_index(
    void
)

Return value:
    SUCCESS   if function succeeded
    ERROR     if function failed
```

IM_put_index is for writing the whole index to SSS.

```
int IM_put_index(
    void
)

Return value:
    SUCCESS   if function succeeded
    ERROR     if function failed
```

6.2.2 Referenced modules

Index Module is used by Control Module, Log Saving Module, Image Saving Module and Recovery Module. Index module itself uses only the Storage Module.

6.2.3 Internal data structures

The essential data structure of the Index is a hash table. Hash table is convenient choice for a data structure, because it allows implementing find (read) and insert (write) operations that work in constant average time.

Another advantage is fast and relatively easy way to store the whole Index to SSS as well as read it back. The hash table is implemented as an array (of structs) which is simply a contiguous block of memory and therefore the whole Index can be written down to SSS (or read back) at once, instead of manipulating it one Index-entry at a time.

Downside of using hash table and array structure is that we need to know the size of the data structure in advance.

The data structure for singular Index entry is a struct. It takes the following form:

```
struct IndexEntry{
    int oid;         /* Object identifier */
    int offset;      /* Offset in a log-file (on SSS) */
    char fileid;     /* Log-file identifier */
    int length;      /* Length of the data object */
    int timestamp;   /* Timestamp of the object */
    char flags;      /* For some state-information */
    int nextindexid; /* Next Index entry */
};
```

The overall size of a singular Index entry should be a nice computational value (16B or 32B sounds good).

The Index itself is an internal data structure to Index Module, and it can be directly manipulated only by the funtions of the Index Module. In addition to this data structures, there are some internal variables that are accessible to all the functions of the Index Module (but they are not accessible outside the Index Module).

6.2.4 Description

All the functions of the Index Module are described here in pseudocode. As we already saw in previous sections, Index Module has several interface functions that are used by other modules. There are also internal functions that deal with allocating the memory for required data structures. Their pseudocode presentation is given first.

IM_init_index is used when the Index is initialized for the first time. It must be called before the IM_get_index function.

```
IM_init_index(){
```

```
    first=malloc(INDEX_FILE_SIZE);
    IF(first==NULL)
        return ERROR;
    ELSE return SUCCESS;
}
```

IM_kill_index is needed when the system is shut down in controlled fashion. It frees the memory reserved by the Index.

```
IM_kill_index(){
    free(first);
}
```

reallocate_index is needed when we need more space for the Index (actually for the overflow-list, for the size of the Index itself is predefined).

```
reallocate_index(){
    newspace=realloc(first,(CURRENT_SIZE+REALLOC_SIZE)*sizeof(IndexEntry));
    IF(newspace==NULL)
        return ERROR;
    ELSE return SUCCESS;
}
```

Here are the pseudocode presentations of the interface functions described in previous section.

```
IM_read_index(*id, *fid, *off, *len){
    compute hash;
    WHILE(entry not found AND entry has next)
        move to next entry;
    IF(entry found){
        set values;
        return SUCCESS;
    }
    ELSE return IM_NOTFOUND;
}
```

```
IM_read_nextindex_ISM(*id, *fid, *off, *len){
    IF(first call){
        get active_log_file;
        move pointer to first oid;
        set values;
        result=SUCCESS;
    }
    ELSE{
        move pointer to next oid;
        IF(no more entries)
            result=IM_END;
        ELSE{
            set values;
            result=SUCCESS;
        }
    }
    return result;
}


IM_read_nextindex_RM(*id, *fid, *off, *len){
    IF(first call){
        move pointer to first dirty;
        set values;
        clear dirty_flag;
        result=SUCCESS;
    }
    ELSE{
        move pointer to next dirty;
        IF(no more entries)
            result=IM_END;
        ELSE{
            set values;
            clear dirty_flag;
            result=SUCCESS;
        }
    }
    return result;
}
```

```
IM_check_index(oid, timestamp){
    compare timestamps;
    IF(timestamp is older)
       return ERROR;
    ELSE return SUCCESS;
}

IM_write_index(id, fid, off, len){
    compute hash;
    IF(collision){
        scan the overflow-list;
        IF(empty){
           create a new Entry;
           IF(no space for new Entry){
               reallocate_index();
               IF(reallocation failed)
                   result=ERROR;
           }
        }
    }
    IF(result!=ERROR){
       set values;
       clear dirty_flag;
       result=SUCCESS;
    }
    return result;
}


IM_get_index(){
    result=SM_index_read(fileid, *data, size);
    IF(result!=SUCCESS)
        return ERROR;
    ELSE{
         WHILE(inside_index){
            raise dirty_flag;
            move to next entry;
         }
    return SUCCESS;
    }
```

```
}


IM_put_index(){
    result=SM_index_write(fileid, *data, size);
    IF(result!=SUCCESS)
        return ERROR;
    ELSE return SUCCESS;
}
```

6.2.5 Error handling

There are basically three types of errors that can be encountered inside the Index Module:

- Internal error due to an invalid parameter: This returns INVALID_PARAMETER response to indicate that the requested task could not be fulfilled, but there was no harm done.
- Internal memory allocation error: This returns an ERROR response. Memory allocation errors are always fatal. (Note: ERROR response is also used in IM_check_index() to indicate that there is no need to update the object at hand to the DB. This isn't really an error).
- External error in SM: Storage Module is the only module that Index Module calls. If SM returns an error code, the Index Module will return ERROR.

The Index Module itself only returns error-codes, and does not try to solve the problem. It is up to the calling routine to decide how to handle the situation.

6.2.6 Alternative design solutions

It might be a good idea to create two different arrays: one for the actual hash-table, and another for the overflow-list (which is used when the original hash produces a collision). This way it would be easier to reallocate the space for the overflow-list. On the negative side, with two different arrays we need two writes to save the index to SSS, and two reads to get it back (but the whole Index would still be saved in one file).

6.2.7 Size estimate

In addition to the Index Module itself there will be quite a lot of code written for testing purposes. Total amount is estimated to be 700-800 lines of code (the Index Module itself is probably around 500 lines of code).

6.3 Log Saving Module

The Control Module calls this module to perform the main task of the system: to handle the log flow. In Mirror Unit and Recovering Unit the log flow is coming from the Primary Unit and in Transient Unit the DBMS of the same unit is generating the log records. In both cases the Log mediator puts the log records to a buffer. This module fetches the log records from the buffer and handles them according to the state of the unit.

In Mirror Unit and Recovering Unit the log records are used to update the main memory database of the unit. After that they are sent to the Storage Module which forwards them to the SSS. Then this module calls the Index Module to update the index. Finally the Log Mediator is signalled so that it can send a clear-message to the Primary Unit. The PU can then clear the handled log records from its own buffers.

In Transient Unit the log records are simply sent to the SSS and after that the Log mediator is signalled. The index is updated only if the unit has one and it is requested that the TU should maintain the index. The TU will eventually become PU which does not need the index. Hence updating the index in TU is made a compile option so that it can be turned on or off as needed for research purposes. If the unit was PU before it became TU there is no index present, so it will not be updated either.

6.3.1 Interface

The only interface to this module is the LSM_handle_logs function. The Control Module calls this function regularly. The interface function is declared as follows:

```
LSM_handle_logs(
    log_write_count
    blocks
)
```

```
Parameters:
     log_write_count   IN  the number of log records handled
                           during one module call
     blocks            IN  should the semaphore check be
                           blocking or not

Return values:
     SUCCESS           function succeeded
     ERROR             fatal error
     SM_TCP_ERROR      connection to the SSS has failed
     INVALID_PARAMETER the log_write_count parameter was
                       invalid (<=0)
```

6.3.2 Referenced modules

This module reads the log record buffer through buffer interface functions defined on page 14. The function BUF_read is used for reading the records from the buffer. After the log records have been handled, the Log mediator is signalled with the BUF_clear function.

In MU and RU the log records are used to update the database. The database interface function DB_log_write defined on page 18 are used for that purpose. The state of the unit is found out with function STA_get_state defined on page 13.

In any state the log records are given to the Storage Module that sends them to the SSS. The function SM_log_write defined on page 52 is used for that. If the unit maintains an index, the offset and file-id of the concerned data records are updated by calling the Index module. This is done by using the IM_write_index and IM_delete_index functions defined on page 29.

When the unit has started recovering and reaches the INDEX_LOADED state, the Log manager continues recovering but also starts polling the buffer for incoming log records. When the first log record has arrived, the Log manager knows that all the log records from the other unit have reached the SSS. From then on this unit knows it is the only one to write anything to the SSS. So this module fetches the file status information after it has read the first log record from the buffer. This is done by calling the SM_file_status_retrieve function defined on page 52.

6.3.3 Internal data structures

The module has no data structures of its own except some local variables. A counter variable is used for counting the number of log records written during this call of the module. Other variables are used for checking the return values of the called modules.

6.3.4 Description

The functionality of this module is presented here as pseudocode. Error handling is omitted for clarity. It is discussed in the next subsection.

```
counter = 0;

WHILE ( counter < log_write_count AND
        read record from buffer; return value was not BUF_EMPTY ) {

   IF ( unit is RU in INDEX_LOADED state AND
        counter = 0 ) {
      retrieve the file status information;
      update second_recovery_offset;
   }

   IF ( unit is MU or RU ) THEN
      update database;

   send record to Storage Module;
   update global offset of the active file;

   IF ( unit maintains index ) THEN
     update index;

   signal Log Mediator;
   counter = counter + 1;
   records_for_image = records_for_image + 1;
   records_for_index = records_for_index + 1;
}

return SUCCESS;
```

The Control Module gives the log_write_count as a parameter when it calls this module. It determines the maximum number of log records this module is allowed to handle at one call. The blocks parameter determines whether this module should wait if there are no records in the buffer. More about this is explained in Section 4.1.2 on page 14.

The second_recovery_offset is a global variable defined on page 7. The presence of an index is also detected from a global variable. The variables records_for_image and records_for_index are shared by this module and the Control Module. The Control Module uses them to determine the time when image or index must be stored to the SSS.

6.3.5 Error handling

Possible error situations and how they are handled:

- The given parameter log_write_count may be invalid (<= 0). In that case the module returns an error code INVALID_PARAMETER.

- The record in the buffer may be invalid. This module does not detect it, but the database update function may notice the invalid oid. In that case it returns the error code INVALID_OID. This module then prints an error message, discards the record and reads the next one from the buffer. It is the responsibility of the Log mediator to ensure that the records in the buffer are valid.

- The database update function may return an error code INVALID_OID if we try to add a data record with an OID that exists in the database. This module prints an error message, discards the log record and reads the next one from the buffer.

- The database update function may return an error code OID_NOEXISTS if we try to delete a data record which does not exist in the database. Normally this module prints an error message, discards the log record and reads the next one from the buffer. But if the unit is recovering, this is not considered an error because the data record has not yet reached the database. This module tries to delete the record from the index and continues log handling normally.

- The database update function may return an error code OID_NOEXISTS if we try to update a data record which does not exist in the database. Normally this module prints an error message, discards the log record and reads the next one from the buffer. But if the unit is recovering, this is not considered an error because the data record has not yet reached the database. This module then adds the record to the database and continues normally.

- The Storage Module may return an error code ERROR if a fatal error occured or SM_TCP_ERROR if the connection to the SSS failed. This module exits and returns the received error code to the Control Module.
- The Index module may return an error code ERROR if a fatal error occured in the update operation. This module exits and returns the received error code to the Control Module.

6.3.6 Alternative design solutions

No other design solutions were considered.

6.3.7 Size estimate

The LOC value for this module is about 105 lines.

6.4 Image Saving Module

The Control Module uses this module when it wants to save the database image to the disk. This is a normal procedure in the Mirror Unit. The Recovering Unit never saves the image. The Transient Unit never starts saving the image, but the Mirror Unit may become Transient in the middle of saving the image. In that case the image saving may or may not be finished. This is a decision of the Control Module as a compile option in it.

The image is saved periodically so that the permanently stored database copy can catch up with the main memory copy. The module does not try to save the whole image at one time. Instead it writes a defined number of records every time it is called. This is because the Control Module also needs to handle the normal log flow that is filling the buffer. So the Control Module calls this module several times to get the whole image written and handles the normal log flow between these calls. The last time this module is called it may write less records than the defined number, because there may not be enough records left.

6.4.1 Interface

The only interface to this module is the ISM_save_image function. When the Control Module wants to store the database image, it calls this function repeatedly until it notices from the return value that the image save has been completed. The interface function is declared as follows:

```
ISM_save_image(
    image_write_count
)

Parameters:
     image_write_count  IN  the number of image records written
                            during one module call

Return values:
     ISM_IMAGE_NOT_COMPLETED the image save is not yet completed
     ISM_IMAGE_COMPLETED     the image save is completed
     ERROR                   fatal error
     SM_TCP_ERROR            connection to the SSS has failed
     INVALID_PARAMETER       the image_write_count parameter
                             was invalid (<=0)
```

6.4.2 Referenced modules

This module calls the Index Module for two purposes. In order to fetch a data record from the database, this module needs to know which data record is the next one that has not been written yet. The IM_read_nextindex_ISM function of the Index Module returns the next unwritten record's OID in the index. The function is defined on page 27. A record is unwritten if the following two conditions are fulfilled:

1. The data record has not yet been written as part of this image.
2. The data record has not been written by the Log Saving Module since the image saving started.

When the image saving starts, the passive log file becomes the new active file and the previously active file becomes passive. The image records and log records are then both written to the active file in turns until the image saving becomes complete. That means that at least one copy of every data record in the database has been written to the active file. From then on only log records are written to the file. The Index Module can check from the index record's file-id whether the corresponding data record is unwritten. All those data records are unwritten whose file-id is the passive file's file-id.

The second time this module calls the Index module is after the data record has been sent to the Storage Module. The offset and file-id of the data record's index record must be updated then. This is done with the IM_write_index function defined on page 29.

The data records are fetched from the database using the DB_image_read database interface function. It is defined on page 15.

This module calls the Storage Module to write the image records to SSS. This is done by using the SM_image_write function that is defined on page 54.

6.4.3 Internal data structures

The module has no data structures of its own except some local variables. A counter variable is used for counting the number of image records written during this call of the module. Other variables are used for checking the return values of the called modules.

6.4.4 Description

Shortly described this module fetches a defined number of data records from the database and sends them to the Storage Module that writes them to the disk. The module gets the data record oid:s from the Index Module. This module also updates the index by calling the Index Module.

The Mirror Unit may fail while saving the image. In that case there is a started image in the active file. This module notices that because the global IMAGE_COMPLETE variable is false. When the unit has recovered and is in Mirror state again, the Control Module calls this module to save the image. But now the image saving is continued normally to the active file instead of starting a new image. Because the unit is fully recovered, it also has an up-to-date index that contains the valid file-id:s. So only those data records need to be written that are yet missing from the active file and hence from the image.

The functionality of this module is presented here as pseudocode. Error handling is omitted for clarity. It is discussed in the next subsection.

```
ISM_save_image(
    image_write_count
) {

   IF ( IMAGE_COMPLETE is true ) THEN {
      change active file to passive;
      change passive file to active;
      IMAGE_COMPLETE = false;
      send a special IMAGE_FIRST_RECORD to Storage Module;
   }

   counter = 0;

   WHILE ( counter < image_write_count AND
           ask next unwritten oid from index;
           return value was not IM_END ) {

      fetch from database the record with the oid index returned;
      send record to Storage Module;

      update file-id and offset in index;
      update global offset of the active file;

      counter = counter + 1;
   }

   IF ( index returned IM_END ) THEN {

      IMAGE_COMPLETE = true;
      send a special IMAGE_LAST_RECORD to Storage Module;
      return ISM_IMAGE_COMPLETED;
   }

   ELSE
      return ISM_IMAGE_NOT_COMPLETED;
}
```

The Control Module gives the image_write_count as a parameter when it calls this module. It determines the number of image records this module should write at one call if there are enough records left. The active and passive file-id:s are maintained in the global file status structure.

6.4.5 Error handling

Possible error situations and how they are handled:

- The given parameter image_write_count may be invalid (<= 0). In that case the module returns an error code INVALID_PARAMETER.
- The Index Module may return an error code ERROR if a fatal error occured when we asked for the oid. This module exits and returns the received error code to the Control Module.
- The database may return an error code OID_NOEXISTS indicating that it does not contain a data record with the specified oid. This module prints an error message and asks the Index Module for another oid.
- The database may return an error code INVALID_OID indicating that the specified oid is not a valid oid. This module prints an error message and asks the Index Module for another oid.
- The Storage Module may return an error code ERROR if a fatal error occured or SM_TCP_ERROR if the connection to the SSS failed. This module exits and returns the received error code to the Control Module.
- The Index module may return an error code ERROR if a fatal error occured in the update operation. This module exits and returns the received error code to the Control Module.

6.4.6 Alternative design solutions

The module could get the data records straight from the database, but we decided not to do so. While we have the index, we may as well make use of it and optimize a little. The Index Module returns oid:s of only such data records that have not been already written to the active file as log records. This cuts off some unnecessary write operations.

6.4.7 Size estimate

The LOC value for this module is about 90 lines.

6.5 Recovery Module

In this section implementation of the Recovery Module (RM) is described. RM is responsible of making recovery in the case of crash. RM is implemented in the Log Manager. In the case of recovery taking place the Control Module (CM) starts calling RM to perform needed functions. RM calls Index Module (IM) and Storage Module (SM) in different phases of recovery. RM also updates the database with log records received from SSS.

6.5.1 Interface

The function interface offered by RM is described here.

RM_init function initializes internal data structures for further use.

```
int RM_init()

Return value:
      SUCCESS         function succeeded
```

RM_recover function takes care of the recovering actions. This function is called by the Control Module in case of recovering.

```
int RM_recover()

Return value:
        SUCCESS       if the function succeeded
        INDEX_OK      when index is up-to-date
        DB_OK         when the whole database has been recovered
        ERROR         if the function failed
```

6.5.2 Referenced modules

RM uses function interfaces offered by the Index Module and the Storage Module. It also uses database interface function. The interface functions that are used are introduced in description section.

6.5.3 Internal data structures

Recovery Module contains a variable (Global) where is stored the phase that RM_recover is executing. RM uses also a variable in order to tell how many log read requests have been done to SM that not yet have been received (read_logs_waiting).

6.5.4 Description

All the functions implemented int the Recovery Module are described here in pseudocode. First are described all the static functions RM contains.

get_logs function calls the Storage Module in order to fetch logs from the SSS.

```
static int get_logs(file_id, offset, data_length)

 Return value:
      SUCCESS      if the function succeeded
      ERROR        if the function failed

static int get_logs(file_id, offset, data_length)
{
      int value;

      value = SM_log_read(file_id, offset, data_length);
      if value == SUCCESS
            change the offset amount of data_length;
            read_logs_waiting++;
            return SUCCESS;
      else
            return ERROR;
}
```

receive_logs function is called to receive earlier requested logs by get_logs function call. Function calls the Storage Module in order to receive logs from the SSS.

```
static int receive_logs(char *data, int *data_length)

 Return value:
```

```
      SUCCESS      if the function succeeded
      ERROR        if the function failed
      NO_RECORDS   if no records are received

static int receive_logs(data, data_length)
{
      int value;

      value = SM_receive_log(data, data_length);
      if value == SUCCESS
            read_logs_waiting--;
      else if value == NO_RECORDS
            return NO_RECORDS;
      else if index_saved_record received
            return
      else if value == SUCCESS
            return SUCCESS
      else
            return ERROR;
}
```

update_db function is called to update the database with the log received by the receive_log function call. Function calls the Index Module in order to update the index. It also updates the database with the database interface function.

```
static int update_db(data, data_length, fid, offset, action)

 Return value:
      SUCCESS      if the function succeeded
      ERROR        if the function failed

static int update_db(data, data_length, fid, offset, action)
{
      int value;

      value = IM_check_index(oid)
      if value == ERROR
          return SUCCESS;
      if action == UPDATE and not delete record
            value = IM_write_index(oid, fid, offset, len);
```

```
        else if delete record
             value = IM_index_delete(oid);
        if value == SUCCESS
              value = DB_image_write(trid, oper, oid,
                 toid, istype, len, source);
        if value == SUCCESS
              return SUCCESS;
        else
              return ERROR;
}
```

get_log_by_index function is called to fetch index information from the Index Module. Calling Index Module receives information of next log record missing from the database. get_log_by_index is called after log record indicating a index save is received by receive_log function.

```
static int get_log_by_index(id, fid, offset)

 Return value:
      SUCCESS      if the function succeeded
      ERROR        if the function failed
      IM_END       if all there is no more index entrys to read
                   into database

static int get_log_by_index(id, fid, offset)
{
      int value;

      value = IM_read_nextindex_RM(oid, fid, offset, len);
      if value == SUCCESS
            return SUCCESS;
      else if value == IM_END
            return IM_END
      else
            return ERROR;
}
```

RM_init function is called by the Control Module.

```
  int RM_init()
  {
```

```
            Global = READ;
            Initialize internal variables and structures;
            return SUCCESS;
}
```

RM_recover function is called by the Control Module.

```
int Global;

        int RM_recover()
        {
           switch(Global)
           {
           case GET:
               Global = UPDATE;
               loop LOOP_COUNT times
                 if get_logs(file_id, offset, len) != SUCCESS
                      return ERROR;
                 return SUCCESS;
           case UPDATE;
               Global = READ;
               loop LOOP_COUNT times
               {
                if (read_logs_waiting > 0)
                   if receive_logs() == SUCCESS
                         static offset = offset - data;
                         if IMAGE_COMPLETE == FALSE and
                              offset == 0
                           fid = passive_fid;
                           offset = passive_offset;
                         else if index_saved_log_record and
                            copied offset == current offset
                           Global = READ_IND;
                           break;
                         else if copied offset < current offset
                           offset = current offset;
                           Global = READ;
                           break;
                         else
                           if update_db(data, data_length,
                               fid, offset, UPDATE) != SUCCESS
```

```
                    return ERROR;
          else
             Global = READ;
             break;
          }
          if read_logs_waiting > 0 and
            global == READ_IND;
          {
              loop until read_logs_waiting == 0
              {
                 if receive_logs() == SUCCESS
                    if update_db(data, data_length,
                     fid, offset, UPDATE) != SUCCESS
                       return ERROR;
              }
          }
          if global == READ_IND
              return INDEX_OK;
          return SUCCESS;
     case READ_IND:
          loop LOOP_COUNT times
          {
           if get_log_by_index(id, fid, offset) == SUCCESS
               if get_logs(id, offset, data_length) == SUCCESS
                 if (read_logs_waiting > 0)
                    if receive_logs(data, length) == SUCCESS
                       update_db(data, data_length, fid, offset, READ_IND)
           else if IM_END is returned
             return DB_OK;
          }
          return SUCCESS;
     }
}
```

6.5.5 Error handling

If some of the Recovery Modules functions fails RM_recovery returns value ERROR to Control Module indicating an error. If this happens then CM should terminate execution because error could be caused by a number of reasons. For example a interface function to an other module has returned

ERROR.

6.5.6 Alternative design solutions

Amount of internal functions in module could be smaller.

6.5.7 Size estimate

```
Function                Size estimate (lines of code)


get_logs                40
receive_logs            30
update_db               40
get_log_by_index        30
RM_recover              80
RM_init 20

Total: 240 lines of code
```

6.6 Storage Module

Storage Module (SM) is responsible of the network traffic between the Log manager and the SSS. SM is a software module inside the Log manager that offers a simple interface that can be used to send and receive messages. The Log manager and the SSS use TCP communication.

Other modules in the Log manager use SM to send log or image records to SSS and to retrieve the latest index or some stored log records from SSS.

6.6.1 Interface

The function interface offered by SM has been described here in detail. The functions are outlined in pseudocode in the section 6.6.4.

SM_init function initializes all the internal structures of the Storage Module and creates the TCP connection to SSS. Control Module uses this function when the Log Manager is starting up.

```
int SM_init(
```

```
    void
)

Return value:
     SUCCESS           function succeeded
     ERROR             fatal error
     SM_TCP_ERROR      unable to create the TCP connection
```

SM_file_status_retrieve function retrieves the file status information from the SSS. This function is used by the Recovering Module. Function blocks until the file status has been received from the SSS.

```
int SM_file_status_retrieve(
    file_status_t *file_status
)

Parameters:
    file_status OUT  returns the file status information

Return value:
     SUCCESS             function succeeded
     ERROR               fatal error
     SM TCP_ERROR        error in TCP connection
```

SM_log_write function sends a log record to SSS. Log Saving Module uses this function.

```
int SM_log_write(
    char file_id,
    char* data,
    unsigned int data_length
)

Parameters:
    file_id      IN  identifier of the target file on SSS
    data         IN  data to be sent
    data_length  IN  length of the data
```

```
Return value:
     SUCCESS        function succeeded
     ERROR          fatal error
     SM_TCP_ERROR   error in TCP connection
```

SM_log_read function sends a read request of a log record to SSS. Recovering Module uses this function.

```
int SM_log_read(
    char file_id,
    unsigned int offset,
    unsigned int data_length
)

Parameters:
    file_id      IN  identifier of the target file on SSS
    offset       IN  the position of the record in the file
    data_length  IN  length of data to be read

Return value:
     SUCCESS       function succeeded
     ERROR         fatal error
     SM_TCP_ERROR  error in TCP connection
```

SM_index_write function stores the index structure to SSS. Index Module uses this function.

```
int SM_index_write(
   char file_id,
   char *data,
   unsigned int data_length
)

Parameters:
    file_id      IN  identifier of the target file on SSS
    data         IN  data to be sent
    data_length  IN  length of the data
```

```
Return value:
     SUCCESS       function succeeded
     ERROR         fatal error
     SM_TCP_ERROR  error in TCP connection
```

SM_index_read reads the latest index structure from the SSS. This function is used by the Index Module. Function blocks until it has read the whole index.

```
int SM_index_read(
    char file_id,
    char *data,
    unsigned int data_length
)

Parameters:
    file_id      IN   identifier of the target file on SSS
    data         OUT  returns the index structure in this buffer
    data_length  IN   the size of the data buffer

Return value:
     SUCCESS             function succeeded
     ERROR               fatal error
     SM_TCP_ERROR        error in TCP connection
```

SM_image_write function is used by the Image Saving Module to send image records (which are actually log records) to SSS. First Image Saving Module calls this function with type parameter IMAGE_FIRST_RECORD and after all the image records has been sent it calls this function with type parameter IMAGE_LAST_RECORD. All the image records are sent with type IMAGE_RECORD.

```
int SM_image_write(
    char file_id,
    char type,
    char *data,
    unsigned int data_length
)
```

```
Parameters:
    file_id      IN  identifier of the target file on SSS
    type         IN  type of the record to be written is one of the
                     followings:

                        IMAGE_START
                        IMAGE_RECORD
                        IMAGE_END

    data         IN  data to be sent
    data_length  IN  the length of the data

Return value:
     SUCCESS       function succeeded
     ERROR         fatal error
     SM_TCP_ERROR  error in TCP connection
```

SM_receive_log function is used by the Recovering Unit to receive answers to earlier sent log read requests. If SM hasn't received any log records then the function will return value NO_RECORDS.

```
int SM_receive_log(
    char *data,
    unsigned int data_length
)

Parameters:
    data         OUT  returns the data in this buffer
    data_length  IN   the length of the data buffer

Return value:
     SUCCESS             if function succeeded
     ERROR               fatal error
     SM_TCP_ERROR        error in TCP connection
     SM_NO_RECORDS       no log record to receive
```

6.6.2 Referenced modules

Storage Module doesn't use any other modules in the Log manager.

6.6.3 Internal data structures

Storage Module contains a variable where is stored the socket's file descriptor and a status information whether the connection to SSS has been created. SM also maintains some internal buffers where it temporarily stores some of the messages the SSS has sent.

6.6.4 Description

All the functions implemented in the Storage Module are described here in detail. SM contains some static functions which are described here first.

send_message function is the low-level function that constructs the message that is going to be sent to SSS and sends it. More specific description about the communication protocol used by SM and SSS has been given in Section 3.3.

```
int send_message(
    char action,
    char file_id,
    unsigned int offset,
    unsigned int data_length,
    char *data
)

Parameters:
    action      IN the action field is one of the followings:

                      SM_FILE_STATUS
                      SM_LOG_WRITE
                      SM_LOG_READ
                      SM_INDEX_WRITE
                      SM_INDEX_READ
                      SM_IMAGE_START
                      SM_IMAGE_END

    file_id     IN the identifier of the file the Log manager wants to
```

```
                    read or write

    offset      IN if the action field is SM_LOG_READ the offset
                   indicates the log record's position in the given file

    data_length IN if action is write action then this parameter tells the
                   length of the data to be written, if action is read
                   action then it tells how much data Log manager
                   wants to read (the parameter is ignored if action is
                   SM_FILE_STATUS of SM_INDEX_READ)

    data        IN contains the data to be sent if the action is write
                   action, otherwise it is NULL

Return value:
     SUCCESS       function succeeded
     ERROR         function failed
     SM_TCP_ERROR  error in TCP connection
```

The implementation of the send_message function in pseudocode level:

```
send_message(
    action,
    file_id,
    offset,
    data_length,
    data
) {

    construct header of the message;

    IF there is data in data parameter THEN
       add data to message;

    send message;

    IF error in TCP communication THEN
        return SM_TCP_ERROR;
    ELSE IF some other error THEN
        return ERROR;
```

```
    return SUCCESS;
}
```

receive_message function is used to receive a message from the SSS. The message can be an answer to a read request about a log record, latest index or the file status information. If the received message is not that kind of message the caller is expecting then the function stores the messages to internal buffers.

```
int receive_message(
    char *data,
    unsigned int data_length,
    int type,
    int block
)

Parameters:
    data        OUT function returns the data in this parameter

    data_length IN  the length of the data buffer

    type        IN  what kind of message the caller is
                    expecting, the value is one of the followings:

                       SSS_LOG
                       SSS_INDEX
                       SSS_FILE_STATUS

    block       IN  if true the function blocks until a message has
                    been received, otherwise function just checks if
                    there is any message waiting

Return value:
     SUCCESS          function succeeded
     ERROR            fatal error
     SM_TCP_ERROR     error in TCP connection
     SM_NO_MESSAGES   no message to receive
```

The implementation of the receive_message function in pseudocode level.

```
receive_message(
    data,
    data_length,
    type,
    block
}

    IF there is this kind of message in internal buffers THEN
        copy the message to data parameter;
        return SUCCESS;

    IF there is a message waiting THEN
        read the message;
        IF the message is rigth type of message THEN
           copy the message to data parameter;
           return SUCCESS;
        ELSE
           store received message to internal buffers;

    IF block THEN
        wait for message;
        IF the message is rigth type of message THEN
           copy the message to data parameter;
           return SUCCESS;
        ELSE
           store received message to internal buffers;
           wait message again...;
    ELSE
        return SM_NO_MESSAGES;

    IF in some situation a TCP communication error THEN
        return SM_TCP_ERROR;
    ELSE IF some other error THEN
        return ERROR;

}
```

Next here are the implementations of the interface functions in pseudocode level.

```
SM_init {

  initialize internal data structures;
  create the socket;
  create the TCP connection to SSS;

  IF error in connection establishment THEN
      return SM_TCP_ERROR;
  ELSE IF some other error THEN
      return ERROR;

  return SUCCESS;
}


SM_file_status_retrieve(
    file_status
) {

    rcode = send_message(SM_FILE_STATUS, 0, 0, 0, NULL);
    IF rcode != SUCCES THEN
        return rcode;
    rcode = receive_message(file_status,
                            sizeof(file_status_t),
                            SSS_FILE_STATUS,
                            1);

    return rcode;
)


SM_log_write(
    file_id,
    data,
    data_length
) {
    rcode = send_message(SM_LOG_WRITE,
                         file_id,
```

```
                      0,
                      data_length,
                      data);
    return rcode;
}


SM_log_read(
    file_id,
    offset,
    data_length
) {
    rcode = send_message(SM_LOG_READ,
                      file_id,
                      offset,
                      data_length,
                      NULL);
    return rcode;
}


SM_index_write(
   file_id,
   data,
   data_length
) {
    rcode = send_message(SM_INDEX_WRITE,
                      file_id,
                      0,
                      data_length,
                      data);
    return rcode;
}


SM_index_read(
    file_id,
    data,
    data_length
) {
```

```
    rcode = send_message(SM_INDEX_READ, file_id, 0, 0, NULL);
    IF rcode != SUCCESS THEN
        return rcode;
    rcode = receive_message(data, data_length, SSS_INDEX, 1);
    return rcode;
)


SM_image_write(
    file_id,
    type,
    data,
    data_length
) {

    switch(type) {
    case IMAGE_START:
        action = SM_IMAGE_START;
        break;
    case IMAGE_END:
        action = SM_IMAGE_END;
        break;
    default:
        action = SM_LOG_WRITE;
        break;
    }
    rcode = send_message(action, file_id, 0, data_length, data);
    return rcode;
}


SM_receive_log(
    data,
    data_length
) {
    rcode = receive_message(data, data_length, SSS_LOG,  0);
    return rcode;
}
```

6.6.5 Error handling

If some system function fails so that the SM can't do anything about it, for example if malloc fails because all the memory has been already used, then SM returns error code ERROR.

If there is an error with the TCP connection, for example if the connection has been closed by SSS, then the SM returns error code SM_TCP_ERROR. After this every time SM is called it will try to re-establish the connection and if it fails it will again return the SM_TCP_ERROR code.

6.6.6 Alternative design solutions

No other design solutions were considered when planning the Storage Module. The most essential part of the desing was to decide how much logic will be included in the Storage Module. In this approach it was decided to have very little inner functionality in the SM.

6.6.7 Size estimate

The estimated size of the Storage Module has been calculated in this section.

SM_init	50 lines
SM_file_status_retrieve	20 lines
SM_log_write	20 lines
SM_log_read	20 lines
SM_index_write	20 lines
SM_index_read	30 lines
SM_image_write	20 lines
SM_receive_log	20 lines
send_message	20 lines
receive_message	30 lines

Static structure definitions et cetera, approximately 50 lines.

Altogether the size of the Storage Module will be about 300 lines of C-code.

6.7 SSS

The Storage Module in the Log system communicates with the SSS Module to save image, index, and log records to disc. The recovering process involves fetching index from the SSS and building up a database image by loading the latest image and fetching data objects on demand using the loaded index.
The SSS module receives these requests from the Mirror- or Transient Unit and takes appropriate action. Below is a list of the requirements set to the SSS:

- storing log records to disk (received from either Mirror- or Transient Unit)
- storing database image to disk (received from either Mirror- or Transient Unit)
- storing index to disk (received from Mirror Unit)
- maintaining information about relevant files (file status information)
- loading latest complete index (when recovering)
- loading log records on demand

6.7.1 Interface

The only interface the SSS has to the other modules in the Log system is the TCP network interface used to communicate with the Storage Module on either Transient- or Mirror Unit.

The string received from the SM contains five parameters. The first parameter is the desired action. The second parameter is a file identifier telling where to store or from where to retrieve information. The third parameter tells the length of the data. The fourth parameter is the offset used when retrieving log records on demand. Last but not least is the data itself. The data is implemented as variable length (using the third parameter to tell how much there is to read). A closer description of the received string from SM can be found in section 3.3 on page 9.

6.7.2 Referenced modules

The SSS resides on a separate linux machine from the rest of the Log system. It therefore does not reference any other module directly, it only communicates with the SM through a TCP connection.

6.7.3 Internal data structures

The SSS maintains four files, an active log file, a passive log file, an index file, and a file containing file status information. These have been described more closely in sections 3.1.1, 3.2.3 and 3.2.4. The files are handled using a pointer array, where (for example) the first pointer (at position 0) points to the index file, the second pointer points to one of the log files and the third points to the other log file. This is because the incoming message (from SM) contains a *file_id* to tell where to store or from where to read data. The *file_id* can then be used to get the appropriate pointer from the pointer array.

When a new image is written the SSS updates its file status information by switching the active and passive log file identifiers. This means that if the active log file identifier was 1 and passive log file identifier was 2, active log file identifier will be set to 2 and passive log file identifier to 1.

When an unit recovers it fetches the file status information from the SSS. The file status information contains entries telling the active and passive log file identifiers (and some other information. see section 3.1.1), and the Recovering Unit then knows what identifier to give to the SSS to save or fetch information from corresponding file.

6.7.4 Description

The SSS module receives requests from the Log manager using TCP. The received message is a string containing five parameters which are extracted into variables. Closer descriptions of the variables are given in previous section and in section 3.3. These variables are then examined and appropriate action is taken based on their values.

Below is the SSS modules functionality described in pseudocode. (Variables are prefixed with a $)

```
initialize TCP connection;
```

```
initialize file status information;
open $active_log_file for read and write;

Loop forever;

wait for message;
parse message into variables($action,$file_id,$offset,$data_length,$data);

switch($action){
 case SM_FILE_STATUS{
  open file $file_id for read;
  return file status information, size of active log file and index file;
  close file $file_id;
 }
 case SM_LOG_WRITE{
  append $data with length $data_length to file $file_id;
 }
 case SM_LOG_READ{
  return log record with length $data_length
  from file $file_id at position $offset;
 }
case SM_INDEX_WRITE{
  IF index_write_begin THEN{
    open new file;
  }
  append $data with length $data_length to new file;
  IF index_write_end THEN{
    close new file;
    set $file_id to refer to new file;
    set $index_stored to true;
  }
 }
 case SM_INDEX_READ{
  open file $file_id for read;
  return contents of file $file_id;
  close file $file_id;
 }
 case SM_IMAGE_START{
  set $image_complete to false in file status information;
  close file $active_log_file;
  set $passive_log_file to $active_log_file in file status information;
```

```
  set $active_log_file to $active_log_file in file status information;
  open file $active_log_file for read-write;
  set $index_stored to false;
 }
 case SM_IMAGE_END{
  set $image_complete to true in file status information;
 }
}
```

6.7.5 Error handling

As earlier mentioned the SSS has minimum logic implemented so that it can be replaced with a dumb disk without having to make drastic changes to the rest of the Log system. The SSS is acting as a dumb disk, and has therefore allowed us to make the assumption that the SSS does not fail at any time.

The SSS assumes that incoming messages are correct. If a failure occures, the SSS does not send any error messages to the SM. Instead the SSS writes an error message to the console and commits suicide. This way the SM eventually faces a SM_TCP_ERROR which is considered a fatal error, forcing the Log manager to terminate program execution.

Errors that can occure and actions taken:

- failure in opening or closing a file - write error message to console and commit suicide
- failure in TCP communication - write error message to console and commit suicide
- failure in allocating memory - write error message to console and commit suicide

6.7.6 Alternative design solutions

The SSS will probably be replaced with a real dumb disk. This changes the design solution by completely moving the logic implemented in the SSS to the SM.

CM	200 LOC (Matti Johansson)
IM	700-800 LOC (Marko Moisio)
LSM	105 LOC (Minna Vasankari)
ISM	90 LOC (Minna Vasankari)
RM	240 LOC (Juha Saarelainen)
SM	300 LOC (Matti Johansson)
SSS	150 LOC (Patrick Doepel)

Table 2: The LOC estimates of the modules

6.7.7 Size estimate

The estimated size of the SSS module and TCP handling is about 150 lines of C-code. This includes the testing codes.

6.8 Total size estimate

The total size of the implemented software is estimated to be about 1700-1800 lines of C code, which includes the testing codes.

Each modules estimated LOC is shown in Table 2, and some modules contain a more specific LOC estimate in their subsection *Size estimate*.

7 Testing plan

In this section is outlined how the testing should be done. First each module is considered separately and descriped what should be tested and how and with methods the testing should be done.

The final subsection discusses the integration testing issues.

7.1 Control Module

The Control Module has very little inner functionality. The whole time it just calls the other modules using very simple function calls. It runs in a loop where it checks the status of the module, whether the unit is Mirror or Transient, and depending on the status it then calls the right modules.

Control Module can be tested in a way that it just prints to console the state of the unit and for example the functions it would call in that state. Rest of the testing can be done in the integration test phase.

If the Control Module receives a fatal error code from a module it calls (ERROR) or if some fatal error occures in the Control Module (malloc fails) then the Control Module will just kill the Log manager's thread of execution.

In this design the Control Module receives only fatal kinds of error codes. And there isn't any kind of error situations inside the Control Module that could be treated as non fatal ones. So in this implementation all the errors the Control Module notices are treated as fatal.

7.2 Index Module

Index Module must be able to provide accurate and up-to-date information for those modules that call it. Therefore it is important to test all the interface functions thoroughly. For testing purposes it's necessary to create a stub program that uses these the interface functions and all the actions that they provide. This stub program writes a log, so that it's later possible to verify that the Index Module is working correctly.

It's also important to test the initialization and reallocation functions as well as the functions that read/write the whole index from/to SSS. When the Index Module is tested independently, a local hard drive is used instead of SSS. There is need for a stub program that simulates the Storage Module.

7.3 Log Saving Module

This subssection presents a testing plan for the Log Saving Module. The conditions in conditional clauses and loops should be tested first with normal values and even slightly wrong values. The function prints out messages as the execution proceeds and these results are then compared with those expected. This testing method reveals typing errors in code.

Then we should test the general functionality, that is we should check that the module does what it promises. At least the following situations should be checked:

- Normal log saving in Mirror Unit

  - the amount of log records in the buffer is not a multiple of log_write_count
  - the amount of log records in the buffer is exactly a multiple of log_write_count
  - there are only some log records in the buffer
  - there is only one log record in the buffer
  - the buffer is empty

- Normal log saving in Transient Unit (index is present, meaning that is the unit was previously Mirror)
  - All the above situations should be tested again.
- Normal log saving, but unit state changes from Mirror to Transient in the middle
  - All the above situations should be tested again.
- Normal log saving in Transient Unit (index is not present, meaning that is the unit was previously Primary)
  - All the above situations should be tested again.
- All the error situations listed in section 6.3.5 on page 39 should be tested.

The testing is performed by using a simple main-function that calls this module in place of the real Control Module. It prints out the returned value. The modules this module uses are also replaced by simple test stubs. The values they return to this module are defined in advance depending on the situation that is being tested.

7.4 Image Saving Module

This subssection presents a testing plan for the Image Saving Module. The white-box testing should be done first. This means testing the conditional clauses and loops with normal values and slightly wrong values and comparing the results with those expected. This testing method reveals typing errors in code.

Then we should test the general functionality, that is we should check that the module does what it promises. At least the following situations should be checked:

- Normal image saving by multiple calls of this module
  - the amount of data records in the database is not a multiple of image_write_count
  - the amount of data records in the database is exactly a multiple of image_write_count
- Normal image saving, only one call needed
  - the database contains less data records than the number of image_write_count
  - the database contains data records exactly the number of image_write_count
  - the database is empty
- All the error situations listed in section 6.4.5 on page 44 should also be tested.

The testing is performed by using a simple main-function that calls this module in place of the real Control Module. It prints out the returned value. The modules this module uses are also replaced by simple test stubs. The values they return to this module are defined in advance depending on the situation that is being tested.

7.5 Recovery Module

This subsection presents a testing plan for the Recovery Module Testing the conditional clauses and loops with normal values and slightly wrong values and comparing the results with those expected are done first. Printing functions are used in debugging.

Then we should test the general functionality, that is we should check that the module does what it promises. At least the following situations should be checked:

- Normal recovering
  - The log file in SSS contains some log records before image_saved record
  - There's no log records before image_saved record in the log file

The testing could be done by using some test function that calls Recovery Module instead of Control Module. The modules this module uses are also replaced by simple test stubs. The values they return to this module are defined in advance depending on the situation that is being tested.

7.6 Storage Module

Storage Module has to be able to create right kind of messages to SSS. This can be tested by creating a little stub program that calls the functions of SM which then prints to console the whole message it would send to SSS. If there is time and need it would also be usefull to create a stub server that accepts real TCP connections and receives messages from the SM. It would then print to console the messages it receives and also send back those messages the SSS would send. The SM or the stub program using it would then print the answers to console. In this way it could be tested that the protocol in the SM has been implemented correctly.

The vulnerable point in the SM is the TCP connection. The SM returns SM_TCP_ERROR when there is some kind of error situation with the connection. After that the SM will try to re-establish the connection every time it is called. But in this implementation the modules calling the SM module will treat the SM_TCP_ERROR code as fatal error. After that fatal error code ERROR will be returned back to Control Module which will then kill the Log manager's thread of execution.

7.7 SSS

The SSS has to be able to take appropriate action when receiving a message from the SM. The SSS communicates with the SM using TCP so the testing of the SSS will probably be implemented as a normal function call where the TCP data packet is instead given as a function parameter. The calling test function prints the returned value to the screen. The values and contents of file status information,log files, and index file can be viewed from the corresponding files.

The TCP communication and handling of different actions will probably first be tested separately. When both the handling of different actions and TCP communication works, the two parts are combined and tested together.

7.8 Integration and validation testing

When every module has been tested separately it's possible to start integrating the modules and test how they work together. First for example only two modules could be joined, let's say Log Saving Module and Storage Module, and test that the the LSM uses the SM in a correct way. It is useful to start this kind of testing in an early stage for closely cooperating modules so that communication problems are detected early. The external interfaces are replaced by test stubs. One imitates the Log mediator by generating log records to the buffer. Another stub plays the part of the database. It receives log records from the Log Saving Module and the Recovery Module and returns records to the Image Saving Module. Some simple test stubs that replace the Index Module and the Storage Module are also needed at first, because most of the other modules use them. Many modules need also other kinds of test stubs for their internal testing. All the written test stubs must be documented and returned to the customer as part of the project results.

Finally we can join all the modules together and test the functionality of the whole Log system. The stub programs that implement the Log mediator and database are still needed. If every module in the Log system prints debug information to console then it would be quite easy to follow the functionality of the Log system and make sure that it works well in every situations. It should be tested that the Log system operates in a right way when the unit is Recovering Unit, Mirror Unit or Transient Unit. These three situations should be tested carefully and check that all the right actions are taken in every step.

8 Implementation restrictions

This section fixes some implementation matters. A list of coding style conventions that we should follow is presented first followed by some words about the programming language and environment.

8.1 Coding style

We should follow these naming conventions:

- The module files are named with the module name abbreviation. For example: ISM.h and ISM.c for the Image Saving Module.

- Function names begin with the module abbreviation and continue with _ and the function name. For example: ISM_save_image()

At least the following things should be commented about each function:

```
/************************************
 * Name of the function
 * Purpose of the function
 * Parameters
 * Returned values
 ************************************/
```

The size of the log record is now fixed, but we should **not** use the fixed number as such. Instead the size must always be figured out with the *sizeof*-operator.

The functions should return defined constant values, for example error codes instead of a plain zero or -1.

8.2 Implementation language and environment

The Log system is designed to be part of the RODAIN 2000 Database Prototype. The prototype runs on Chorus which is a real-time operating system. We will implement the Log system in Linux and use C as the programming language. The code is then ported to the Chorus environment. Some simple test stubs need to be written to replace the external interfaces so that testing can be carried out in the Linux environment.