# The Shore Storage Manager Programming Interface[1]

The Shore Project Group
Computer Sciences Department
UW-Madison
Madison, WI
Version 2.0


August 25, 1999

[1]This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

# Contents

# 1 Introduction

The Shore Storage Manager (SSM) is a package of libraries for building object repository servers and their clients. The core library in the package, `libsm.a`, is a multi-threaded system managing persistent storage and caching of un-typed data and indexes. It provides disk and buffer management, transactions, concurrency control and recovery. A second library, `libshorecommon.a`, provides many common utilities need for implementing both client and servers. The third library, `librpclib.a`, is a slightly modified version (for use in a multi-threaded environment) of the Sun RPC package.

We use the term *value-added-server* (VAS) to refer to systems built with the SSM. A VAS relies on the SSM for the above capabilities and extends it to provide more functionality. One example of a VAS is the Shore server, which extends the SSM to provide typed objects with permissions and ownership and organizes storage as a tree structured name-space. The overall role of the SSM in Shore is further described in *Shoring Up Persistent Applications*.

This document provides an overview the SSM facilities and interface. Details of the programming interfaces are presented in a set of manual pages. Where each facility is discussed, references are made to the appropriate manual pages. The introductory sections for these manual pages are:

- *Storage Manager proper*
- *Thread package*
- *Common utility classes*
- *Foundation classes*

The tutorial, *Writing Value-Added Servers with the Shore Storage Manager*, complements this document by explaining how to use the SSM to build an example client-server system.

The rest of this document is organized as follows. The first six sections describe the basic facilities provided by the SSM. Each sections has pointers to manual pages with details on using the facility. The final section describes how to write and compile a VAS and its clients.

## 1.1 Conventions

This document follows these notational conventions:

- File and path names are displayed in a `fixed size font`. The symbol, `SHROOT`, indicates the root directory of the Shore software installation.
- Reference to manual pages look like: *intro(ssm)*.
- Name of classes and methods are displayed in a `fixed size font`.

# 2 Initialization and Shutdown

The class `ss_m` is the core of the SSM interface. Creating an instance of `ss_m` starts the SSM. Destroying the `ss_m` instance causes the SSM to shutdown. Details on initialization and shutdown are available in *init(ssm)*.

When the SSM is started, it processes configuration options described below an initialized all the SSM data structures. This initialization includes allocation of the buffer pool. The buffer pool is located in shared memory, so the operating system must have shared-memory support to accommodate the size of the buffer pool. Next, the SSM checks the log to see if recovery is needed. If so, it follows the steps discussed in Section5.3.

## 2.1 Setting SSM Configuration Options

The SSM has a number of configuration options that must be set before it is started with the `ss_m` constructor. These options include such things as buffer pool sizes and location of the log. Many have default values. Those without default values must be set or the SSM will fail. Below we list all options and their default values.

- sm_bufpoolsize `no-default`

  This is the size of buffer pool in K-bytes. The minimum value is 64. Increasing the size will usually lower the amount of I/O done by the SSM.

- sm_logdir `no-default`

  The SSM currently uses OS files for log storage. This option sets the path name of the directory where log files will be placed.

- sm_logsize `10000` This is the maximum size, in K-bytes, of the log. All updates by transactions are logged, so the log size puts a limit on how much work any transaction can do. See Section5.1 for a discussion of log space usage.

- sm_logging `yes`

  This options controls whether or not logging is performed at all. Turning it off, by setting it to `no`, is used primarily for evaluating logging performance. No recovery or transaction rollback can be performed if logging is off.

- sm_diskrw `diskrw`

  This is the path name of the program forked by the SSM to perform asynchronous I/O. Usually this will point to `bin/diskrw` where Shore is installed.

- sm_locktablesize `64000`

  This is the number of buckets in the hash table used by the lock manager.

- sm_backgroundflush `yes`

  This option controls whether or not there is a background thread started to flushing the buffer pool periodically.

- sm_errlog `- (stderr)`

  This is the location to send error logging messages. The default is the standard error file. Other options are `syslogd` (to syslog daemon), or to a specific filename.

- sm_errlog_level `error`

  This is the level of error logging detail. Possible values (from least amount of logging to most amount) are `none emerg fatal alert internal error warning info debug`.

## 2.2 Adding VAS-Specific Options

In addition, a VAS, will often have options of its own that need to be set. The SSM provides an options facility, *options(common)* for this purpose. Included with the option facility are functions to set options from the program command line and from files containing configuration information.

A discussion of how to use the options facility is given in the tutorial.

# 3 Storage Facilities

The SSM provides a hierarchy of storage structures. A description of each type of storage structure is given below, followed by a description of the identifiers used to refer to them.

## 3.1 Devices

A *device* is a location, provided by the operating system, for storing data. In the current implementation, a device is either a disk partition or an operating system file. A device is identified by the name used to access it through the operating system. Each device is managed by a single server. A device has a quota. The sum of the quotas of all the volumes on a device cannot exceed the device quota. *Note*: devices are currently limited to containing only one volume.

The device management interface is part of class `ss_m` and is described in *device(ssm)*.

For each mounted device, the server forks a process called `diskrw` (determined by the `sm_diskrw` option) to perform asynchronous I/O on the device. These processes communicate with the server through sockets and shared memory, so your OS must be configured with shared memory support.

## 3.2 Volumes

A *volume* is a collection of file and index storage structures (described below) managed as a unit. All storage structures reside entirely on one volume. A volume has a quota specifying how must large it can grow. Every volume has a dedicated B+-tree index, called the *root index* to be used for cataloging the data on the volume.

The volume management interface is part of class `ss_m` and is described in *volume(ssm)*.

## 3.3 Files of Records

A *record* [1] is an un-typed container of bytes, consisting of a *tag*, *header* and *body*. The tag is a small, read-only location that stores the record size and other implementation-related information. The header is variable length (limited to what will fit on a page) location for a VAS to store meta-information about the record (such as its type). The body is the primary data storage location and can range in size from zero bytes to 4-GB. A record can grow and shrink in size by operations that append and truncate bytes at the end of the record.

A *file* is a collection of records. Files are used for clustering records and have an interface for iterating over all the records they contain. The number of records that a file can hold is limited only by the space available on the volume containing the file. The minimum size of a file is 64K-bytes (8 pages). We are working on ways to reduce this to 8K, but in either case, using a file to store a collection containing only a few small records will waste space.

Methods for creating/destroying files and creating/destroying/modifying records are part of class `ss_m` and described in *file(ssm)*. There is a `pin_i` class for pinning records for reading and modifying. This class is documented in *pin_i(ssm)*. There are the classes `scan_file_i` for iterating over the records in a file, and `append_file_i` for appending records to a file. Both are described in *scan_file_i(ssm)*.

## 3.4 B+tree Indexes

The *B+tree index* facility provides associative access to data. Keys and their associated values can be variable length (up to the size of a page). Keys can be composed of any of the basic C-language types plus variable length character strings. A bulk-loading facility is provided. The number of key-value pairs that an index can hold is limited only by the space available on the volume containing the index. The minimum size of a B+tree index is 8K-bytes (1 page).

Methods for index operations are part of class `ss_m` and described in *btree(ssm)*. There is `scan_index_i` class for iterating over a range of keys in the index This class is documented in *scan_index_i(ssm)*.

## 3.5 R*Tree Indexes

An *R-Tree* is a height-balanced tree structure designed specifically for indexing multi-dimensional spatial objects. It stores the *minimum bounding box* (with 2 or higher dimension) of a spatial object as the key in the leaf pages. The current implementation in SHORE is a variant of R-Tree called *R*-Tree* [BKSS], which improves the search performance by using a better heuristic for redistributing entries and dynamically reorganizing the tree during insertion. Currently, only 2-dimensional R*-trees with integer coordinates are supported by the SSM. A bulk-loading facility is provided. The number of key-value pairs that an index can hold is limited only by the space available on the volume containing the index. The minimum size of an R*tree index is 64K-bytes (8 pages).

The R*-Tree implementation stores [key, value] pairs, where the key is of type `nbox_t`. and the value is of type `vec_t`. A 2-D `nbox_t` is a rectangle which stores coordinates in the order of

[1] The term record is used to distinguish them from objects (which have type and methods).

x_low, y_low, x_high, y_high (lower left point and higher right point). Currently, only integer values are supported for the coordinates.

Methods for R*-tree index operations are part of class `ss_m` and described in *rtree(ssm)*. There is `scan_rt_i` class for iterating over a range of keys in the index This class is documented in *scan_rt_i(ssm)*.

### 3.6 Identifiers

Volumes, files, records and indexes all have identifiers. There are two broad categories of identifiers: *logical* and *physical*. Logical IDs are location-independent meaning there is a level of indirection for mapping the ID to a physical location. Physical IDs are location-dependent, meaning they refer to the physical location (usually location on disk) of the referenced object. *Although the SSM has both physical and logical ID versions of its interface, only the version using logical IDs is supported at this time.*

Volume IDs are a globally unique, 8-byte long ID called `lvid_t` described in `lid_t(common)`. File, record and index IDs are identified by a serial number, `serial_t`, unique to the volume containing them. Serial numbers are currently 4 bytes long, but we plan to make them 8 bytes long in the future. The complete ID for a file, index or record is a combination of the volume ID and serial number call `lid_t` and described in *lid_t(common)*. Once the object a serial number identifies is destroyed, the serial number will never be reused.

The purpose of having two-part IDs (volume ID, serial number) is to allow *local* (ie. intra-volume) references to be stored using only the serial number. This can significantly reduce the size of databases containing records with lots of pointers. To keep *remote* (ie. inter-volume) references just as short, special serial numbers for remote references are used.

Methods for operating on IDs and generating remote references are described in *lid(ssm)*.

## 4 Transaction Facilities

As a database storage engine, the SSM provides the atomicity, consistency, isolation, and durability (often referred to as ACID) properties associated with transactions. More information on transaction processing issues can be found in the book *Transaction Processing: concepts and techniques* [GrRe].

### 4.1 Transactions

A transaction bounds an atomic and set of operations on records, files, and indexes. The manual page, *transaction(ssm)*, describes methods for beginning, committing and aborting transactions. Updates made by committed transactions are guaranteed to be reflected on stable storage, even in the event of software or processor failure. Updates made by aborted transactions are rolled back and are not reflected on stable storage.

Although nested transactions are not provided at this time, the notion of save-points are. Save-points delineate a set of operations that can be rolled back without rolling back the entire transaction. The interface is described in *transaction(ssm)*.

## 4.2 Concurrency Control

Transactions are also a unit if isolation. Locking is provided by the SSM as a way to keep one transaction from interfering with another. When designing the SSM interface there was considerable debate on whether the SSM should automatically do locking or instead require the VAS writer to obtain appropriate locks. We chose to have the SSM automatically obtain locks, but the SSM interface does provide methods allowing locks to be explicitly acquired. See *lock(ssm)* for details.

The SSM performs concurrency control using the standard hierarchical two-phase locking protocol [GrRe]. The lock hierarchy for files is: volume, file, page, slot-containing-record. The lock hierarchy for indexes is: volume, index, key-value.

Chained transactions are also provided. Chaining involves committing a transaction, retaining its locks, starting a new transaction and giving the locks to the new transaction.

# 5 Crash Recovery Facilities

The crash recovery facilities of the SSM can be divided into: logging, checkpointing, and recovery management.

## 5.1 Logging

Updates performed by transactions are logged so that the can be rolled back (in the event of a transaction abort) or restored (in the event of a crash). Both the old and new values of an updated location are logged. This allows a steal, no-force buffer management policy, which means the buffer manager is free to write dirty pages to disk at any time and yet does not have to write dirty pages for a a transaction to commit.

The log is a location holding log records. Currently the log is stored in Unix files in a special directory (we plan to support using a raw device partition in the future). The size and location of the log is determined by configuration options described in Section2.

The proper value for the size of the log depends upon the expected transaction mix. More specifically, it depends on the age of the oldest (longest running) transaction in the system and the amount of log space used by all active transactions. Here are some general rules to determine the amount of free log space available in the system.

- Log records between the first log record generated by the oldest active transaction and the most recent log record generated by any transaction cannot be thrown away.
- Log records from a transaction are no longer needed once the transaction has committed or completely aborted and all updates have made it to disk. Aborting a transaction causes log space to be used, so space is reserved for aborting each transaction. Enough log space must be available to commit or abort all active transactions at all times.
- Only space starting at the beginning of the log can be reused. This space can be reused if it contains log records only for transactions meeting the previous rule.

- All `ss_m` calls that update records require log space twice the size of the space updated in the record. All calls that create, append, or truncate records require log space equal to the size created, inserted, or deleted. Log records generated by these calls (generally one per call) have an overhead of approximately 50 bytes.
- The amount of log space reserved for aborting a transaction is equal to the amount of log space generated by the transaction.
- When insufficient log space is available for a transaction, the transaction is aborted.
- The log should be at least 1 Mbyte.

For example, consider a transaction T1 that creates 300 records of size 2,000 bytes, writes 20 bytes in 100 objects, and is committed. T1 requires at 615 Kbytes for the creates and 9 Kbytes of log space for the writes. Since log space must be reserved to abort the transaction, the log size must be over 1.248 Mbytes to run this transaction. Assuming T1 is the only transaction running in the system, all the log space it uses and reserves becomes available when it completes. If another transaction, T2, is started at the same time as T1, but is still running after T1 is committed, only the reserved space for T1 is available for other transactions. The portion of the log used by T1 and T2 is not available until T2 is finished.

Transactions that fail because of insufficient log space are commonly those that load a large number of objects into a file during the creation of a database. A solution to this problem is to load the file in a series of smaller transactions. When the last transaction is committed, the load is complete. If the load needs to be aborted, a separate transaction is run to destroy the file.

## 5.2 Checkpointing

Checkpoints are taken periodically by the SSM in order to free log space and shorten recovery time. Checkpoints are "fuzzy" and can do not require the system to pause while they are completing.

## 5.3 Recovery

The SSM recovers from software, operating system, and CPU failure by restoring updates made by committed transactions and rolling back all updates by transactions that did not commit by the time of the crash. when an instance of class `ss_m` is created.

Recovery has three phases:

- Analysis

  During the analysis phase the log is scanned to determine what transactions were active and which devices were mounted at the time of the failure.

- Redo

During the redo phase the devices are remounted and the log is scanned starting at a location determined by analysis. The operation recorded in each log record is redone if necessary. After redo, the database is in the state it was just before the crash.

- Undo

  During the undo phase, all active transactions at the time of the crash are undone. The devices are dismounted, and a checkpoint is taken.

The time it takes for recovery depends on several factors, including the number of transactions in progress at the time of the failure, the number of log records generated by these transactions, and the number of log records generated since the last checkpoint.

# 6 Thread Management

Providing the facilities to implement a multi-threaded server capable of managing multiple transactions is one of the distinguishing features of the SSM. Other persistent storage systems such as the Exodus Storage Manager (http://www.cs.wisc.edu/exodus/)) only support writing clients that run one transaction at a time and are usually single-threaded.

The Shore Thread Package is documented in *intro(sthread)* All threads are derived from the abstract base class `sthread_t`. Any thread that uses the SSM facilities must be derived from class `smthread_t` described in *smthread_t(ssm)*

A discussion of how to use threads facility is given in the tutorial.

Any program using the thread package automatically has one thread, the one running `main()`. In addition, the SSM starts one thread to do background flushing of the buffer pool and another to take periodic checkpoints.

We have also implemented some extensions to the thread package. These are not formally part of the thread package, but we've found them useful enough in building the SSM and the Shore VAS to warrant including them as part of the documented interface.

## 6.1 Latches

Latches are a read/write synchronization mechanism for threads, as opposed to locks which are used for synchronizing transactions. Latches are much lighter weight than locks, have no symbolic names, and have no deadlock detection. Latches are described in *latch_t(common)*

## 6.2 Thread-Protected Hash Tables

The Resource Manager, `rsrc_m`, template class manages a fixed size pool of *shared resources* in a multi-threaded environment. The `rsrc_m` protects each resource with a latch and uses them to enforce a protocol in which multiple threads have consistent and concurrent access to the shared resources. For instance, the Shore buffer manager uses rsrc_m to manage buffer control blocks. The `rsrc_m` is implemented using a hash table. When a entry needs to be added and the table is full, on old entry is removed based on an LRU policy. More details can be found in *rsrc(common)*

## 7 Error Handling

Errors in the SSM (and the rest of Shore) are indicated by an unsigned integer encapsulated in a class that includes stack traces and other debugging aids. The class is `w_rc_t` (commonly `typdef`ed to `rc_t`) described in *rc(fc)*. It is the return type for most SSM methods. When linking with a debugging version of the SSM (compiled with `#define DEBUG`), the destructor of an `w_rc_t` object performs auditing to verify that it was checked at least once. If not checked, the destructor calls `w_rc_t::error_not_checked` which prints a warning message. An `w_rc_t` is considered checked when any of its methods that read/examine the error code are called, including the assignment operator. Therefore, simply returning an `w_rc_t` (which involves an assignment) is considered checking it. Of course, the newly assigned `w_rc_t` is considered unchecked.

The domain of error codes is an extension of the Unix error codes found in `#include <errno.h>`. Each layer of the Shore software adds its own extension to the domain. The following layers have error codes which may be returned by SSM methods:

- Storage Manager proper, see *errors(ssm)*.
- Thread package, see *errors(sthread)*.
- Configuration options package, see *options(common)*.
- Foundation Classes, see *intro(fc)*.

Further discussion of error handling is given in the tutorial.

VAS writers may wish to use the error handling facility to add there own error codes. See *error(fc)* for more details.

## 8 Communication and RPC Facilities

Clients (applications) need a way to communicate with servers. The release of the SSM does not contain any communication or RPC facilities for client-server communication. However, the Version 2.0 release contains a version of the publicly available Sun RPC package, modified to operate with the SSM's thread package. The Shore value-added server uses this package. The tutorial contains an example of using the RPC package and a discussion of its integration with the thread package.

One of the goals of Shore is that clients only need to communicate with a server on their local machine. The SSM in the local server cooperates with other remote SSMs to cache data locally. Our work on this capability is not complete. Eventually, the communication system we built to support this will also be available to VAS writers.

# 9 Miscellaneous Facilities

## 9.1 Statistics

The SSM keeps many statistics on its operation such as lock request and page I/O counts. Details are available in *statistics(ssm)*. A utility for formatted printing of these statistics is described in *statistics(fc)*.

## 9.2 Sorting

The SSM has sorting facilities, however they are still under development, so the interface may change. Descriptions of the sorting facilities can be found in *sort_stream_i(ssm)* and *sort(ssm)*

## 9.3 Data Vectors

A data vector is an array of pointers-length pairs to in-memory data. The array can be arbitrarily long, and methods are provided to comparing and copying data. They are further described in *vec_t(common)*.

Data vectors reduce the number of parameters in many SSM methods be combining pointer and length information. More importantly, they allow more flexibility in structuring data. For example, consider record that is stored in memory in three parts. To create the record, all that is necessary is to build a vector pointing to the three parts and pass the vector to the `ss_m::create_rec` method.

# 10 Writing and Compiling a VAS and Client

This section discusses some of the general issues in compiling and linking with the SSM libraries. Still, the best way to learn about writing and compiling a VAS and client is to read the tutorial, *Writing Value-Added Servers with the Shore Storage Manager*.

## 10.1 Include Files and Libraries

All of the include files needed to build servers and clients are located in `SHROOT/include`. Any server code using the SSM should include `sm_vas.h`. Since clients do not need all of the SSM functionality, they need only include `sm_app.h`. The RPC package include files are located in `SHROOT/include/rpc` and are usually included with this line:

```
#include <rpc/rpc.h>
```

All of the libraries needed to build servers and clients are located in `SHROOT/lib`. Clients only need `libshorecommon.a`. Servers need both `libsm.a` and `libshorecommon.a` The RPC package library is `librpclib.a`.

There are two pre-compiled version of these libraries. The are included in the *debugging* and *no-debugging* binary releases. The debugging version not only includes symbol table information (`-g` option to `gcc`), but also has considerable additional auditing and assert checking code.

This includes code that audits data pages whenever an update is made, performs monitoring to detect thread stack overflow, and checks over 1,400 additional assertions. See the Shore Release document for more information on these releases.

*Note:* when compiling for linkage with the debugging release, the compiler flag `-DDEBUG` should be used.

## 10.2 Template Instantiation

The SSM uses a number of templates One of the issues that is often confusing is controlling template instantiation. All of the template instantiations needed by the SSM are already included in the libraries.

However, due to a bug in `gcc` 2.6.* (supposedly to be fixed in 2.7.0), it is possible to have problems during linking due to multiple definitions of template code. To avoid this, and to have smaller executables, we use the `gcc` option `-fno-implicit-templates` in the `Makefile` from the tutorial example. This causes `gcc` not to emit any template code unless the template is explicitly instantiated.

Here is an example of explicit instantiation from the tutorial

```
#ifdef __GNUG__
// Explicitly instantiate lists of client_t.
template class w_list_t<client_t>;
#endif
```

## 10.3 Other Example Code

The SSM has been used to build a number of value-added servers. Some of these are publicly available. You may find these helpful in writing your own.

- Shore Server

  The Shore Server is the server for the Shore object repository. The Shore Server actually has two interfaces. One is used by SDL applications and the other is the NFS interface. The Shore Server code is available in `src/vas`.

- SSM Testing Shell

  The SSM testing shell is a server with a TCL interface designed to test the SSM. The code is available in `src/sm/ssh`. No documentation is available yet.

- Paradise

  Paradise is a GIS system still under development. It will be publicly available in the future. See http://www.cs.wisc.edu/paradise/ for more information.

# References

[BKSS] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. “The R*–Tree: An Efficient and Robust Access Method for Points and Rectangles”. Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp. 322-331.

[GrRe] Gray, J., Reuter, A. Transaction Processing: concepts and techniques, 1993.