Getting Started Writing a Value-Added Server with the Shore Storage Manager[1]

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

May 31, 2008

[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.

1 Introduction

This tutorial explains, through the use of a simple example, how to get started writing a value-added server (VAS) using the Shore Storage Manager (SSM) programming interface. This document is a guide to the companion program in the directory `hello/` located in the directory where this document is found.

1.1 Goals

1.1.1 What this Tutorial Is

This tutorial illustrates writing a VAS, including

- Configuring the SSM using the options package,
- Starting the SSM,
- Using the storage files and records,
- Simple transactions,
- Handling errors, and
- Shutdown.

1.1.2 What this Tutorial Is Not

- A general introduction to the SSM, its goals, structure or status,
- A demonstration of all the features of the SSM, or
- A tutorial on multi-threaded programming in general.

1.2 The Hello Example

The example used throughout this tutorial is a simple "hello world" program that initializes a volume, creates a file and a record in that file, and writes and reads back the record.

1.2.1 What the Hello Example Demonstrates

- Storage facilities

  The program creates a file on a single volume, writes to the file or reads from the file, based on the command-line options given to the program.

- Threads

  The program is multi-threaded in that it uses the SSM thread to manage the file and record that it creates, as is required for any use of the storage manager. While this example uses only one such thread, a VAS can create as many threads as are required

for handling its clients requests. Shared state among the threads is protected using the synchronization mechanisms provided by the thread package.

- Configuration options

  The server use the SSM's configuration options package to read configuration information from configuration files and the program command line arguments.

1.2.2 What the Example Does Not Demonstrate

This example does not demonstrate all of the features of the SSM. In particular:

- Appending/truncating records,
- Pinning large records (longer than 8K bytes),
- Gathering statistics,
- Creating and using an index,
- Bulk-loading an index, or
- Sorting.

1.3 Tutorial Organization

This tutorial walks through the example program sources in detail. The source resides in `hello/hello.cpp` in the directory where this documentation is installed.

The rest of this tutorial is organized as follows. First is a section describing return values from storage manager operations. Next comes a section discussing the storage manager operations demonstrated by the example, with reference to the code in the program, and with references to some of the distributed SSM manual pages. Finally, a section describes how to compile and run the example.

2 Error Codes

Most SSM methods return an error code object of type `w_rc_t` (usually typedef'ed as `rc_t`. It is important to always check the return values of these methods. To help find places where return codes are not checked, the `w_rc_t` destructor has extra code (when compiled with `DEBUG` defined) to verify that the error code was checked. 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 macros `W_DO` and `W_COERCE`, declared in `w_rc.h`, are helpful in checking return values and keeping the code concise. The `W_DO` macro takes a function to call, calls it and checks the return code. If an error code is returned, the macro executes a `return` statement returning

the error. The W_COERCE does the same thing except it exits the program if an error code is returned by the called function.

For more details on error checking and the rc_t, w_rc_t structures, see the manual pages rc(fc).

3 Getting Started

3.1 Configuration Options

A number of SSM configuration options must be set before the SSM is started with the ss_m constructor. In addition, most servers, including the hello example server, will have options of their own that need to be set. The SSM provides an option facility, *options(common)*, for this purpose. Included with the option facility are functions to find options on the program command line and from files of configuration information.

In hello.cpp, main creates a 3-level option group (levels will be discussed shortly), and adds the server's options to the group with a call to ss_m::setup_options.

```
cout << "processing configuration options ..." << endl;
const int option_level_cnt ( 3 );
option_group_t options(option_level_cnt);
```

Now the option group is complete so we call init_config_options to initialized the options' values:

```
W_COERCE(init_config_options(options));
```

The init_config_options is a static server (not SSM) function that initializes option values.

The class w_ostrstream used here is a backward-compatibility class that implements the old strstream capabilities; it is in fc/w_strstream.h.

```
rc\_t
init_config_options(option_group_t& options)
{
    W_COERCE(options.add_class_level("example"));         // program type
    W_COERCE(options.add_class_level("server"));          // server or client
    W_COERCE(options.add_class_level("hello"));               // program name

    rc_t rc;        // return code
    const char* opt_file = "exampleconfig";         // option config file

    // have the SSM add its options to the group
    W_DO(ss_m::setup_options(&options));

    // read the config file to set options
```

```
    {
        w_ostrstream      err_stream;
        option_file_scan_t opt_scan(opt_file, &options);

        // scan the file and override any current option settings
        // options names must be spelled correctly
        rc = opt_scan.scan(true /*override*/, err_stream, true);
        if (rc) {
            char* errmsg = err_stream.str();
            cerr << "Error in reading option file: " << opt_file << endl;
            cerr << "\t" << errmsg << endl;
            if (errmsg) delete errmsg;
            return rc;
        }
    }

    // check required options
    {
        w_ostrstream      err_stream;
        rc = options.check_required(&err_stream);
        char* errmsg = err_stream.str();
        if (rc) {
            cerr << "These required options are not set:" << endl;
            cerr << errmsg << endl;
            if (errmsg) delete errmsg;
            return rc;
        }
    }

    return RCOK;
}
```

The first thing this function does is add classification level names for the option group. The option group used for the example has 3 levels. First level is the system the program belongs to, in this case `example`. The second is the type of program, in this case `server`. The third is the filename of the program executable, which is also `hello`. The classification levels allow options to be set for multiple programs with a single configuration file.

After setting the level names, `init_config_options` reads the file `exampleconfig` to scan for options. Then the command line is searched for any option settings so that command line settings override those in the configuration file. Any option settings on the command line are removed by changing `argc` and `argv`.

Now the main program reads the command-line options to determine if it is to read an existing file (-r) or to initialize a file (no command-line options):

For more details on options, see the manual pages `options(common)`, `ssm_options(ssm)`,

`init(ssm)`.

Now the main program reads the command-line options to determine if it is to read an existing file (-r) or to initialize a file (no command-line options):

```
bool init_device(true);
if(argc > 2)
{
    usage(options);
    ::exit(1);
} else if(argc==2) {
    // -r for read-only (already exists)
    if (strcmp(argv[1], "-r") == 0) {
        init_device = false;
    } else {
        usage(options);
        ::exit(1);
    }
}
```

3.2 SSM Initialization, Threads, Shutdown

Once all of the configuration options have been set, the SSM can be started. The SSM is started by constructing an instance of the `ss_m` class. The instance is global, but creation of the instance must be done inside a thread because one of the things the `ss_m` constructor does is perform recovery if necessary. (Recovery will be necessary if a previous server process crashed before successfully completing the `ss_m` destructor.) Recovery, like all storage manager operations, use thread-synchronization primitives and must be performed in the context of a storage manager thread. This is so you can write a server that handles multiple simultaneous activities, such as responding to input from the terminal, listening for new connections from clients, and processing RPCs from clients. Any one of these activities can become blocked while acquiring a lock or performing I/O, for example. By assigning activities to threads, the entire server process does not block; only threads block.

For more details on Shore (generic) threads, see the manual pages `sthread_t(sthread)`.

For more information about Shore (generic) threads synchronization primitives, see the manual pages `scond_t(sthread), sevsem_t(sthread), smutex_t(sthread)`.

In the hello example, all operations are performed in a single thread, which is an instance of a `startup_smthread_t` class. This is an extension of the `smthread_t` class, a Storage Manager thread. The Storage Manager thread class has a virtual `run` method that is called when the thread is *fork*ed.

The server thus specializes thread classes for performing storage manager operations by writing a `run` method for the specialized class's specific purpose.

For more details on Storage Manger threads, which are built on the generic Shore threads, see the manual pages `smthread_t(ssm)`.

```
    startup_smthread_t *doit = new startup_smthread_t(init_device);

    if(!doit) {
        W_FATAL(fcOUTOFMEMORY);
    }

    cerr << "forking ..." << endl;

    W_COERCE(doit->fork());
```

The first code to be executed by any newly forked SSM thread is its `run` method:

```
void
startup_smthread_t::run()
{
    rc_t rc;
    cout << "Starting SSM and performing recovery ..." << endl;
    ssm = new ss_m();
    if (!ssm) {
        cerr << "Error: Out of memory for ss_m" << endl;
        return;
    }

    lvid_t lvid;  // long ID of volume for storing hello world data
    vid_t handle;  // short ID of same volume
    W_COERCE(init_device_and_volume(_init_device, lvid, handle));

    ...

    cout << "\nShutting down SSM ..." << endl;
    delete ssm;
}
```

Once the SSM is constructed, `run` calls `setup_device_and_volume` to initialize the device and volume, described in this next subsection.

After initializing the device and volume, the program either writes to the volume or reads from the volume, as determined by the command-line argument (-r means read from the volume), as described in the in the subsequent section below.

The program creates a file and a record in that file, and stores the identifier for the record (`rid_t`) in a

Shutting down the SSM involves ending all threads, except the one running `main` and then destroying the `ss_m` object. In this example, there is only the main thread, so it does not have to await the finishing of other threads.

3.3 Initializing Devices and Volumes

All files and indexes are located on a volume. For the example, only one volume is used. This volume is located on the single device mounted, the name of which is hard-coded in this example, but which can be located with a configuration option, in a file given in a configuration option, by command-line options, etc.

Before creating a storage volume, the *device* (raw device or Unix file) where it will reside must be initialized. A flag to `init_device_and_volume()` indicates whether the device is to be initialized (ie. we're starting from scratch), in which case the flag will be *true*, or if there is already an initialized device, in which case it will be *false*.

If initialization is needed, the device is formatted using `ss_m::format_dev`. To begin using the device, we mount it with `ss_m::mount_dev`. We then generate a long volume id for a volume with `ss_m::generate_new_lvid`. and create a volume on the device with `ss_m::create_vol`.

```
rc_t
init_device_and_volume(bool _init_device, lvid_t& lvid, vid_t &handle)
{
    u_int        vol_cnt;
    devid_t        devid;

    vid_t         vid(1);
    if(_init_device) {
        cout << "Formatting and mounting device "
             << device_name << "..." << endl;
        // format a 1000KB device for holding a volume
        W_DO(ssm->format_dev(device_name, 1000, true));
        cout << " ... " << device_name << " formatted." << endl;

        cout << "Generating a new long volume id..." << endl;
        W_DO(ssm->generate_new_lvid(lvid));
        cout << " ... long volume id =" << lvid << endl;
        //
        // Just for the sake of simplicity, we'll mount the "main"  or "root"
        // (and, in this case, only) volume as volume 1
        //
        cout << "Mounting device " << endl;
        W_DO(ssm->mount_dev(device_name, vol_cnt, devid));
        cout << " ... " << device_name << " contains " << vol_cnt
                    << " volumes." << endl;

        // create the new volume (1000KB quota)
        cout << "Creating a new volume on the device ..." << endl;
        W_DO(ssm->create_vol(device_name, lvid, 1000, false/*skip raw init*/,
                    vid ));
```

```
        cout << " ... volume " << vid << " created. " << endl;

    } else {
        cout << "Using already existing device: " << device_name << endl;
        // mount already existing device
        w_rc_t rc = ssm->mount_dev(device_name, vol_cnt, devid, vid);
        if (rc) {
            cerr << "Error: could not mount device: " << device_name << endl;
            cerr << "   Did you forget to run with -i the first time?"
                << endl;
            return rc;
        }
        cout << " ... " << device_name << " contains " << vol_cnt
                    << " volumes." << endl;
    }

    handle = vid;
    return RCOK;
}
```

If no initialization is needed, we mount the device and find its volume handle through the mount.

These actions are performed outside the context of a transaction.

For more details on Storage Manger devices, volumes, see the manual pages `device(ssm)`, `volume(ssm)`.

4 Storage Structures and Transactions: Writing and Reading the Hello Record

The first time the hello program is run, the volume is formatted and contains no files or indexes. The main program chooses, based on the command-line option, whether to create a file and write a record in the file (`write_hello`), or to locate an already-written record (`find_hello`). In either case, it then reads the record and prints its contents (`say_hello`).

To locate the record when the -r option is used, the program stores the record identifier `rid_t` of the record in an index when the record is created, using a well-known key, "HI". The index used is a well-known index that each volume contains, called its *root index* (described below).

```
void
startup_smthread_t::run()
{
    ...
```

```
    rid_t rid;
    vec_t KEY("HI",2);
    if(_init_device) {
        W_COERCE(write_hello(handle, KEY, rid));
    } else {
        W_COERCE(find_hello(handle, KEY, rid));
    }
    W_COERCE(say_hello(rid));

    ...

}
```

4.1 Transaction and Lock Management

All operations on data managed by the SSM must be done within the scope of a transaction, so `write_hello, find_hello` and `say_hello` wrap all their data operations with with `begin_xct` and `commit_xct()`.

The SSM automatically acquires locks when data is accessed, providing serializable transactions. The hello program relies on the automatic locking done by the SSM. The `concurrency_t argument` to many of the SSM methods and the `lock` method allow a VAS to use different locking schemes. Savepoints and transaction chaining are also available, but none of these more complicated locking and transaction features are demonstrated in this simple example.

For more details on transactions, locks, and concurrency arguments, see the manual pages `transaction(ssm), lock(ssm), enum(ssm)`.

4.2 Writing a Record: `write_hello()`

Each volume contains a *root index*, in which a server can store information for bootstrapping purposes (or any purpose, for that matter). The program uses this index to locate the record id of the record that was created so that it can be found later.

First the example function `write_hello` creates a file with `create_file`, which takes as arguments the local handle for the volume on which the file is to be created, and returns a file identifier for the new file.

```
rc_t
write_hello(const vid_t& handle, const vec_t& key, rid_t &rid)
{
    W_DO(ssm->begin_xct());

    cout << "Creating a file for holding the hello record ..." << endl;
    // create a file for holding the hello world record
    stid_t fid;
    W_DO(ssm->create_file(handle, fid, ss_m::t_regular));
```

```
    cout << " ... file " << fid << " created." << endl;

    ...
}
```

To create a record in the file, `write_hello` creates the data vector for the record. Each storage manager record has a header for (optional) use by the server. In our example, the header is not used, so its data vector is empty. The body vector contains the string "Hello". The program will first write the record "Hello", then append "World!" to the record.

```
rc_t
write_hello(const vid_t& handle, const vec_t& key, rid_t &rid)
{
    ...

    cout << "Creating the hello record ..." << endl;
    // just contains "Hello" in the body.
    const char* hello = "Hello";
    vec_t header_vec();
    vec_t body_vec(hello, strlen(hello));

    W_DO(ssm->create_rec(fid, header_vec, 0, body_vec, rid));
    cout << " ... record " << rid << " created."  << endl;

    // create a 2 part vector for "world" and "!" and append the
    // vector to the new record
    vec_t world(" World", 6);
    world.put("!", 1);
    cout << "Appending to record ..." << world << endl;
    W_DO(ssm->append_rec(rid, world));
    cout << " ... " << world << "appended " << endl;

    ...
}
```

For details about the vector class (used for headers, bodies, index keys, index values, etc), see the manual pages `vec_t(common)`.

For more details on files, scanning files, and pinning records, see the manual pages `file(ssm), scan_file_i(ssm), pin_i(ssm)`.

Once the record is created, the program stores its record identifier in the root index, so that the record can be found by an invocation of the program with the -r command-line option.

The index identifier of the root index is acquired by calling `vol_root_index`. Then the program creates key and value vectors for the root index entry. Finally, it creates the root index entry with `create_assoc`, and commits the transaction.

```
rc_t
write_hello(const vid_t& handle, const vec_t& key, rid_t &rid)
{
    ...

    // Stuff the record id into the root index
    stid_t rootIID;
    W_DO(ssm->vol_root_index(handle, rootIID));
    cout << "Volume root index is " << rootIID << endl;


    cout << "Creating index entry for for key " << key
        << " --> " << rid << endl;
    vec_t el(&rid, sizeof(rid_t));
    W_DO(ssm->create_assoc(rootIID, key, el));

    W_DO(ssm->commit_xct());
    return RCOK;
}
```

For more details on indexes, see the manual pages `btree(ssm)`, `scan_index_i(ssm)`.

4.3 Locating a Record: `find_hello()`

If the program is run with the -r command-line option, it looks for the record with the well-known key. If such a record is not found, it writes the record as if called without the command-line option.

To look up the record identifier associated with the well-known key, the server calls `find_assoc`, giving the index identifier (also known as a s*tore* identifier, `stid_t`, of the root index, the key, and a vector describing the memory into which the storage manager will write the value associated with the key, if found. The vector points to a record identifier instance and is sized for the record identifier instance.

```
rc_t
find_hello(const vid_t& handle, const vec_t &key, rid_t &rid)
{
    W_DO(ssm->begin_xct());

    // Find the record id from the root index
    stid_t rootIID;
    W_DO(ssm->vol_root_index(handle, rootIID));

    bool found(false);
    smsize_t elen(sizeof(rid_t));
```

```
    cout << "Finding record id for key " << key << endl;
    W_DO(ssm->find_assoc(rootIID, key, &rid, elen, found));
    if(found) {
        cout << " ... rid for key " << key << " is " << rid << endl;
    } else {
        W_COERCE(write_hello(handle, key, rid));
    }

    W_DO(ssm->commit_xct());
    return RCOK;
}
```

For more details on indexes, see the manual pages `btree(ssm)`, `scan_index_i(ssm)`.

4.4 Reading a Record: `say_hello()`

Once the record identifier is known, the program can print the contents of the record.

```
rc_t
say_hello(const rid_t& rid)
{
    W_DO(ssm->begin_xct());

    //  pin the vector so we can print it
    // open new scope so that pin_i destructor is called before commit
    {
        cout << "Pinning the record for printing ..." << endl;
        pin_i handle;

        // pin record starting at byte 0
        W_DO(handle.pin(rid, // record
                            0    // starting byte
                        //, lock mode = SH
                        ));

        cout << "Printing record  ..." << endl;
        // iterate over the bytes in the record body
        // to print "hello world!"
        cout << endl;
        for (smsize_t i = 0; i < handle.length(); i++) {
            cout << handle.body()[i];
        }
        cout << endl;
    }
```

```
    W_DO(ssm->commit_xct());

    return RCOK;
}
```

Reading records is accomplished by using the pin iterator `pin_i` class to pin the record in the buffer pool. A `pin_i` object is often called a "handle" to a record. There are a few things to keep in mind when using the `pin_i` interface.

The `body` method returns a `const` pointer to the record body. *Never modify the record through these pointers, otherwise roll back and recovery information will not be generated.*

The most efficient way to modify a pinned record is to use the `update_rec`, `append_rec` and `truncate_rec` methods of class `pin_i`. A record can also be updated using the `update_rec`, `append_rec` and `truncate_rec` methods of class `ss_m`, however, if the record is already pinned, the pin object will not be changed to reflect the update and `pin_i::repin` must be called to update the pin iterator.

For more details on files, scanning files, and pinning records, see the manual pages `file(ssm), scan_file_i(ssm), pin_i(ssm)`.

5 Building and Running the Example

The example program is among the 'make check' targets in the distribution, so it builds by running 'make check' in the hello/ directory after the distribution is configured and made. 'Make check' also runs the program through the script 'hello.sh'. This automagically uses the same configuration that was used when the storage manager was built:

```
# in UNINSTALLED SSM source tree:
make check
```

If you are using an *installed* storage manager (one for which 'make install' was run), the header files, libraries, and configuration file have to refer to the proper install directories. The 'make install' handles this and creates a make file called `hello.make`, which is put in the installed hello/ directory. To use this file, run

```
# in INSTALLED DOC hello/ directory:
make -f hello.make hello
```

NOTE that the server *must* be built with the same compiler flags (macros, etc) that were used when the installed storage manager was built. To this end, 'make install' puts these flags into a file called `makeflags` in the `<installdir>/include` directory.

```
# in INSTALLED include/ directory:
# Use these flags for any server code compiles>
cat makeflags
```

5.1 Running the Example

The configuration options for the server and client can be set in the file `exampleconfig` or on the command line. You must edit `exampleconfig` to set the `sm_diskrw` option to the location of the installed `diskrw` program.

The first thing to do is to run `hello` to format and make the volumes containing the data. Next, type `hello -r` to read back what was written.

```
# in UN-INSTALLED SSM source tree:
make check
# or
hello.sh


# in INSTALLED DOC hello/ directory:
hello.sh
```