Writing Value-Added Servers with the Shore Storage Manager[1]

The Shore Project Group
Computer Sciences Department
UW-Madison
Madison, WI
Version 1.1.1
Copyright ©1994–7
Computer Sciences Department, University of Wisconsin—Madison.
All Rights Reserved.

October 27, 1997

[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.

Contents

1 Introduction **1**
- 1.1 Goals . . . 1
  - 1.1.1 What this Tutorial Is . . . 1
  - 1.1.2 What this Tutorial Is Not . . . 1
- 1.2 The Example System . . . 1
  - 1.2.1 What the Grid VAS and Grid Client Do . . . 2
  - 1.2.2 What the Grid Example Demonstrates . . . 2
  - 1.2.3 What the Grid Example Does Not Demonstrate . . . 2
- 1.3 Tutorial Organization . . . 3

2 Storage Structure Organization **3**
- 2.1 Implemented Design . . . 3
- 2.2 Alternative Designs . . . 4

3 Operations on Storage Structures **4**
- 3.1 Code Road-map . . . 4
- 3.2 Initializing Devices and Volumes . . . 4
- 3.3 Updates and Save-points . . . 5
- 3.4 Pinning and Updating Records . . . 5

4 Implementing a Multi-Threaded Server **5**
- 4.1 Error Codes . . . 6
- 4.2 Startup . . . 6
  - 4.2.1 Configuration Options . . . 6
  - 4.2.2 SSM Initialization . . . 7
- 4.3 Thread Management . . . 7
  - 4.3.1 Listener Thread . . . 7
  - 4.3.2 Client Threads . . . 8
  - 4.3.3 Cleaner Thread . . . 8
  - 4.3.4 Terminal Input Thread . . . 8
- 4.4 Transaction and Lock Management . . . 8
- 4.5 RPC Implementation . . . 9
  - 4.5.1 Declarations . . . 9
  - 4.5.2 C++ Wrappers . . . 9
  - 4.5.3 RPC Startup . . . 10
  - 4.5.4 Multi-threading Issues . . . 10
  - 4.5.5 Steps to add a New RPC . . . 10
- 4.6 Shutdown . . . 11

5 Implementing Clients **11**
- 5.1 Connection Management . . . 12
- 5.2 Client Side of RPCs . . . 12

6 Compiling the Example **12**

7 Running the Example **13**

8 Appendix: Program Sources **15**

1 Introduction

This tutorial explains, through the use of an example client-server system, how to write a value-added server (VAS) and client programs using the Shore Storage Manager (SSM) programming interface.

1.1 Goals

1.1.1 What this Tutorial Is

This tutorial illustrates many aspects of writing a VAS, including

- using the storage facilities,
- managing transactions and recovery,
- using the multi-threading and synchronization facilities,
- writing a server to support multiple clients using Sun RPC,
- configuring the SSM using the options package,
- handling errors, and
- compiling and running programs.

1.1.2 What this Tutorial Is Not

This tutorial is not intended to be

- A general introduction to the SSM, its goals, structure or status.

  For a general introduction, see An Overview of Shore or The Shore Storage Manager Programming Interface. See The Shore Release for an index to the rest of the documentation.

- A demonstration of all the features of the SSM.
- A general guide to using Sun RPC.

  We assume you are already familiar with RPC programming. If not we recommend the book *Power Programming with RPC*, by John Bloomer, published by O'Reilly & Associates, Inc.

- A tutorial on multi-threaded programming in general.

1.2 The Example System

The example used throughout this tutorial is a client-server system called *grid* that illustrates many aspects of building a value-added server and a corresponding client.

1.2.1 What the Grid VAS and Grid Client Do

The grid example is a simple client-server database system for *items* located on a 2-dimensional grid. Items have a string name and coordinates on the grid. Multiple items can reside at any location, but each item has a unique name.

The server uses the SSM to manage the grid database. The server implements several update and query commands. Update commands on the grid include adding items, removing items and clearing the grid (removing all items). Query commands on the grid including printing the entire grid, looking up items by name, and locating all items within a rectangular region of the grid. There are also commands for committing and aborting transactions. The server accepts commands from standard input and processes remote procedure calls (RPCs) from multiple clients.

The client accepts commands from standard input and sends them to the server with remote procedure calls (RPCs).

1.2.2 What the Grid Example Demonstrates

- Storage facilities

  The grid data file and indices are stored on a single volume. The items are stored, one per record, in an SSM file. Each record contains a name and a location. A B+-tree index maps item names to the IDs of the corresponding item records. This index is used to lookup items by name. An R*-tree index maps location coordinates to item records. This index is used to locate all items within a rectangular region of the grid.

- Threads

  The grid server is a multi-threaded threaded program that manages connections from multiple clients and also monitors the terminal (standard input) for commands. The server starts a separate thread for each of these tasks to demonstrate how to use SSM's thread package. Shared state among the threads is protected using the synchronization mechanisms provided by the thread package.

- Configuration options

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

1.2.3 What the Grid Example Does Not Demonstrate

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

- appending/truncating records,

- pinning large records (longer than 8K bytes),
- gathering statistics,
- bulk-loading an index,
- sorting, or
- using sophisticated logical ID features such as inter-volume references

Most of the unused features are used in the Shore VAS (see `src/vas`) and the Paradise database system (see http://www.cs.wisc.edu/paradise/). Almost every feature is also tested by the SSM testing program `ssh`, located in `src/sm/ssh`.

1.3 Tutorial Organization

This tutorial walks through the example program sources in detail. The sources, as well as associated test programs and data, may be found in the `examples/vas/grid` sub-directory of the distribution.

The rest of this tutorial is organized as follows. The following two sections present the storage structures and operations on them. The next two sections discuss how to implement servers and clients. The final two sections explain how to compile and run the example.

2 Storage Structure Organization

One of the first decisions in designing a server is what persistent storage structures to use. This section discusses how we organized storage structures for the grid example. There are numerous ways to organize storage. The primary decision is where to store the information about each item on the grid. We first discuss the implemented design and then present a couple of alternatives.

2.1 Implemented Design

We chose a relational database approach to organizing the data. Recall from the introduction that each item has a short string name and a coordinates on the grid. Each item is stored in a record. All item records are stored in a single file. Items can be retrieved using their record ID or by scanning the file.

To improve performance of lookups, we use two indices. To support name lookups, we use a B+-tree index mapping item names to record IDs. To support lookups on location, we use an R*-tree mapping an item's coordinates to its record ID.

All files and indices must be located on a volume. For the grid example, only one volume is used. This volume is located on a device whose name is specified with a configuration option. When the server starts, it must locate the file and indices for the grid, so we store the IDs of the file and indices in the root index of the volume.

This design has three desirable properties:

- Storage requirements are based on the number of items in the grid, not the size of the grid.

- Since each item is stored as a separate record we can take advantage of the fine-granularity (record-level) locking provided by the SSM.
- Adding new indices is easy.

2.2 Alternative Designs

One alternative design is to store items directly in a B+-tree index, with the item name as the key and coordinates as the data associated with it.

This design has some problems. First, unlike records, entries in a B+-tree index do not have IDs, so the the R*-tree index would then map to names, making spatial lookups awkward. Second, if items are enlarged to hold more data, they may no longer fit within the size limitation (1-page) of index entries. Third, indices do not support updating of individual entries, so changing the location of an item would require removing it from the index and reinserting it.

Another alternative design is to treat the grid as a 2-D array of items and store the entire grid in one large record. This representation would be efficient for densely populated grids, but is wasteful for sparse ones. It would also not support multiple items per location. Worse yet, the granularity of locking would be the entire grid.

3 Operations on Storage Structures

After deciding on the basic storage structures, we can begin implementing operations on them. This section starts by providing road-map to example code for specific types of operations. The code should be self explanatory. Later sub-sections discuss some of the less obvious aspects of the implementation.

3.1 Code Road-map

- Volume and Device initialization: `setup_device_and_volume()` in `server.C`
- Creating Files and Indexes: `command_server_t::init()` in `command_server.C`
- Creating Records and Index Entries: `grid_t::add_item()` in `grid.C`
- Destroying Records and Index Entries: `grid_t::remove_item()` in `grid.C`
- Pinning and Updating Records: `grid_t::move_item()` in `grid.C`
- Scanning Files: `grid_t::generate_display()` in `grid.C`
- Index Lookups: `grid_t::locate_item()` in `grid.C`
- Scanning Indexes: `grid_t::spatial_query()` in `grid.C`

3.2 Initializing Devices and Volumes

Before creating a storage volume, the "device" (raw device or Unix file) where it will reside must be initialized. A flag to `setup_device_and_volume()` indicates whether the device is to be initialized (i.e., we are starting from scratch) or if there is already an initialized device.

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` and create a volume on it with `ss_m::create_vol`. Since we will be using the SSM's logical ID support the `ss_m::add_logical_id_index` method must be called to create the index that maps from logical ID to physical IDs. The reason the logical ID index is not added automatically by `ss_m::create_vol` is an artifact of the fact that the SSM also has an interface based on physical IDs (the physical ID interface it not supported yet).

If no initialization is needed, we mount the device and use `ss_m::list_volumes` to find the ID of the volume on the device (currently Shore supports only one volume per device, but the interface is designed to allow this rule to be relaxed in the future).

The code in `command_server_t::init()` uses the root index of the volume to determine if the grid storage structures have already been created. If not it creates them.

3.3 Updates and Save-points

All the grid interface commands that update data involve changing more than one storage structure. For example, the *add* command creates an item and makes corresponding entries in the name and location indices. If any of these operations should fail, it is necessary to roll back to the point just before the first operation to keep the database consistent. Therefore, the first thing `grid_t::add_item` does is create a save-point using `ss_m::save_work`. If any of the later create operations reports an error, `ss_m::rollback_work` is called.

The use of save-points around all operations that make changes to storage structures is highly recommended. That way even if an `ss_m` update method should fail, any partial work it completed is rolled back. The only other safe choice is to abort the entire transaction.

3.4 Pinning and Updating Records

Reading records is accomplished by using the `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 couple of things here 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. If you do, 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` rather than those in `ss_m` . If the append or truncate methods cause the record to be moved, the pin object will continue to point to it.

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` would need to be called.

4 Implementing a Multi-Threaded Server

The capability to implement a multi-threaded server that manages 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 allow writing clients that run only one transaction at a time and are usually single-threaded.

The grid example server is a multi-threaded threaded program that manages requests from multiple clients, and interactive commands through its terminal interface.

4.1 Error Codes

Most SSM methods return an error code object of type `w_rc_t` (usually typedef'ed as `rc_t`. It is important always to 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. More details on error checking are available in the SSM interface document.

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.

Many of the grid methods return `w_rc_t` codes as well. However, the RPC-related methods of ĉommand_server_t return error message strings. The conversion from `w_rc_t` to string is done by `SSMDO` macro found at the top of ĉommand_server.C.

4.2 Startup

4.2.1 Configuration Options

Several SSM configuration options must be set before the SSM is started with the `ss_m` constructor. In addition, most servers, including the grid 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 `server.C`, `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`. Once the option group is complete we call `init_config_options` in `options.C` to initialized the options' values.

The `init_config_options` function is used by both the client and server programs to initialize option values. The first thing it 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 two, in this case `grid`. The second is the type of program, in this case `server`. The third is the filename of the program executable, which is also `server`. The classification levels allow options to be set for multiple programs with a single configuration file. For example,

both the client and server programs have a `connect_port` option for specifying the port clients use to connect to the server. The following line in a configuration file sets the connection port so that client and server always agree:

```
grid.*.connect_port:  1234
```

The following line would be ignored by the grid programs as it is for the Shore VAS system:

```
shore.*.connect_port:  1234
```

After setting the level names, `init_config_options` reads the configuration file `./exampleconfig` scanning 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`.

4.2.2 SSM Initialization

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 (as is done in `main` in `server.C`).

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.

Once the SSM is constructed, `main` calls `setup_device_and_volume` to initialize the device and volume as described htmlrefabovessmvas:initializing. With the SSM constructed, we can now start the threads that do the real work.

4.3 Thread Management

The grid server manages multiple activities. It responds to input from the terminal, listens for new connections from clients, and processes 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 no longer blocks, only threads do.

The subsections below explain the three types of threads used by the grid server. The thread classes are declared in declared in `rpc_thread.h` and implemented in `rpc_thread.C`. Notice that each thread class is derived from `smthread_t`. All threads that use SSM facilities must derived from `smthread_t` rather than the base class, `sthread_t`.

The first code to be executed by any newly forked thread is its `run` method. The `run` method is virtual so that it can be specialized for each type of thread.

4.3.1 Listener Thread

Once the RPC facility has been initialized, `main` creates a new thread, type `listener_t`, that listens for client connections. The listener thread does two jobs:

- Wait for client connection requests and fork a thread to handle the connection
- Manage an orderly shutdown of clients when the `shutdown` method is called

The work of the listener thread is all done in its `run` method (as is true for all most threads). The first thing `run` does is create a file handler (`sfile_read_hdl_t`) for reading from the connection socket. The code then loops waiting for input to the socket. When a connection

request arrives, the RPC function `svc_getreqset` is called allowing the RPC package to process the connection request. Then, a `client_t` thread (discussed in the next section) is created to handle the connection. The new client thread is added to the listener's list of clients. Notice that since the client list may be accessed by multiple threads, it is protected by a *mutex*.

When the server is ready to shutdown, `main` calls `listener_t::shutdown`, which in turn calls shutdown on the file handler for the connection socket. This causes the listener thread to wakeup and break out of the while loop in the `run` method. The listener then notifies the cleaner thread (see below) to destroy defunct threads.

4.3.2 Client Threads

When the listener thread detects a new connection it forks a new thread, type `client_t`, to process RPC requests on the connection. The `client_t` constructor is given a socket on which to wait for requests and a pointer to the listener thread to notify when it is finished. Notice that the client thread has a buffer area for generating RPC replies, called `reply_buf`.

The `client_t::run` method begins by creating a a file handler (`sfile_read_hdl_t`) for reading from the socket where requests will arrive. Next, a `command_server_t` object is created to process the requests.

The code then loops waiting for input on the socket. When an RPC request arrives the RPC function `svc_getreqset` is called which in turn dispatches the RPC to the proper RPC stub function (implemented in `command_server.C`). When the connection is broken, the loop is exited and the file handler and `command_server` are destroyed. Then `listener_t::child_is_done` is called to notify the listener that the client thread is finally finished.

4.3.3 Cleaner Thread

The cleaner thread waits on a condition variable and when awoken, checks for defunct threads in the list of client threads. Any defunct threads found are removed and destroyed.

Normally it wakes up when a client thread finishes its `client_t::run` method, checks the list, and the waits again on the condition variable `cleanup`. When the listener thread ends, it causes the cleaner thread to destroy itself.

4.3.4 Terminal Input Thread

The main program simply starts a main thread after processing options. The `main` thread then takes over the work of the server. After starting the listener thread, `main` creates another thread, of type `stdin_thread_t`, which processes commands from standard input.

The work of the standard input thread is all done in its `run` method (as is true for all most threads). The first thing `run` does is create a file handler (`sfile_read_hdl_t`) for reading from the file descriptor for standard input. Next, a `command_server_t` object is created to process the commands. The code then loops waiting for input. When input is ready, a line is read and fed to `command_server_t::parse_command` for processing. If `parse_command` indicates that the quit command has been entered or if EOF is reached on standard input, the input loop is exited and the thread ends.

4.4 Transaction and Lock Management

The grid server uses a simple transaction management scheme. All operations on data managed by the SSM must be done within the scope of a transaction. Each client thread starts a transaction for the client it manages. Clients decide when to end (either commit or abort) the transaction. When this occurs a new one is automatically started by the grid server. If a client disconnects from the server, its current transaction is automatically aborted.

The SSM automatically acquires locks when data is accessed, providing serializable transactions. The grid server relies on the automatic locking done by the SSM. One example of where the server explicitly acquires locks is in the `grid_t::clear` method, which removes every item from the database. Here we acquire an EX lock on the item file and indices to avoid the overhead of acquiring finer granularity locks.

More sophisticated transaction and locking schemes are possible. For example, the `grid_t::generate_display` method (used by the print command) locks the entire file containing items, thus preventing changes to the grid. For greater concurrency, it is possible to have `generate_display` start a separate transaction before scanning the item file. Afterward, it can commit the transaction, releasing the locks on the file. To do this the client uses the `smthread_t::attach` method to attach to the original client transaction.

Another way to get a similar effect is to use the `t_cc_none` flag to the concurrency control (`cc`) parameter of the `scan_file_i` constructor.

4.5 RPC Implementation

At the heart of the grid system are the RPCs called by the client and serviced at the server. We use the publicly available Sun RPC package to implement the RPCs.

4.5.1 Declarations

The RPCs are declared in `msg.x`. This includes `grid_basics.h` which contains some additional declarations used throughout the grid code. The first part of `msg.x` contains declarations for structures use to hold RPC arguments and return values, followed by a listing of the RPCs. The final part of the file contains ANSI-C style function prototypes for the server and client side RPC stubs since the RPC package does not generate them.

The `msg.x` file is processed by the `rpcgen` (see rpcgen(1) manual page) utility to create the following files:

- `msg_clnt.c`: client-side stubs for the RPCs
- `msg_svc.c`: server-side dispatch routine
- `msg.h`: declarations used by both the client and server
- `msg_xdr.c`: xdr functions

4.5.2 C++ Wrappers

The output of `rpcgen` is inconvenient for two reasons: it is is C not C++ and the client stubs take different parameters than those of the server. Therefore, we encapsulate the RPCs in

the abstract base class `command_base_t` declared in `command.h`. The pure virtual functions in this class represent RPCs. Class `command_client_t` (in `command_client.h`) is derived from `command_base_t` and implements the client side of the RPCs by calling the C routines in `msg_clnt.c`. Also derived from `command_base_t` is `command_server_t` (in `command_server.h`) that implements the server side of the RPCs.

The server-side C stubs for the RPCs, implemented in `server_stubs.C`, call corresponding `command_server_t` methods.

The only function that makes RPC requests is `command_base_t::parse_command`. It parses a command line and calls the appropriate `command_base_t` method implementing the RPC.

To process RPC requests on the server, an instance of `command_server_t` is created for each client thread. When an RPC arrives, the thread managing the client is awakened and the RPC dispatch function in `msg_svc.c` is called. This calls the server-side C stub which in turn calls the corresponding `command_server_t` method. The methods in `command_server_t` call `grid_t` methods (in `grid.C`) to access and update the grid database.

To execute commands on the server, an instance of `command_server_t` is created for the thread managing standard input. This thread calls `command_base_t::parse_command` for each line of input. The parse `parse_command` method calls the `command_server_t` methods, short-circuiting the RPC facility.

4.5.3 RPC Startup

Once the SSM and volumes are initialized, the grid server is ready to start the RPC service and begin listening for connections from clients. RPC start-up is done by the function `start_tcp_rpc` in `server.C`. This function creates the socket used to listen for connection requests, binds a port to the socket, and then calls RPC facility's initialization functions.

4.5.4 Multi-threading Issues

The multi-threaded environment of the server requires changes to a couple common practices in Sun RPC.

Replies are usually placed in a statically allocated structure. With multiple threads, each threads needs its own space for replies, so a reply area is created for each thread as described in Section4.3.2.

The RPC package allocates (`mallocs`) space for pointer arguments in RPCs. The convention is that the function processing a request frees the space from the previous request of the same type. Because the convention requires that the reply be saved in static storage, this does not work in a multi-threaded environment. The Sun RPC package shipped with the Shore release has modified `rpcgen` to generate a dispatch routine that automatically frees the space after the reply is sent, relieving the function of the burden of freeing the space. Because of this change, the library does not lend itself to saving replies for the purpose of retransmitting them in response to duplicate requests (for the UDP service).

4.5.5 Steps to add a New RPC

As an example. of how to add an RPC we explain how the `locate` command was added to the grid example.

- Add argument and reply types to `msg.x`.

  The argument type for the locate command is `location_arg` and the reply type is `location_reply`. The constant, `thread_reply_buf_size` must also be changed to reflect the size of the `location_reply` structure.

- Add RPC and stub declarations to `msg.x`.

  The RPC is called `location_of_rpc` and is listed in the `program` section of `msg.x`. Below this are declarations for the client and server-side stubs.

- Add declaration to `command.h`, `command_server.h` and `command_client.h`.

  Recall that there are C++ wrapper methods for the RPCs in the abstract base class `command_base.h` and its derived classes. So, declarations for the RPC wrapper method must be added to them. We used the method name `location_of`.

- Implement wrapper method in `command_client.C`.

  The client side of `location_of` must call the RPC stub, `location_of_rpc_1`.

- Implement server stub to call the wrapper method in `server_stubs.C`.

  The server-side stub for the RPC, `location_of_rpc_1`, must call the wrapper method, `command_server_t::location_of`.

- Implement wrapper method in `command_server.C`.

  The wrapper method implements the RPC by called a corresponding method of the `grid_t` class. It converts any error into a string to be sent in the reply.

- Implement the `location_of` method in `grid.C`.

  All access to the grid database is done by methods of `grid_t`. Therefore there is a `location_of` method that does the actual index lookup to find the location of an item. Of course, a declaration for `location_of` must be added to `grid.h`.

- Add locate command to parser in `command.C`.

  Implemented in `command.C` is `command_base_t::parse_command`, which parses a command line and calls the the RPC's C++ wrapper. To add the command, edit the `enum` for command tokens and the array of command descriptions. Then add the command to the `switch` statement in `parse_command`.

4.6 Shutdown

Shutting down the SSM involves ending all threads, except the one running `main` and then destroying the `ss_m` object. After `main` starts the thread for commands on the terminal, main calls `wait` on the thread. When the `quit` command is entered, the terminal thread ends causing `wait` to return, thus waking up the main thread. `Main` then tells the listener listener thread to shutdown and does a `wait` for it. The shutdown process for the listener thread is described in Section4.3.1. The main thread wakes up when the listener thread is done. The final shutdown step is to `delete` the instance of the class `ss_m` created at the beginning of `main`.

5 Implementing Clients

The client is simpler than the server. It is single threaded and calls server RPCs to do most work. It uses the the configuration options and error handling facilities of the SSM similar to the server use of them.

This section focuses on RPC issues.

5.1 Connection Management

The code for connection management is all in file `client.C`. The function `connect_to_server` connects to a server at a specific host and port. The original Sun RPC package has no direct support for this since it assumes you will use the `port mapper` facility. However, the Shore version of the package has a `clnt_create_port` function that could replace most of the code in `connect_to_server`.

The function `disconnect_from_server` demonstrates how to end a session with the server.

5.2 Client Side of RPCs

The client-side stubs for the RPCs are generated by `rpcgen` and placed in `msg_clnt.c`. Recall that we wrap the RPCs in C++ methods from the `command_base_t` abstract base class. From that class we derive `command_client_t` and implement the methods by calling the RPC stubs.

In `client.C` the function `process_user_commands` creates an `command_client_t` object, reads lines from standard input, and calls `command_client_t::parse_command` for each line. Method `parse_command` in turn calls the `command_client_t` wrapper methods.

6 Compiling the Example

The Shore documentation release contains the source code for the examples in this tutorial. Assuming you have fetched and unpacked the documentation release as described in the *Shore Software Installation Manual*, you will have a directory `$SHROOT/examples`, where `$SHROOT` is the root directory of the documentation relase.

You must set the environment variable `$SHORE` to point to the location where you have installed the Shore binaries and libraries. If you are using a binary distribution, that would be the same as `$SHROOT`. If you built the binaries from sources, it would be `$SHROOT/installed`.

```
mkdir grid
cp -R $SHROOT/examples/vas/grid/* grid
cd grid
make all
```

You should see something like this:

```
cp /usr/local/shore/include/ShoreConfig.h .
/usr/local/shore/bin/rpcgen msg.x
rm -f msg_svc.c
/usr/local/shore/bin/rpcgen -m -o msg_svc.c msg.x
```

```
g++ -g -I/usr/local/shore/include   -c grid.C
g++ -g -I/usr/local/shore/include   -c rpc_thread.C
g++ -g -I/usr/local/shore/include   -c server.C
g++ -g -I/usr/local/shore/include   -c command_server.C
g++ -g -I/usr/local/shore/include   -c server_stubs.C
g++ -g -I/usr/local/shore/include   -c options.C
g++ -g -I/usr/local/shore/include   -c command.C
g++ -g -I/usr/local/shore/include   -c grid_basics.C
gcc -g -traditional -I/usr/local/shore/include   -c msg_xdr.c
gcc -g -traditional -I/usr/local/shore/include   -c msg_svc.c
g++ -g -I/usr/local/shore/include    -o server msg_svc.o grid.o \
    rpc_thread.o server.o command_server.o server_stubs.o options.o \
    command.o grid_basics.o msg_xdr.o /usr/local/shore/lib/libsm.a \
    /usr/local/shore/lib/libshorecommon.a -lnsl
g++ -g -I/usr/local/shore/include   -c client.C
g++ -g -I/usr/local/shore/include   -c command_client.C
gcc -g -traditional -I/usr/local/shore/include   -c msg_clnt.c
g++ -g -I/usr/local/shore/include    -o client client.o \
    command_client.o options.o command.o grid_basics.o msg_clnt.o \
    msg_xdr.o /usr/local/shore/lib/libshorecommon.a -lnsl
sed -e "s,DISKRW,/usr/local/shore/bin/diskrw," exampleconfig > config
mkdir log.grid
```

Notice that the modified `rpcgen` shipped with Shore is used by default. The make target "all" also customizes the `config` file to indicate the location of the `diskrw` program and makes a directory `log.grid` to hold the log file for running the grid VAS server (the name of this directory is set by the `wm_logdir` option in `config`). When the compilation is complete, you should have a two executables, `server` and `client`.

7 Running the Example

The first thing to do is to run `server -i` to format and make the volumes containing the grid data. Before running the grid server, you must edit `exampleconfig` and change the value of the `sm_diskrw` option to `$SHORE/bin/diskrw`. (Note: You must replace `$SHORE/bin/diskrw` with the actual pathname of the `diskrw` executable; you cannot use a "shell variable" in this file.) Information about all the options available to the programs can be found by running them with a `-h` flag.

Running `grid` will leave log files in `./log.grid` and a storage device file called `./device.grid`. These can be removed when you are done.

```
% server -i
```

Answer "y" to the question "Do you really want to initialize the Grid database?". You should see something like this:

```
processing configuration options ...
```

```
Do you really want to initialize the Grid database? y
Starting SSM and performing recovery ...
Formatting and mounting device: device.grid with a 2000KB quota ...
Creating a new volume on the device
    with a 2000KB quota ...
starting up, listening on port 1234
allocating a tcp socket for listening for connections ...
binding to port 1234
creating tcp service
registering rpc service
main starting stdin thread
creating file handler for listener socket
Command thread is running
Creating a new Grid
Server>
```

With the server running in one window, type `client` in another window to start a client. At either the client or server prompt, you can type `help` to get this list of available commands.

```
% client
processing configuration options ...
trying to connect to server at port 1234
attempting server connection
Client ready.
client> help
Valid commands are:

    commit
        commit transaction and start another one
    abort
        abort transaction and start another one
    clear
        clear grid
    print
        print grid
    add name x y
        add new item <name> at <x,y>
    remove name
        remove item <name>
    move name x y
        move item <name> to location <x,y>
    locate name
        print location of item <name>
    spatial x_lo y_lo x_hi y_hi
        print count of items in rectangle and list first few items
    quit
        quit and exit program (aborts current transaction)
```

```
    help
        prints this message

    Comments begin with a '#' and continue until the end of the line.
client>
```

At all times a transaction is running for the server and client prompts. As commands are run, appropriate locks are obtained. Locks will be released when a commit or abort command is given.

Here are some commands you might try typing to the client.

```
print                   # print the empty grid
add Junk 20 10          # add and item named Junk at coordinate 20,10
add Car 15 5
print
commit                  # commit the current transaction
locate Car              # find location of Car
spatial 0 0 39 14       # count all items on grid
spatial 20 10 20 10     # list all items at location 20,10
clear                   # clear the grid
print                   # it should be empty
abort                   # abort the current transaction
print                   # grid should have items on it now
quit                    # quit -- aborts current transaction
```

When you are done, you can type "`make clean`" to remove the compiled programs, or "`make distclean`" to remove the compiled programs as well as the "database" and log files generated by running them.

8 Appendix: Program Sources

Program sources are not included in the Postscript version due to their length. The sources can be found in `SHROOT/examples/vas/grid`.