Scanning Pools in a Shore Application[1]

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

Version 1.1.1

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

The *PoolScan* class provides a convenient interface for scanning through the objects in a pool in Shore VAS clients. This document describes how to use PoolScans, as well as known bugs and limitations of the PoolScan class. The Shore release includes three example programs that illustrate pool scans. The examples can be found in the `examples/pscan` directory of the Shore release. The PoolScan class is part of the SDL library, but it is not necessary to use SDL or the SDL compiler to use it.

2 The PoolScan Class

An application scans a pool by creating a PoolScan object. PoolScan objects are transient C++ objects, not persistent Shore objects. A PoolScan object can be in one of two states: *open* or *closed*. A scan must be opened inside a transaction, and can only be used inside the transaction in which it was opened. Once a scan is opened, it stays open until it is explicitly closed, or until the scan object is destroyed, or the transaction terminates. In particular, the scan object remains open even after the end of the scan has been reached or an error has occurred.

The public interface to the PoolScan class is

```
class PoolScan {
 public:
    PoolScan();
    PoolScan(const char *path);
    PoolScan(const ref<Pool> pool);

    shrc open(const char *path);
    shrc open(const ref<Pool> pool);

    shrc next(ref<any> &ref, bool fetch = false, LockMode mode = SH);

    bool is_open();
    shrc rc();
    int operator==(shrc rc);
    int operator!=(shrc rc);
```

```
        shrc close();
        ~PoolScan();
    };
```

`PoolScan()` is the default constructor. The resulting PoolScan object is in the *closed* state. It may be opened by the `open` member function.

`PoolScan(const char *path)`

`PoolScan(const ref<Pool> pool)` These constructors invoke the corresponding versions of the `open` member function. The caller should whether the `open` operation was successful by calling `is_open()` or by testing the return code with `rc()`, `operator==()`, or `operator!=()`.

`shrc open(const char *pool)`

`shrc open(const ref<Pool> pool)` The `pool` argument should be the pathname of a valid directory in the Shore filesystem namespace or a reference to a Pool object, and the PoolScan object should be in the *closed* state. The caller must have (at least) read permission for the indicated pool and must have a transaction open. Upon successful completion, `RCOK` is returned. Any other return value indicates an error condition.

`shrc next(ref<any> &ref, bool fetch = false, LockMode mode = SH)` returns a reference to the next object in the pool (if the scan has just been opened, then it returns the first object in the pool). Objects are returned in apparently random order, but each object is returned exactly once by a scan. If `fetch` is `true`, the object will be fetched into the object cache, and a lock will be obtained on the object in the mode indicated by `mode`, if such a lock is not already held by the transaction. If `fetch` is `false` (the default), `mode` is ignored and the object is not fetched from the server.

The return value of `next` is `RCOK` if there was a next object in the scan. If an attempt is made to go beyond the end of the scan (i.e., a call to `next` is made after the last object in the scan has already been returned), then the return code will be `OC_EndOfScan`. Any other return code indicates an error condition.

`bool is_open()` indicates whether the scan is open or closed. Note that the scan remains open after retrieving the last entry.

`shrc rc()` returns the return code generated by the last operation (`open` or `next`).

`int operator==(shrc rc)`

`int operator!=(shrc rc)` These operators provide a convenient way to check the status of the scan. For example, applications can say `if (scan == RCOK) ...`.

`shrc close()` closes the scan object. Closing a scan object releases any resources associated with the scan in both the server and client processes (such as the scan buffer). It also allows the scan object to be reused. The scan is closed implicitly when the PoolScan object is destroyed (either via operator delete or when the DirScan object goes out of scope), or when the transaction in which the scan was opened terminates.

`~PoolScan()` The PoolScan destructor closes the scan object if it is open.

3 A Pool Scan Example

The following function counts the total number of objects in the indicated pool.

```
int count(const char *pathname)
{
    ref<any> ref;
    int count;

    // Open a scan over the pool given by "pathname"
    PoolScan scan(pathname);

    // Make sure the scan was successfully opened.
    if(scan != RCOK){
        cout << "Error scanning pool " << pathname << ": "
             << rc << endl;
        return 0;
    }

    // Scan until end-of-scan or an error is encountered.
    for(count = 0; scan.next(ref) == RCOK; ++count);

    // Check for errors
    if(scan.rc().err_num() != OC_EndOfScan){
        cout << "Error scanning pool " << pathname << ": "
             << rc << endl;
        return 0;
    }

    cout << path << " has " << count << " objects." << endl;

    // The destructor will close the scan object.

    return count;
}
```

A complete program using code similar to this may be found in the directory `examples/pscan` in the documentation distribution. See the instructions for *Compiling and Running an Application* in the *Shore Software Installation Manual*.

4 Bugs and Limitations

In the current release of Shore, an application can have only one scan open at a time, although any number of scans can be open over the course of a single transaction.

In the current release of Shore, an application scanning a pool will not see any objects that it has created inside the pool during the current transaction. Therefore, new-object creation and pool scans should be performed in separate transactions.