NAME

transactions – methods and macros for transactions

SYNOPSIS

```
#include <ShoreApp.h>

shrc     Shore::begin_transaction(int degree = 2);

shrc     Shore::commit_transaction(bool invalidate = false);

shrc     Shore::chain_transaction();

shrc     Shore::abort_transaction(bool invalidate = false);

TxStatus Shore::get_txstatus();

         SH_BEGIN_TRANSACTION(rc);

shrc     SH_COMMIT_TRANSACTION;

shrc     SH_COMMIT_TRANSACTION_INV;

         SH_CHAIN_TRANSACTION;

         SH_ABORT_TRANSACTION(shrc rc);

         SH_ABORT_TRANSACTION_INV(shrc rc);


         SH_DO(shrc);
```

DESCRIPTION

Begin_transaction begins a new transaction. Currently, application programs can have only a single active transaction at a time. An error will be signalled if a transaction is already running when this method is called.

Commit_transaction flushes the object cache, writing any new or modified objects to the Shore server, and commits the current transaction, releasing all locks acquired while the transaction was running. If a critical error occurs during a transaction, all subsequent operations will return an error indicating that the transaction must be aborted, and it will not be possible to begin a new transaction until the current transaction has been aborted.

Chain_transaction commits the transaction and starts a new transaction. It writes any new or modified objects to the Shore server, but does not release the locks on the objects. Copies of the objects remain in the object cache.

Abort_transaction aborts the current transaction and throws away the contents of the object cache. Aborting a transaction releases all locks acquired while the transaction was running. While aborting a transaction undoes the effects of any operations on persistent objects, it does not affect the state of the application program or any of its transient data. The **SH_ABORT_TRANSACTION** macro is a partial solution to this problem.

Get_txstatus returns the current transaction status. If there is no active transaction, either because no transaction has been begun, or because all transactions have been committed or aborted, ‘NoTx’ is returned. If there is an active transaction then ‘Active’ is returned. If the current transaction has encountered a fatal error that requires the transaction to be aborted, but the application has not yet aborted the transaction, ‘Aborting’ is returned. When an error condition is returned by a Shore method call, before attempting to

handle the error, the application should use this method to determine whether the error was severe enough to cause the server to abort the transaction. If this method returns the 'Aborting' status, then the application must call **abort_transaction** or **SH_ABORT_TRANSACTION** to abort the current transaction, after which a new transaction can be started.

SH_BEGIN_TRANSACTION, SH_COMMIT_TRANSACTION, and **SH_ABORT_TRANSACTION** are an alternate interface to the transaction methods that employ the ANSI C **setjmp/longjmp** macros. **SH_BEGIN_TRANSACTION** is similar to **begin_transaction,** except that in addition to beginning a new transaction, it performs a **setjmp.** The argument to SH_BEGIN_TRANSACTION should be a variable of type shrc. Upon return, this variable will indicate whether the transaction was successfully begun. If it is set to RCOK, a transaction was started. Otherwise, it indicates either why a new transaction could not be started or why the current transaction was aborted. In the latter case, the value of the variable is whatever was passed to SH_ABORT_TRANSACTION.

SH_COMMIT_TRANSACTION is equivalent to **commit_transaction.**

SH_ABORT_TRANSACTION aborts the current transaction and performs a **longjmp,** which returns control to the application at the line containing the call to **SH_BEGIN_TRANSACTION.** The value of the *rc* passed to SH_ABORT_TRANSACTION becomes the return value of SH_BEGIN_TRANSACTION. Because of the use of setjmp and longjmp, the function containing the call to SH_BEGIN_TRANSACTION must not have terminated when SH_ABORT_TRANSACTION is called, although the two calls can be in different functions.

SH_COMMIT_TRANSACTION_INV and **SH_ABORT_TRANSACTION_INV** are equivalent to **SH_COMMIT_TRANSACTION** and **SH_ABORT_TRANSACTION,** respectively, but the former pass 'true' for the *invalidate* parameter, where the former use the default parameter of 'false.'

SH_DO tests its argument (an shrc) to determine if an error occurred. If not, then the macro returns. If an error did occur, the behavior of SH_DO depends on the context. If no transaction is active, or if the current transaction was begun with Shore::begin_transaction (not with SH_BEGIN_TRANSACTION), shrc is printed to the standard output and the process is terminated. If the current transaction was begun with SH_BEGIN_TRANSACTION, SH_ABORT_TRANSACTION is called with the given shrc as its argument. Typically, the argument to SH_DO is a method call that returns an shrc, as in the example below.

ARGUMENTS

Degree indicates the degree of consistency of the new transaction, with respect to operations on directories. (Degrees 3 and 2 are supported.) Degree 3 corresponds to full serializability, and should be used with care because looking up a registered object causes a share lock to be acquired on every directory in the path name. The default, degree 2, means that locks on the directories inspected and updated are released, but locks on directories created within the transaction are retained.

Invalidate indicates the fate of the object cache manager's internal data structures, and of any variable of type REF(T), when a transaction completes, either by committing or aborting. During the course of a transaction, the object cache manager builds data structures to keep track of the contents of the cache and to maintain the validity of any REF(T) variables. If a transaction touches a large number of objects, these structures can become quite large. Passing 'true' indicates that these structures can be freed, but this causes any existing variables of type REF(T) (global variables, procedure arguments, and local variables) to become invalid. Variables of type LOID are unaffected. An application should only pass 'true' if all of the following conditions are met:

1) the application intends to begin another transaction,

2) the set of objects that the next transaction will touch has little overlap with the set of objects touched by the current transaction, and

3) the application can guarantee that it will not make any use whatsoever of any existing REF(T) variables.

EXAMPLE

The following code fragment shows how the transaction macros can be used:

```
shrc rc;

// A subsequent call to SH_ABORT_TRANSACTION will return us here.
SH_BEGIN_TRANSACTION(rc);

if(rc){

    // Some error occurred.  The rc indicates why new transaction
    // could not be started or why the transaction was aborted.

    cerr << rc << endl;
}

else {


    // We successfully started a transaction; the main body of the
    // transaction goes here.

    SH_DO(operation 1);
    SH_DO(operation 2);
    SH_DO(...);


    // If we completed the body of the transaction without
    // errors, we try to commit the transaction.  Note
    // that if the commit fails, SH_DO will call
    // SH_ABORT_TRANSACTION for us.  The shrc returned by
    // SH_COMMIT_TRANSACTION will become the return value of
    // SH_BEGIN_TRANSACTION, above.

    SH_DO(SH_COMMIT_TRANSACTION);
}
```

BUGS

In general, **longjmp** interacts poorly with C++ destructors. In particular, if an automatic object (an object allocated on the stack, rather than from the heap) has a destructor, the destructor is called when the program exits the block in which the object was allocated. However, if a call to longjmp causes the program to jump out of a block, the destructors of any automatic objects declared in that block will not be called. Applications that use the transaction macros must be able to tolerate these destructors not being called.

Applications should be very cautious when committing or aborting a transaction while there are methods active on Shore objects (i.e., if the function in which the commit or abort is performed is a method on a Shore object, or if any of the functions below the current method on the stack are methods on Shore objects), as these operations cause all objects to be removed from the object cache. If a transaction is committed or aborted while a method is active on a Shore object, any subsequent references to the data members of the object (or any further method calls on the object), will have unpredictable results, even if a new transaction is immediately begun. Therefore, applications that commit or abort a transaction inside a method of an object should then immediately exit the method (it is safe to use the SH_ABORT_TRANSACTION or SH_DO macros inside a method, as they **longjmp** out of the method).

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

COPYRIGHT

SEE ALSO

setjmp(3).

REFERENCES

Jim Gray and Andreas Reuter, Transaction Processing: Concepts and Techniques