NAME

begin_xct, commit_xct, abort_xct, chain_xct, save_work, rollback_work, tid_to_xct, xct_to_tid, state_xct, prepare_xct, enter_2pc, recover_2pc − Class ss_m methods for transactions

SYNOPSIS

```
#include <sm_vas.h>  // which includes sm.h

static rc_t                 begin_xct(
    long                        timeout = WAIT_SPECIFIED_BY_THREAD);
static rc_t                 begin_xct(
    tid_t&                      tid,
    long                        timeout = WAIT_SPECIFIED_BY_THREAD);
static rc_t                 commit_xct(
    bool                        lazy = false);
static rc_t                 abort_xct()
static rc_t                 chain_xct(
    bool                        lazy = false);
static rc_t                 save_work(sm_save_point_t& sp);
static rc_t                 rollback_work(
    const sm_save_point_t&      sp);
static xct_state_t          state_xct(const xct_t*);

static xct_t*               tid_to_xct(const tid_t& tid);
static const tid_t&         xct_to_tid(const xct_t*);

#define max_gtid_len 256
#define max_server_handle_len 100

typedef opaque_quantity<max_gtid_len> gtid_t;
typedef opaque_quantity<max_server_handle_len> server_handle_t;

enum vote_t {
       vote_bad, // illegitimate
       vote_readonly, // no log written
       vote_abort, // cannot commit
       vote_commit, // can commit if so directed
};

static rc_t                 prepare_xct(vote_t &v);
static rc_t                 enter_2pc(const gtid_t &);
static rc_t                 recover_2pc(const gtid_t &,
    bool                        mayblock,
    tid_t&                      tid);
static rc_t                 set_coordinator(const server_handle_t &h);
static rc_t                 query_prepared_xct(int &numtids);
static rc_t                 query_prepared_xct(int numtids, gtid_t l[]);
```

DESCRIPTION

The above class **ss_m** methods all deal with transaction management. See the transaction section of the **SSM interface document** for more information.

TRANSACTION METHODS

begin_xct(timeout)

begin_xct(tid, timeout)

The **begin_xct** method begins a new transaction and associates the current thread with the transaction. The *tid* returns the transaction ID of the new transaction. The *timeout* parameter specifies the default amount of time the current thread should block when waiting to obtain a lock on behalf of the transaction. There are three commonly used values for this parameter:

WAIT_FOREVER
: This value indicates that the thread should block for up to an unlimited amount of time when waiting for a lock.

WAIT_IMMEDIATE
: This value indicates that the thread should not block for any locks.

WAIT_SPECIFIED_BY_THREAD
: This value indicates that the amount of time to block should be based on the thread's lock timeout setting. See **smthread_t(ssm)** for more details.

Using the default, WAIT_SPECIFIED_BY_THREAD, is usually appropriate as the default for the thread is WAIT_FOREVER.

Note: Nested transactions are not supported. Therefore, it is an error to call **begin_xct** while the current thread is already associated with a transaction.

commit_xct(lazy)

The **commit_xct** method commits the transaction associated with the current thread. If **commit_xct** returns successfully, all changes made by the transaction are guaranteed to be persistent, even if the server should crash. All locks held by the transaction are released. When the *lazy* parameter is set the **true,** the the transaction commit log record is not actually written to the disk. The transaction's changes (on volumes) are not persistent until the commit log record is written to the disk. Therefore, if a crash should occur immediately after a "lazy" commit, the transaction would be rolled back. However, if any subsequent log record makes it to disk, then the commit record will also be on disk. In addition, all lazy commit log records will be written to the disk every 30 seconds.

abort_xct()

The **abort_xct** method rolls back all changes (on volumes) made by the transaction and ends the transaction.

chain_xct(lazy)

The **chain_xct** method commits the current transaction and begins another one just as **begin_xct** would do. In addition, all lock held by the current transaction are transfered to the new transaction. The *lazy* parameter functions as it does for **commit_xct.**

save_work()

The **save_work** method marks a "save point" and fills the *sp* parameter with information about it.

rollback_work()

The **rollback_work** method uses the log to roll back all changes (on volumes) made since the save point indicated by the *sp* parameter. No locks are released.

TRANSACTION STATES

state_xct()

The **state_xct** method returns the current state of the transaction pointed to by *xct.*

The states are `{xct_stale, xct_active, xct_prepared, xct_aborting, xct_chaining, xct_committing, xct_ended}`.

ATTACHING TRANSACTIONS TO THREADS

The transaction pointer for the current threads is available via **smthread_t::xct.** See **smthread_t(ssm).** for more information.

TRANSACTION IDENTIFIERS

Transactions are identified by a transaction ID, **tid_t. Internal to the SM, there xct _t,** holding information about the transaction. So, a pointer to an **xct_t can also be used to** SM interfaces, including **smthread_t(ssm)** use **xct_t** pointers to avoid the transaction from **tid_t.**

tid_to_xct()

The **tid_to_xct** methods converts a transaction ID in to an **xct_t** pointer.

xct_to_tid()

The **xct_to_tid** method converts a **xct_t*** to a transaction ID.

DISTRIBUTED TRANSACTIONS

The Shore storage manager can participate in transactions coordinated by other software modules that employ the "presumed abort" two-phase commit protocol. The coordinator in such a situation is external to the Shore storage manager; it is assumed to have its own stable storage, and it is assumed to recover from failures in a *short time ,* the precice meaning of which is given below.

A prepared transaction, like an active transaction, consumes log space and holds locks. Even if a prepared transaction does not hold locks needed by other transactions, it consumes resources in a way that can interfere with other transactions. If a prepared transaction remains in the system for a long time while other transactions are running, eventually the storage manager needs the log space used (reserved) by the prepared transaction. A coordinator must resolve its prepared transactions before the storage manager effectively runs out of log space for other transactions in the system. The amount of time involved is a function of the size of the log and of the demands of the other transactions in the system.

For the purpose of this discussion, the portion of a global transaction that involves a single Shore transaction is calld a *thread* of the global transaction.

A Shore transaction participates as a thread of a *global transaction* as follows:

Start a Shore transaction with **begin_xct.**

Acquire
: a global transaction identifier from the coordinator.

Indicate
: to the Shore storage manager that this Shore transaction is a thread of a global transaction, and associate the global transaction identifier with this thread by calling **enter_2pc.**

Associate
: a coordinator with the transaction for recovery purposes, by calling **set_coordinator.**

Prepare
: the Shore thread of the transaction and get the Shore storage manager's vote with **prepare_xct.** It is an error to commit a global transaction thread without first preparing it. It is an error to do anything else in a transaction after it is prepared, except to end the transaction or retry the prepare (to get the vote again).

Convey
: the vote to the coordinator, and determine the transaction's fate from the coordinator.

End
: the thread with **commit_xct** or **abort_xct.**

GLOBAL TRANSACTION IDENTIFIERS

A global transaction identifier is an opaque value to the Shore storage manager. It uses a template class defined as follows:

```
template <int LEN> class opaque_quantity {
private:
    uint4         _length;
    unsigned char _opaque[LEN];
public:
                           opaque_quantity();
                           opaque_quantity(const char* s);
    friend bool operator  ==(const opaque_quantity<LEN>&,
                          const opaque_quantity<LEN>&);
    friend ostream& operator <<(ostream &o, const opaque_quantity<LEN>&);
    opaque_quantity<LEN>& operator=(const opaque_quantity<LEN>&);
    opaque_quantity<LEN>& operator=(const char*);
    opaque_quantity<LEN>& operator+=(const char*);
    opaque_quantity<LEN>& operator-=(uint4 len);
    opaque_quantity<LEN>& append(const void* data, uint4 len);
    opaque_quantity<LEN>& zero();  // zero entire max-sized data structure
    opaque_quantity<LEN>& clear(); // zero length only
    operator              const char *();
    void *                data_at_offset(uint i)  const;
    uint4               wholelength() const; // including _length member
    uint4               length() const;      // excluding _length member
    uint4               set_length(uint4 l); // of _opaque part only
    void                ntoh();              // put in host byte-order
    void                hton();              // put in net byte-order
    bool                is_aligned() const; // to sizeof(int)
};
```

VOTING

The Shore storage manager implements the "read-only optimization" for presumed-abort. If a prepared transaction did not log any updates, the transaction is committed at the time it is prepared, and the vote returned indicates that the transaction thread is read-only. Once the vote is communicated to the coordinator, and the coordinator has recorded this vote on stable storage, this thread of the global transaction can be omitted from all further processing of the transaction.

The votes are `{vote_bad, vote_readonly, vote_abort, vote_commit}`.

CRASH RECOVERY

If the application (value-added server) should crash during a two-phase commit, a new application (representing the coordinator) must run, and it must contact the Shore storage manager in order to complete the two-phase-commit protocol.

If the application crashes before the prepare is done the transaction thread is aborted.

If the application crashes during the first phase (after the prepare is done, but before the vote is written to stable storage, the application must retry the prepare phase to get the vote and resolve the transaction.

If a crash occurs during the second phase (after the prepare is done and its vote is written to stable storage, but before the transaction is resolved), the application cannot always tell if the second phase completed. It is always safe to try again to complete the transaction thread. If the transaction thread is unknown to the Shore storage manager at this point, the second phase completed.

In order to locate a prepared transaction after a crash, the application calls **recover_2pc.** If a prepared thread with the given global transaction identifier is found, the (local) Shore transaction identifier is returned, and the thread is attached. The application can subsequently call **commit_xct** or **abort_xct.**

The Boolean argument *mayblock* indicates whether the application considers it acceptable for the **recover_2pc** call to block (e.g., in the event that it is awaiting connection to its internal coordinator).

After recovery after a crash, a value-added server may discover what transactions were prepared and need recovery by calling the two forms of **query_prepared_xct .** The first call returns the number of such transactions. With that information, the value-added server can allocate memory in for storing the global transaction identifiers of the prepared transactions. The value-added server then invokes the second form of **query_prepared_xct** to get a list of the global transaction identifiers, and then recover the prepared transactions.

ERRORS

EXAMPLES

ToDo.

VERSION

This manual page applies to Version 2.0 of the Shore Storage Manager.

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. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

COPYRIGHT

SEE ALSO

lock(ssm), smthread_t(ssm), intro(ssm),