# The ACID SimTools Documentation

Anakreon Mentis, Panagiotis Katsaros

January 27, 2008

**Abstract**

In modern network centric information systems, multi-tier applications are now becoming mainstream. Their design is mainly based on the fundamental object-orientation principles and the employed recovery solutions ensure at-most-once service request processing, through some form of "all-or-nothing" guarantee by an underlying transaction processing architecture. Transactional objects cooperate with an assigned transaction manager to provide system-wide Atomicity, Consistency, Isolation and Durability (ACID) properties for the performed operations. Architectural solutions are standardized in reference specifications, with the most notable case the one of the OMG Object Transaction Service (OTS). However, there is a lack of simulation tools with metrics, which can provide insight into the performance and availability trade-offs that arise when applying different combinations of concurrency control, atomic commit and recovery protocols (and protocol parameters). This article introduces ACID Sim Tools, a publicly available interactive and batch-mode simulation tool and an open source library developed to cope with the foresaid performance design considerations.

## 1 Introduction

In the past year we have observed the increasingly importance of distributed network oriented applications. The literature on transaction processing offers many protocols. Some of them reduce the network traffic by reducing the number of messages exchanged between the cooperating parties, other reduce the blocking time or the impact of the Input/Output required by the protocols on the overall performance. A tool which can simulate the different factors impacting the performance and provide metrics on the performance and the impact of the various parts of the protocol would help considerly in the study of the protocols.

Section 2 introduces basic concepts and definitions. The terminology of *OMNETPP* used in this article is explained in section 3. Section 4 describes in detail the parts of the simulator and their function.

## 2 Basic definitions and problem statement

An object is a collection of attributes and methods. State of the object is called the values of the object's attributes. The methods provide the means to observe or alter the state of the object.

An object has a size, that is the number of bytes required to store the object in memory or in stable storage.

The methods are divided in two categories. A method is called read-only if it does not alter the state of the object when invoked and write method otherwise. Each method has a duration which is the time required for the compilation of the method invocation.

An object resides on a server who provides the environment for it's existence.

A transaction is a collection of methods invocation. The objects where the methods belong may exist in one ore more servers. If there is only one server involved, the transaction is called local and distributed if there are more then one server. For distributed transactions we distinguish the servers into workers and coordinator. The coordinator is the server which coordinates the cooperation of all the servers involved in the distributed transaction.

Transactions should fulfil the ACID properties. The properties are satisfied by an atomic commit protocol, a concurrency protocol and a method for ensuring the durability of objects state.

Two transactions belong to the same class if and only if the sets of the operations they contain are identical. In other words, the class of a transaction defines the operations the transaction invokes when processed.

### 2.1 Atomic Commit Protocol

The ACP guaranties that the effects of a transaction execution are visible only if the transaction is committed. If a transaction is aborted it's effects are undone (rolled back) and before the completition of the transaction's execution the intermediately effects are not visible.

For now, ACID SimTools can simulate variations of the 2PC family of protocols. We provide the implementations of the basic 2pc protocol and two variations, the presume commit and presume abort protocols. ACID SimTools also provide the means to implement other variations. A detailed description of the extendibility of ACID SimTools is provided in the following sections.

### 2.2 Concurrency protocol

Concurrency protocol allows the concurrent execution of transactions in an isolated manner. The transactions do not interact with one an other as if each transaction was the only one executing.

There are many concurrency protocols in the literature. We have implemented a two phase locking variation and timestamp ordering. One can combine different concurrency protocols and atomic commit protocols and study their combined behavior and performance.

### 2.3 Durability

Server failure is an anticipated event. In such an occasion the server should be able to resume it's operation. The durability aspect of the system stores the state of the objects and the information on the execution of transactions needed for recovery after a failure.

In our implementation, this information is stored in a log file. Periodically, redundant information such as execution monitoring of finished transaction and deprecated states of objects are removed from the log file. This process is called checkpoint.

## 3 *OMNETPP* terminology

A C++ class which extend the cSimpleModule provided by the *OMNETPP* framework is called a module. A module may have zero or more parameters which are set from the user. A parameter alters the behavior of the module during the simulation.

Channels provide the means for message exchange between modules. A channel connects two modules and is consisted of an input port and an output port. An input port is used by a module for incoming messages where an output port is used to send messages to other modules. A channel may have a delay attribute. If delay is non zero, the messages is delayed before it is delivered to the target module.

A message is a C++ class which extend the cMessage class provided by the *OMNETPP* framework. Messages have various attributes such as an identifier for the sender, a reference to the sender, the sending time and others. Messages provide the means for communication among the various modules which consist the simulator.

Self–message or an internal event is a message which the sender and receiver are the same entity. Such messages are used to implement timers.

## 4 Modules and their parameters

*Some words here*

### 4.1 QSource

QSource generates transaction requests for a transaction class. The request is sent to the coordinator of the transaction through the *out* gate. The QSource accepts messages for failed transactions through the *in* gate. The failed transactions are resubmitted to the coordinator.

The parameters it expects are:

- mean. Is a numeric parameter which defines the mean time between two requests. This is the mean of an exponentional distribution which requests obey.

- timeout. Is a numeric parameter which defines the time in which the voting phase should be completed. If a transaction has not finished the voting phase during this time, it is aborted.
- classId. An integer parameter. The value of classId is used to distinguish one class of a transaction from an other.
- nodes. An xpath expression which defines the part of the xml document which defines the attributes of the transaction the QSource instance submits (a more detailed description follows in the section 10.1).

## 4.2 QSink

QSink is the module which accepts all completed transactions. If a transaction is aborted, one can define how many times an aborted transaction may be resubmitted.

The module has N *in* gates from where it accepts completed transactions from their coordinator and M *out* gates from where dispatches failed transactions to the appropriate QSource module for resubmission. The value of N is equal to the number of Acp (see below) instances and the value of M is equal to QSource instances.

It also has an *out* gate named stat_out which uses to send messages to the statistics module. When a transaction is aborted, the coordinator is unable to tell if the abort is final or it will be submitted later for reprocessing. QSink informs the statistics module if the abort is final.

It has two parameters named max_submition_count of integer type which defines how many times the transaction may be resubmitted and resubmit_delay which defines the delay before resubmitting an aborted transaction. If the value of resubmit_delay is negative or zero, there will be no delay and the transaction is resubmitted imediatly to the appropriate QSource.

## 4.3 Concurrency control

Concurrency control module is responsible for the concurrency and isolation of transaction execution. The module communicates with the ACP module via two gates. It listens for incoming messages from the *in* gate and sends messages through the *out* gate. There are no parameters for this module.

## 4.4 Stable storage module

The module receives messages from the ACP module which regard monitoring of transaction execution and stores the objects state in stable storage. It is responsible for gathering recovery information after a system crash. The information are used for transaction recovery. Messages are sent to the ACP module through the *out* gate.

The parameters it accepts are

- read. A float parameter which defines the latency when reading data from stable storage.
- write. A float parameter which defines the latency when witting data to stable storage.

## 4.5 Atomic commit protocol

Acp is the Transaction Manager component of the simulator. It accepts transactions for which is coordinator from QSource modules and request from other Acp instances for the transactions where it acts as an worker.

This module exchanges messages with all other modules of the simulation, thus it many gates. The gates are

- in. An array of gates of length equal to the number of the transactions for which the module is the coordinator. To each in gate, a QSource instance is connected and submits transaction requests.
- proc. An array of gates of length equal to the number of other Acp modules with which the module cooperates in distributed transactions.
- lock_in. Is used to receive messages from the Concurrency control module.
- log_in. Is used to receive messages from the Log Manger module.
- out. Send the completed transactions to the QSink module.
- oproc. Sends messages to the other Acp modules with which it cooperates in distributed transactions.
- lock_out. Sends messages to the Concurrency control module.
- log_out. Sends messages to the Log Manger module.
- stat_out. Sends messages to the statistics module.

The parameters of Acp are:

- process. An integer parameter which defines the maximum number of concurrent threads. Each task in order to start executing needs a free thread. Hence, the greater the value of *process* the greater the number of simultaneously executing tasks.
- checkpoint. Defines the interval between two checkpoints.
- crash. Is the mean of an exponentional distribution when a system failure should occur.
- crashPause. Defines the amount of time which should elapse before the system starts recovering. During this time, the server is out of order and can not accept messages.

- xmlId. A unique identifier of integer type.
- objects. An xpath expression which defines the part of the xml document containing information about the objects the server hosts (see 10.1 ).

### 4.6 Statistics

The statistics module collects information from the ACP and QSink modules and generates the result of the simulation. It is a passive module in the sense that it doesn't send messages to any module, it just observers the simulation without affecting it's execution.

ACP modules send messages regarding significant events for the system or individual transactions. Whenever a significant system event occurs a message is sent immediately to statistics module.

The system events are listed at table 4.6:

The statistics module is informed about events regarding a transaction when the transaction is either aborted or committed. The reason is twofold:

- The statistics should not be affected by ongoing transactions since no decision is made about them
- Due to the number of active transactions and the significant events regarding them, sending a message for each event would hurt performance.

Table 4.6 shows the attributes sent to the statistics module when the processing of a transaction is over:

The QSink module will send a message when an aborted transaction is abandoned and not resubmitted. The module has array of *in* gates equal in length with the number of the Acp modules plus one more gate for the QSink module.

## 5 Events

Each module of the simulator communicates with the other modules by exchanging messages. A message is a C++ class which inherits the cMessage class of the *OMNETPP* framework.

Most messages exchanged regard some transaction. In order to disseminate the information regarding a transaction, we use the TransactionMessage class which has as attributes the transaction's unique identifier and it's class. An other attribute of the message useful when a new transaction is submitted for processing, is a pointer to a Transaction object. In other occasions the pointer is *NULL*.

JobMessage class is a dependant of the TransactionMessage class and has one extra attribute. The attribute stores a ObjectRef instance which describes an operation(the server where the object belong, a unique identifier for the object, a flag which indicates if the method is read-only and the method's duration when executed. JobMessage instances are used only in self–messages to schedule the termination of a method invocation.

| Message | Description |
|---|---|
| SYS_THREAD_COUNT_INC | Current level of multiprogramming increased |
| SYS_THREAD_COUNT_DEC | Current level of multiprogramming decreased |
| SYS_CRASHED | Server crashed |
| SYS_RECOVERY_STARTED | System recovery initiated |
| SYS_RECOVERED | Recovery is over. Server is functioning properly |
| SYS_CHECKPOINT_STARTED | Checkpoint started |
| SYS_CHECKPOINT_ENDED | Checkpoint finished |

Table 1: Messages regarding system events

| Attribute | Description |
|---|---|
| arrival | Time of transaction arrival |
| vote | Time when voting phase was entered. -1 if transaction aborted before voting phase started |
| abort | Time when the decision to abort was reached (if coordinator) or was received (if worker). If the transaction is committed, the value of abort is -1 |
| commit | Time when the decision to commit was reached (if coordinator) or was received (if worker). If the transaction is aborted, the value of commit is -1 |
| recovery | Time when recovery started. If the transaction is committed the value is -1 |
| locks | A collection of object id and time value pairs. Stores the time required to gain a lock for each object |

Table 2: Information sent to statistics about a transaction

Finally, StatisticMessage instances are sent to the Statistic module and carry statistical information. 5 shows the class hierarchy of the messages used in the simulator. 5 shows the events used for the simulation and a description of their function.

# 6 Extending ACID SimTools

The goal of ACID SimTools is not just to provide a simulator for Transaction Processing Protocol but also to provide an extendible set of classes which will aid other researchers in implementing different protocols. In order to achieve this goal we utilize the class inheritance mechanism offered by C++, the message based contract which *OMNETPP* provides and a code generator for Finite State Machines for the transaction processing protocol.

## 6.1 Class hierarchy of Atomic Commit Protocols

Figure 2 presents the class hierarchy of atomic commit protocols. The BaseAcp class is the root of the hierarchy and provides the methods and attributes which we believe are useful in all conceivable kinds of ATP protocols. The methods of BaseAcp are short and well documented. No assumptions are made about the concurrency control, state serialization protocol, existence of system failures and recovery etc. The class is abstract in order to make it a good candidate for an ancestor class of a different ATP protocol.

The ConcreteAcp class is a descendant of BaseAcp and provides functionality required for system recovery. Still the class is abstract enough to make it a possible ancestor of a ATP protocol which support system failures and recovery. The methods of the class are declared virtual and as such allow refinement of their behavior by the descendants of the class.

BasePresume is a descendant of the ConcreteAcp and gathers the common functionality of the variations of the Base 2PC protocols such as PresumeCommit and Presume Abort protocols. It implements abstract methods of ConcreteAcp. Still the class is abstract because it does not include the code for the FSM needed to process transactions (see 6.3 ).

BaseTimestamp is be common ancestor of all 2PC variations which use timestamp ordering instead of lock based concurrency control. Is a minor modification of BasePresume class.

## 6.2 Message based contract

We have already mentioned that the modules do not interact with one another with method calls but instead use messages to exchange information and accomplish calculations. This functionality offered by *OMNETPP* allows the user of the simulator to choose at runtime the different protocol implementations available for a certain aspect of the system, such as concurrency control, state serialization and atomic commit protocol. The only requirement imposed on

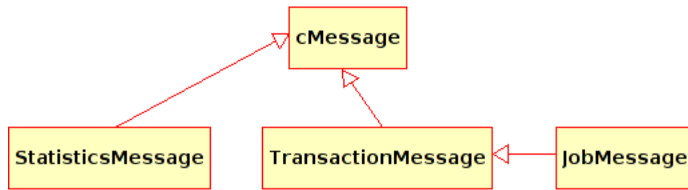


Figure 1: Class hierarchy of messages

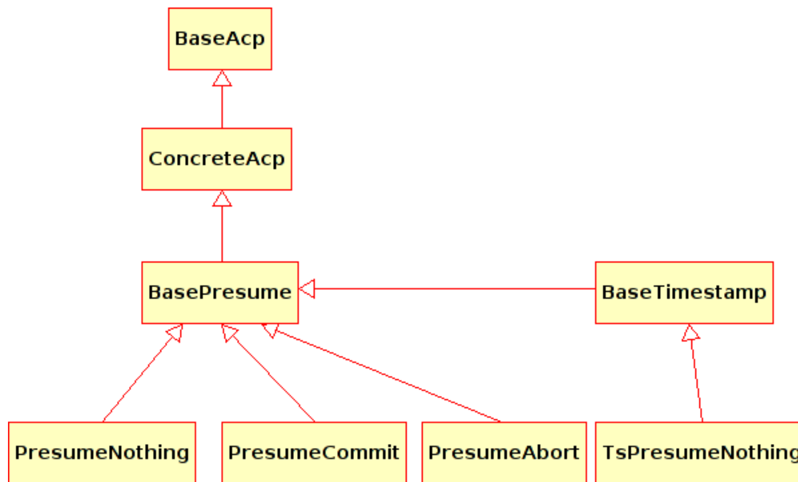


Figure 2: Class hierarchy of Atomic Commit protocols

the existing or the future implementations is that the message contract is not broken. In such a case the simulation will stop with a runtime error.

We have already shown the messages exchanged in table 5 . An implementation of a module is expected to recognize a message sent to it from an other module and respond with an appropriate message.

## 6.3 The finite state machine

A protocol can be described with a *FSM* (finite state machine). It is certainly possible to implement the *FSM* directly in C++. However, it was obvious from the beginning of the development of ACID SimTools that a direct implementation approach was complex and would discourage farther development by other researchers.

Instead, we use a simple description of the *FSM* and built a tool which generates the appropriate C++ code. The tool also generates a state transition graph which is useful for demonstration or inspection purposes.

We use three classes to support the *FSM* at runtime. The hierarchy of the classes is shown at figure 6.3 . Fsm is the base class of *FSM* classes. It stores the current state and provides two methods to reset the state or observe the current state of the *FSM* . If debugging is enabled, it also stores the previous states of the *FSM* and the creation time. We will elaborate on debugging in the following sections.

Fsm is extended by the TransactionFsm class which also stores information about the transaction it regards. It stores the class of the transaction, a set of jobs which are executed on the server at a given instance and some information needed for statistics.

TransactionFsm is further refined by the CoordinatorTransactionFsm which is the kind of *FSM* needed by the coordinator. It stores a pointer to a Transaction object, the number of servers expected to acknowledge the coordinator's decision, and a set of server ids who have already acknowledged the decision. It also stores a set of server ids who have voted for the outcome of the transaction.

A *FSM* starts in an initial states and terminates in a final state. The initial state is named ST_EMPTY and the final state ST_FINISH. When an event occurs, the *FSM* will move to a new state in response to the event. After the transition is completed, a set of actions are executed. If the *FSM* reaches the ST_FINISH state, the *FSM* is no longer needed and the transition is completed. The next state of the *FSM* depends only on the current state and the kind of the event which occurred.

When a new request for service arrives to the ACP module, a new CoordinatorTransactionFsm instance is created which is responsible for monitoring the execution of the transaction. For all other events, the event kind is propagated to the *FSM* responsible for the transaction the event regards.

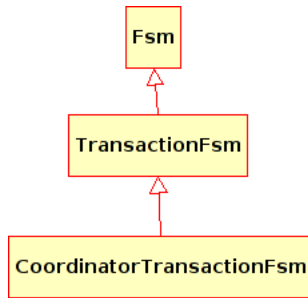


Figure 3: Hierarchy of *FSM* classes

### 6.4 States

Except for the initial and final state of the *FSM* , states are protocol dependant. Table 6.4 describes the meaning of the states used in the Presume Nothing, Presume Commit and Presume Abort protocols.

### 6.5 FSM description file

There are two distinct roles in the 2PC family of protocols. Coordinator should manage the process of reaching a global decision and also manage his part of the work. The worker needs to accomplish his part of the work and also participate in the decision reaching process and comply with the decision of the coordinator.

The description file, contains state transitions and comments. A comment is defined by inserting the characters "//" at the beginning of the line and ends at the end of the line.

Each state transition is consisted of four elements.

**Role** Does the state transition regard the coordinator ($c$), the worker ($w$) or both roles (_).

**State** Defines the state of the *FSM* before the transition.

**Event** The event's code which causes the transition.

**Next state** The state of the *FSM* after the transition.

**Actions** The actions which are executed after the state transition is completed.

The actions which are called after the *FSM* state has changed are methods of the class implementing the protocol or a method of a parent class (like BaseAcp for example). All methods accept at least one parameter, an integer value which uniquely identifies a transaction. There are methods which accept one more parameter. The parameter is specified in the configuration file by the usage of special codes enclosed in brackets.

The extra parameter may be

**sender** is substituted by the id of the server which sent the message. The sender of the message *should* be an ACP module.

**id** is replaced by the id of the server which received the message.

**job** is replaced by the ObjectRef object which the message caries. A runtime error occurs if the message received is not an instance of the JobMessage class.

**msg** is replaced by a pointer to the message received. A runtime error occurs if the message is not an instance of the JobMessage class.

An example taken from the configuration file of Prepare Nothing (PN) protocol follows:

```
1  //when a lock is aquired
2  _,     ST_LOCK,   LOCKED,         ST_JOB,    startJob:statLockAquired
3  c,     ST_EMPTY,  INIT,           ST_INIT,   sendInitLog:scheduleTimeout
4  c,     ST_VOTES,  OUTCOME_ASKED,  ST_VOTES,  -
5  c,     ST_VOTES,  VOTE_LOGGED,    ST_VOTES,  collectVote{id}
6  w,     ST_WAIT,   START_JOB,      ST_LOCK,   workerLock{job}:statGettingLock
7  w,     ST_JOB,    JOB_FINISHED,   ST_WAIT,   removeJobMessage{msg}:jobEnded
8  w,     ST_EMPTY,  OUTCOME_ASKED,  ST_FINISH, replyAbort{sender}
```

Line one contains a comment describing line two. Line two defines a state transition which applies to coordinator and worker. The actions receive only one parameter, the transaction id. Line three regards the coordinator. The event received at line four is ignored. The initial and final state are the same and no actions are taken. Line five illustrates an example of action call with one more parameter, the server's unique identifier. Line six illustrates an action call with a *job* parameter where line seven and eight illustrate an example for the *msg* and *sender* case respectively.

Line eight states that it describes a state transition for the worker but this is an artefact of a limitation of the simulator. The simulator creates a new CoordinatorTransactionFsm instance when the message with kind `CLIENT_SEND_TRANSACTION` is received. In any other case, the event is propagated to the *FSM* monitoring the transaction execution. If there is no *FSM* for the transaction however, the simulator assumes that the message regard a worker and creates a new instance of TransactionFsm class with initial state ST_EMPTY. This assumption is not always true. Line eight describes a situation where the coordinator has finished the transaction processing and is asked by the worker about the outcome of the transaction. Since the *FSM* created is an instance of TransactionFsm, we have to state that the transition regards the worker, event if this is not true.

Generally, the *FSM* is non deterministic. The reason for this is that there are information which decide the next state of the *FSM* external to the automaton. An example is the vote collection phase. When a worker votes for the outcome of the transaction, coordinator should either wait for an other worker to vote or proceed with the announcement of the final decision to all workers.

In order to decide which of the two should happen depends on the total number of participants but the *FSM* does not have this information. The code generator can understand which transitions are non deterministic. Non determinism occurs when there are two or more transitions where initial state and event are the same. In this case, a method is called with one parameter of type CoordinatorTransactionFsm if the non determinism occurs in the coordinator *FSM* or TransactionFsm if regards the worker. The method is expected to return the next state of the *FSM* and thus resolve the non determinism. The name of the method is created by concating the name of the current state and event which cause the non determinism. If for example the current state is ST_VOTES and the event is VOTED then the method is called resolveVotesVoted.

# 7 Validation of state description file

During the development of a new protocol it is possible that the description of the *FSM* is incorrect. Possible problems are:

- There are unreachable states. The design of the protocol is problematic although there will be no run-time errors.
- Not all paths terminate. If such a path is followed during simulation, some transactions will never terminate, consuming memory and producing wrong statistical results.
- A combination of (state, event) is not handled. If this occurs during simulation, the simulator will print an error message and halt.

The graphs generated by *genfsm and* GraphViz can help resolving the first two problems. Both unreachable states and the paths which do not end in state ST_FINISH, are easily detected on the graph.

An other utility called graph_check helps to resolve the third problem. The utility is not fully functional yet. It does report combinations of (state, event) which would halt the simulator but it also reports combinations which could not happen at all.

## 7.1 The implementation of graph_check

graph_check expects two parameters, the first is the state description file and the second is a file which describes some properties of the methods invoked when the *FSM* enters a state. We call the second file, “method description file”.

The method description file tells for each method an event which is generated on invocation of that method. An example follows:

```
aknowledgeAbort,AKNOWLEDGED,c,w
askDecision,OUTCOME_ASKED,c,w
cancelAskOutcomeMessage,-ASK_OUTCOME,w
lockObject,LOCKED,c,lock_manager
```

```
logAbort,ABORT_LOGGED,b,log_manager
stopJobs,-JOB_FINISHED,b
```

Each line describes a method and contains at most four fields. Fields are separated by commas. The first field is the method's name. The second is the name of an event. If the first character is "-" then the method will remove a scheduled event of that type. For example, cancelAskOutcomeMessage will remove a scheduled ASK_OUTCOME message. The third field indicates the receiver of the event. Possible values are c (coordinator), w (worker) and b (both). The fourth field is optional. If it exists, it indicates the producer of the event. The producer may be:

- c. Coordinator
- w. Worker
- lock_manager
- log_manager

If the fourth field is not supplied it is assumed that the producer is the same as the consumer of the event.

After the definition files are loaded, the program starts building each possible path starting from ST_EMPTY for both worker and coordinator.

Tha to synexiso ayrio

## 8 Code generator

The BaseAcp class, included in the ancestry line of every ACP protocol of ACID SimTools, declares two pure virtual methods. The handleTransactionMessage and getStateNames. The implementation of those two methods is generated from the state transition description by the *genfsm* tool.

Every TransactionMessage event, except the CLIENT_SEND_TRANSACTION event, is propagated to the handleTransactionMessage method. handleTransactionMessage accepts three parameters:

- msg. A pointer to the TransactionMessage received
- senderId. The unique identifier for the sender of the message. senderId equals -1 if the sender is not a ACP module.
- isCoordinator. Indicates if the message is intended for the coordinator of the transaction.

The generated method implements the transitions as described in section 6.5 . If the transition is not described in the state transition file, the simulation is aborted with an error message indicating the cause. We could have chosen to ignore unanticipated transitions but this could compromise the results of the

simulation or cause hard to find bugs at an other point of execution. This choice makes the state transition file more verbose, since the implementer of a 2PC protocol has to describe even the transitions which should be ignored, but we believe that this cost is acceptable on the long term.

During the implementation of a new protocol unanticipated events will occur. For this reason, the generated code includes instructions useful for debugging. There are two pre-processor instructions which include or exclude the debugging code.

**DEBUG_FSM** If this pre-processor variable is defined, the *FSM* will track all the previous states it has reached. In case an event causes a simulation halt, before the simulation is aborted, all the previus states of the *FSM* , the id of the transaction and the code of the event which caused the abort are printed to the standard output. This information will help the implementer of the protocol to track the problem.

**DEBUG_MESSAGES** During the simulation, each state transition is printed on standard output. The information printed are the event code, the state of the *FSM* when the event was received and the next state. The output is very verbose but it is useful in some occasions.

DEBUG_FSM also controls if the second method generated, the getStateNames, is compiled. getStateNames creates a map where the keys are numeric event codes and the values are the names of the events. This method is useful only when the previous states of the *FSM* should be printed.

Two files are produced by *genfsm*. Suppose the class implementing the protocol is named PresumeNothing. The file generated_presume_nothing.cc contains the implementation of the two methods mentioned above. The other generated file is named states_presume_nothing.hh. It contains the definition of a C++ enumeration. Each element of the enumeration is a state of the *FSM* . Care should be taken to prevent two different states having the same numeric code. In case a protocol class extends an other concrete class, the user can inform the *genfsm* tool to take into account the states of the ancestor class. If the descendant has more states then the ancestor, the state enumeration will have only the states introduced by the descendant, ensuring that the numeric codes do not overlap with the states of the ancestor. If the descendant does not introduce any new states then no state file is generated and the states of the ancestor are used instead.

An other product of the state definition file is a state transition graph. Each state is displayed as an oval shape with the name of the state inside and events are vertices connecting two states. The event's name is shown on the vertex. *genfsm* has an option to simplify the graph. That is to not show transitions of ignored events. This makes the graph more readable.

Figure 8 shows the simplified state transitions for the coordinator of Presume Nothing protocol.

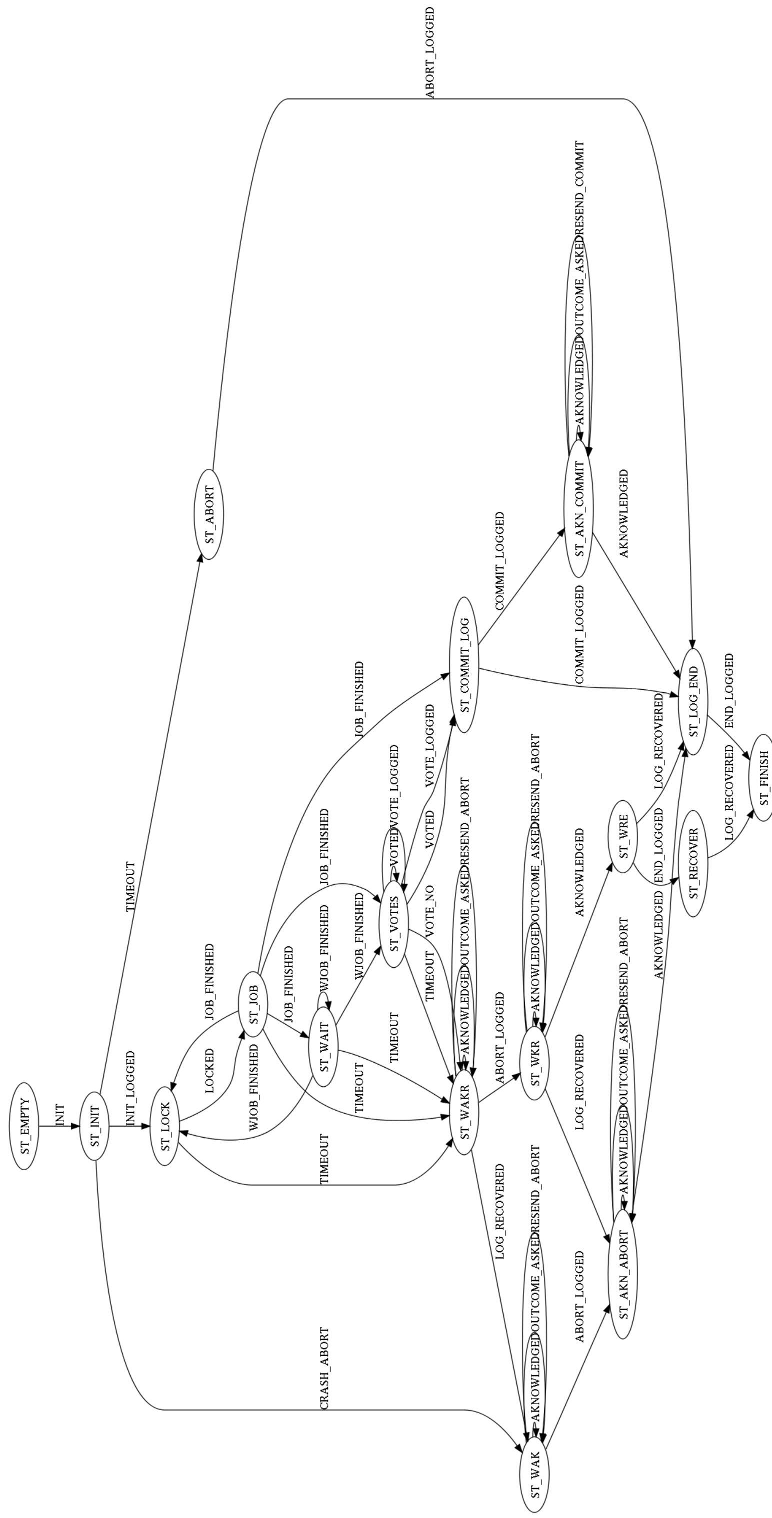




Figure 4: State transitions for coordinator of Presume Nothing protocol

# 9 Examples of protocol implementation

Until now we have implemented three different variations of 2PC protocols, the Presume Nothing, Presume Commit and Presume Abort protocols. Also, we provide two different concurrency control schemes, a lock based concurrency control and timestamp ordering protocol.

An implementation of a 2PC should support recovery from system failures, thus, ConcreteAcp should be in the class hierarchy of the class. We recognized that all variations of Presume family of protocols are very similar. For this reason a class called BasePresume is the common ancestors of all protocol implementations.

The BasePresume class implements the methods to resolve nondeterminism of the *FSM* and provides concrete implementation of the various phases of system recovery. BasePresume is still an abstract class since it does not implement the state transitions of the *FSM* .

PresumeNothing class is a descendant of BasePresume. It differs from BasePresume because it provides the implementation of handleTransactionMessage generated from the state transition file for the Presume Nothing protocol.

The next protocol we implemented is the PC (Presume Commit) protocol. The ancestor of the class module is BasePresume. Strictly speaking, the PresumeCommit class needs not provide a different implementation of any inherited method, except of course the handleTransactionMessage. The only overridden method is the handleRecoveryCommited method. The implementation of handleRecoveryCommited method causes the simulator to abort in case it is called. This method should never be called and we provided this implementation in order to verify this restriction. Hence, the implementation of PC did not require any code at all but it does require a different state transition file which reflects the difference with Presume Nothing.

PC stores less log entries then PN (Presume Nothing) protocol. PN stores a log entry indicating the end of a committed transaction but presume commit does not. This difference leads to a minor modification of the checkpoint procedure and system recovery. While LogManager class (the implementation of the Stable Storage module) considers a transaction completed only if the "end" log entry is found, the stable storage module of PC should consider completed all transactions which are aborted and the "end" log entry is found or committed transactions for which no "end" entry must exist. We derive the PrcLogManager class from LogManager which implements this difference. The total lines of code required for the PC variation of stable storage are 38, including comments and only two methods must have a different behavior from the corresponding methods of the ancestor.

An other slightly more complicated example is the implementation of timestamp based concurrency control. The LockManager class which implements the lock based concurrency control is not adequate for this family of protocols. The concurrency control module is implemented by the TimestampConcurency class. Also, all the protocols which use timestamps instead of locks differ in how they handle the recovery of a transaction which was halted in the voting phase due

to a system failure. For this reason we derive a class named BaseTimestamp which inherits BasePresume and overrides the handleRecoveryVoted method.

The state transition file of the timestamp based concurrency control for PN protocol need to take into account that timeouts are not employed for deadlock prevention since no locks are used. We derive the TsPresumeNothing class from BaseTimestamp where the handleTransactionMessage method is generated from the state transition file. No other functionality is added to BaseTimestamp other then the implementation of the method.

In conclusion, the steps required to implement an other variation of 2PC protocol are:

- Check if any of the existing concurrency control implementations is adequate for the protocol. If not, either derive a new class from an existing implementation, or provide a class which implements the concurrency control.
- Use a stable storage protocol from the existing implementations or if none is adequate, derive a new one from the available classes.
- Derive a class from BasePresume or any other adequate class for the atomic commit protocol. Override the methods of the ancestor if necessary and re-declare the handleTransactionMessage. The implementation of the method is generated by the *genfsm* tool.
- Write a new state transition file for the protocol. An existing state transition file can be used as a template. *genfsm* will use the file to produce the implementation of handleTransactionMessage.
- If the protocol uses an event not recognized by existing protocols, add the new event to the enumeration defined at event.hh

If instead of a new 2PC protocol we needed to change the locking scheme, use for example a variation of 2PL (two phase locking), all needed done is to configure *OMNETPP* to use the new variation of 2PL instead of the LockManager class.

# 10 Configuration

In order to run an experiment, the user needs to define the topology of the network, the classes of transactions which participate and finally the values of module parameters.

Topology is defined in the qnet.ned file. It defines the C++ classes which model each module, the names and types of parameters each module accepts and the connections between the modules.

Trnasaction classes are defined in an xml file named config.xml. In this file the user defines the ACP servers, the objects each server manages and the methods of each object. For each server there must be an instance of an Acp

protocol module which simulates the server. To define transaction classes the user should specify a series of method calls, the object where the method belongs and the server of the object.

Finally, the values of the parameters each module expects are defined in the omnet.ini file. Below, follows a more detailed explanation of each configuration file.

### 10.0.1 Definig the topology

There are classes of transactions which are submited for service to their coordinator. The number of classes determines the topology of the network. First of all, for each transaction class there is a QSource module which submits the requests to the coordinator. Also, for each class there should be a connection between the QSource module and the coordinator and a connection should also exist among all the servers participating in the transaction.

To ilustrate the varius aspects of the ned file we will use an example configuration. In this example, there are 9 classes of transactions and two servers. Three of the classes have the first server for coordinator and the other six are coordinated by the second one. Three of the classes are local to their coordinator while the other six are distributed.

It follows that the topology configuration is

```
module TPS
    submodules:
        source: QSource[9];
```

In the previus code section we defined the name of the network and stated that there are nine instances of the QSource module.

Since the first server is coordinator to three transactions, it needs three *in* gates where QSource modules are attached. Also, because it cooperates with the other server for distributed transactions, it needs one *in* gate and one *out* gate in order to exchange messages with it.

Acp stands for the name of the actual protocol. If one wants to simulate Presume Nothing, then Acp should be replaced by PresumeNothing. In general, substitute the Acp token with the actual name of the C++ class implementing the protocol.

Hence the full configuration will be:

```
acp1: Acp;        -- declare the first server
    gatesizes:    -- define the number of gates
        in[3],    -- QSource instances are connected here
        proc[1],  -- Receives messages from the other Acp module
        oproc[1]; -- Sends messages to the other Acp module
```

In the same way, the configuration of the second Acp module should be:

```
acp2: Acp;        -- declare the second server
    gatesizes:    -- define the number of gates
```

```
            in[6],    -- 6 QSource are connected. Cooperdinates six transactions
            proc[1],  -- Receives messages from the other Acp module
            oproc[1]; -- Sends messages to the other Acp module
```

Each Acp module requires a module which simulates the concurrency control and an other one for stable storage managment. In this experiment, the classes of the auxilary modules are LockManager and LogManager who simulate the concurrency and stable storage respectively.

LockManager and LogManager have no arrays of gates so the *gatesizes* section is not needed.

```
lock1: LockManager; -- LockManager of acp1
lock2: LockManager; -- LockManager of acp2
log1: LogManager;   -- LogManager of acp1
log2: LogManager;   -- LogManager of acp2
```

The QSink module receives the completed transactions (either commited or aborted) from the coordinator of the transaction. Hence, each Acp module which coordinates at least one transaction should be connected to the QSink module. When a transaction is aborted and is to be resubmited for processing, QSink forwards the transaction to the appropriate QSource module which in turn will send the transaction to it's coordinator as a new request for service. For this reason, the QSink module is connected to all QSource modules. One more gate is required for the connection of QSink with the statistics module so it can report when a transaction is aborted and will not be resubmited.

```
leave: QSink;
    gatesizes:
        in[2],          -- Acp modules are connected here
        out[9],         -- QSource modules are connected
        out: stat_out; -- Statistics module
```

The statistics module receives messages from all Acp modules and the QSink module. It does not send messages to any other module, hence it needs no *out* gates.

```
stats: Stats;
    gatesizes:
        in[3];  -- Acp modules and QSink send messages to this gates
```

#### 10.0.2 Binding the components

The last part of the topology configuration, defines the connections among the different components. Each *out* gate of a component should be connected to an *in* gate of an other component. If a gate is not connected the simulation will stop with an error message.

The command which binds two gates is:

```
component_a.out_gate -->  component_b.in_gate
```

or if there should be latency in message delivery

```
component_a.out_gate -->  delay [amount of delay] --> component_b.in_gate
```

The varius servers who cooperate in distributed transactions are connected via a network. Thus, the Acp modules should have a delay parameter in their gate connections.

We have already described the connections between the various components of the simulator and in our example, the connections should be:

```
source[0].out --> acp1.in[0]; -- coordinator for transaction 1
source[1].out --> acp1.in[1]; -- coordinator for transaction 2
source[2].out --> acp1.in[2]; -- coordinator for transaction 3

source[3].out --> acp2.in[0]; -- coordinator for transaction 4
source[4].out --> acp2.in[1]; -- coordinator for transaction 5
source[5].out --> acp2.in[2]; -- coordinator for transaction 6

source[6].out --> acp2.in[3]; -- coordinator for transaction 7
source[7].out --> acp2.in[4]; -- coordinator for transaction 8
source[8].out --> acp2.in[5]; -- coordinator for transaction 9

-- Each QSource is connected to the QSink
leave.out[0] --> source[0].in;
leave.out[1] --> source[1].in;
leave.out[2] --> source[2].in;
leave.out[3] --> source[3].in;
leave.out[4] --> source[4].in;
leave.out[5] --> source[5].in;
leave.out[6] --> source[6].in;
leave.out[7] --> source[7].in;
leave.out[8] --> source[8].in;

-- send completed transactions to the QSink
acp1.out --> leave.in[0];
acp2.out --> leave.in[1];

-- The two Acp modules cooperate, so they should be connected
-- The latency is set to 0.06 seconds per message
acp1.oproc[0] --> delay 0.06 --> acp2.proc[0];
acp2.oproc[0] --> delay 0.06 --> acp1.proc[0];

-- Connect the LockManager and LogManager to the
-- Acp module where they belong
acp1.lock_out --> lock1.in;
lock1.out --> acp1.lock_in;
```

```
acp2.lock_out --> lock2.in;
lock2.out --> acp2.lock_in;

acp1.log_out --> log1.in;
log1.out --> acp1.log_in;

acp2.log_out --> log2.in;
log2.out --> acp2.log_in;

acp1.stat_out --> stats.in[0];
acp2.stat_out --> stats.in[1];

-- Finally, connect the QSink module with Statistics module
leave.stat_out --> stats.in[2];
```

### 10.1 Transaction description

Transactions and the objects which they affect are described in an xml file. The first section of the file describes the servers. Each server contains a number of object which in turn have methods. An object has also a size which is the mean of exponential distribution. A method has a name, the duration of the method execution and an attribute which indicates if the method is readonly or affects the state of the object.

Transactions are a list of method calls. The first call defines the coordinator of the transaction. For each call the following atributes are defined: the server where the object resides, the object where the method belongs and finally the method it self.

An example of the xml file follows:

```
<config>
  <node>
    <object statesize="5">
      <method time="0.01"  />
      <method time="0.05"  />
    </object>
    <object statesize="5">
      <method time="0.01"  write="1" />
      <method time="0.01" />
      <method time="0.01" />
    </object>
    ...
  </node>
  <node>
    <object statesize="5">
      <method time="0.05" write="1" />
      <method time="0.05"  />
```

```
    </object>
    ...
  </node>

  <transaction>
    <!-- local read only transaction -->
    <job sid="0" oid="0" mid="0" />
    <job sid="0" oid="1" mid="1" />
    <job sid="0" oid="2" mid="1" />
  </transaction>

  <transaction>
    <!-- distributed read only transaction -->
    <job sid="0" oid="0" mid="0" />
    <job sid="1" oid="1" mid="1" />
    <job sid="0" oid="0" mid="1" />
  </transaction>
  ...
</config>
```

### 10.2 Paramaters of omnet.ini file

There are four sections in the configuration file. The first, named General, defines general attributes of the simulation like total execution time, number of random number generators etc. The second section contains parameters specific for the graphical user interface and the third for the command line interface. If the simulation runs in a graphical mode, the parameters of the second section are used, otherwise the second section is ignored and the third is used instead.

Generally one can run many experiments sequentially. For each run there is a different section which defines experiment specific parameters. The parameters expected for each experiment, are the ones defined in the network topology configuration. Here is an example of parameter definition:

```
*.acp1.xmlId=0
*.acp1.crash=1080
*.acp1.process=2
....
*.source[0].timeout=0.9
*.source[0].mean=0.4
```

acp1 is the name of the module where xmlId is the name of the parameter. One parameter of the Acp module, named objects, accepts as a value an xml document fragment. In order to define the value of the parameter, one should use the xmldoc function which receives two parameters, the path of the xml file and an xpath expression which selects an element. For example

```
*.acp1.objects=xmldoc("config.xml", "/config/node[0]/object")
```

states that the value of the objects parameter for module acp1 is extracted from the uconfig.xml u file after applying the xpath expression above.

# 11 Installation

The development and testing of ACID was done on a Linux operating system. However, the libraries used by the simulator are ported on MS Windows as well. We plan to create a binary for the windows operating system.

The libraries which are required for the simulator are:

- libxml2[1] is used for xml parsing
- boost lambda and boost pool from boost[2] site.
- Standard Template Library[3] This libraries should be already installed in your system
- TCL/TK[4] is required for the GUI of the simulator
- and finally the OMNETPP libraries which can be downloaded from Omnet++[5] site.

For compilation, we use the gnu make utility. Possibly other implementations of make will do as well, but we have not tested this.

A small modification is required for the Makefile. The variable *OMNETPP_DIR* should point at the path where *OMNETPP* files reside at your system. After changing this variable the simulator should compile whithout problems by just issuing the *make* command.

A minor issue which should be kept in mind is that the OMNETPP_DIR/bin should be in the PATH system variable and OMNETPP_DIR/lib should be visible to the linker (for Linux system, modify the LD_LIBRARY_PATH to include the OMNETPP_DIR/lib directory).

In Linux systems, issue this commands from a console:

```
PATH=$PATH:$OMNETPP_DIR/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OMNETPP_DIR/lib
export PATH LD_LIBRARY_PATH
```

[1] http://gnome.org/libxml2
[2] http://boost.org
[3] http://stl.org
[4] http://tcl.org
[5] http://omnetpp.org

# 12 Experiment execution

# 13 Gathering the results

The statistics module, stores the results gathered during the simulation run in a file called "statistics.txt". The file is separated in varius sections defined by a label and the data concerning that section follow. The sections are:

**Time** The total simulated time. The next line contains the value.

**Threads** The next lines show the total time and the mean time a server utililized a specific number of threads. If there a $x$ servers and each has a maximum of $y$ threads then there will be $x * (y + 1)$ lines (time spent in zero threads utilization is recorded as well). The first number is the server concerned, the second the number of threads utililized, the third the total time spent in this number of threads and finally the fourth number records the mean time the server utilized that number of threads.

**Crashes** Records the total time a server was crashed and not responding to messages, and the mean duration of a crash.

**Recovery stats** Time spent recovering from a crash and mean duration of recovery for each server.

**Checkpoint stats** Time spent performing a checkpoint and the mean duration of a checkpoint for each server.

**Transaction recovery** Records the mean time needed to recover the state of the objects affected by an aborted transaction. The first field identifies the server, the second identifies the transaction and the last stores the value.

**Abort times** Same structure as "Transaction recovery" but stores the mean time needed to complete aborted transactions.

**Prepare** Same structure as "Transaction recovery". Stores the mean time required until a server is ready to vote.

**Blocking** Mean of the time a worker was blocked an waiting for coordinator's decision.

**Locks** Locks are acquired on an object which belongs to a server on behalf of a transaction. So the fields are, server, transaction, object and the two last fields store the mean time for a lock acquisition and the number of times a lock was acquired.

**Commit times** Same as "Abort times" only that it stores mean time to commit.

**Aborts** Number of aborts per server and transaction

**Commits** Number of commits per server and transaction

| Message | Sender | Receiver | Description |
|---|---|---|---|
| 2PC Initiation | | | |
| CLIENT_SEND_TRANS | Source | Acp C | Transaction request arrival |
| INIT | Acp C | Acp C | Start processing a trnasaction. Some protocols log this event. |
| Sc TIMEOUT | Acp C | Acp C | Schedule a timeout event |
| LOG_INIT_TR | Acp C | LogManager | Write an INIT log entry. |
| INIT_LOGGED | LogManager | Acp C | An INIT log entry was written. |
| Method invocation (Job) Processing | | | |
| FIRST_JOB | Acp C | Acp W | First job arrived |
| START_JOB | Acp C | Acp W | A job arrived (except the first one) |
| Sch JOB_FINISHED | Acp | Acp | A method returned |
| WJOB_FINISHED | Acp W | Acp C | Coordinator is notified for having finished with a job |
| LOCK_OBJ | Acp | LockManager | Acquire a lock on an object needed for the ongoing transaction |
| LOCK_OBJ | Acp | TimestampConcurrency | Request to use the object |
| LOCK_RELOCK | Acp | LockManager | Acquire the locks needed (like LOCK_OBJ) after the occurrence of a server crash recovery |
| LOCK_RELOCK | Acp | TimestampConcurrency | Similar to LOCK_OBJ only that it occurs after a server crash recovery |
| LOCKED | LockManager | Acp | A lock was acquired (job) |
| TS_OK | TimestampConcurrency | Acp | The right to use the object is granted |
| TS_ABORT | TimestampConcurrency | Acp | Object can not be accessed. The transaction should abort |
| Prepare | | | |
| PREPARE | Acp C | Acp W | Asks the worker to prepare for the voting phase |
| Voting Phase | | | |
| LOG_VOTE | Acp W | LogManager | Log the server's intent to vote |
| VOTE_LOGGED | LogManager | Acp W | Log entry VOTE stored |
| Sc ASK_OUTCOME | Acp W | Acp W | Schedule a query to the coordinator for the transaction outcome |
| VOTED | Acp W | Acp C | Send the vote to the coordinator |
| VOTE_NO | Acp W | Acp C | Veto against commit |
| VOTE_LOGGED | LogManager | Acp | A log entry is written recording the intention to vote |
| Commit Phase | | | |
| COMMIT | Acp C | Acp W | Send commit decision to the transaction worker |
| Sch RESEND_COMMIT | Acp C | Acp C | Start a timer in order to resend the decision in case not all workers have aknowledged the commit decision |
| LOG_COMMIT | Acp | LogManager | Log in simulated stable storage the commit event. Used in PRN, PRA. |
| COMMIT_LOGGED | LogManager | Acp | Commit log entry stored. Used in PRN, PRA. |
| LOG_SAVE_STATE | Acp | LogManager | Ask to save the state in stable storage |
| Abort Phase | | | |
| ABORT | Acp C | Acp W | Send abort decision to the transaction worker |
| Sch RESEND_ABORT | Acp C | Acp C | Start a timer in order to resend the decision in case not all workers have aknowledged the abort decision |
| LOG_ABORT | Acp | LogManager | Log in simulated stable storage the transaction abort |
| LOG_RECOVER_STATE | Acp | LogManager | Recover the object states of the last committed transaction |
| LOCK_STOP | Acp | LockManager | Ask the lock manager to stop acquiring locks |
| ABORT_LOGGED | LogManager | Acp | Abort log entry stored |
| LOG_RECOVERED | LogManager | Acp | Object states recovered |
| Transaction finilazation | | | |
| AKNOWLEDGED | Acp W | Acp C | The worker aknowledges the coordinator's decision |
| OUTCOME_ASKED | Acp W | Acp C | The worker asks for a transaction outcome |
| LOG_END | Acp C | LogManager | Place a transaction end mark in the log |
| END_LOGGED | LogManager | Acp C | Log manager notifies that the entry is recorded |
| LOCK_UNLOCK | Acp | LockManager | Release the locks acquired for a transaction |
| Checkpointng related messages | | | |
| Sc CHECKPOINT_MSG | Acp | Acp | Schedule the occurrence of a checkpoint |
| LOG_START_CHECKPOINT | Acp | LogManager | Initiate a checkpoint |
| LOG_FINISHED_CHECKPOINT | LogManager | Acp | Checkpoint accomplished |
| Server crash | | | |
| LOG_RESET | Acp | LogManager | A server crash failure has occurred. Clear the I/O queue |
| LOCK_RESET | Acp | LockManager | A server crash failure has occurred. Release all locks |
| Server crash recovery | | | |
| LOG_COLLECT_RECOVERY_INFO | Acp | LogManager | Start gathering information for server recovery |
| LOG_RECOVERY_INFO_COLLECTED | LogManager | Acp | Information needed for crash recovery was gathered. |
| CRASH_ABORT | Acp C | Acp C | Abort a transaction after a server crash. No object revovery is needed. |
| Statistics | | | |
| STAT_FINISHED | Acp | Stats | Inform that a transaction processing is over and send statistics gathered during it's execution |
| STAT_REJECTED | Acp C | Stats | A transaction is rejected because the coordinator is crashed |
| SYS_THREAD_COUNT_INC | Acp | Stats | A new thread is used |
| SYS_THREAD_COUNT_DEC | Acp | Stats | A thread is no longer needed and is released |
| SYS_CRASHED | Acp | Stats | Server crashed |
| SYS_RECOVERY_STARTED | Acp | Stats | Server started the recovery procedure |
| SYS_RECOVERED | Acp | Stats | Server has recovered from a system failure |
| SYS_CHECKPOINT_STARTED | Acp | Stats | Checkpoint has begun |
| SYS_CHECKPOINT_ENDED | Acp | Stats | Checkpoint is completed |
| Auxiliary messages | | | |
| LOCK_INIT | Acp | LockManager | Send information about the server. The information are used for debugging messages |
| LOG_INIT | Acp | LogManager | Send information about the server. The information are used for debugging messages |
| Other messages | | | |
| Sch CHECKPOINT_CONTINUE | LogManager | LogManager | Create a new log read request for an ongoing checkpoint |
| Sch I/O | LogManager | LogManager | Schedule an I/O event for the first request (read/write) in the simulated I/O queue. |

Table 3: Messages exchanged between ACID Sim Tools modules

| Role | State | Description |
|---|---|---|
| Both | ST_ABORT | Wait for the ABORT log entry to be stored |
| Coordinator | ST_AKN_ABORT | Wait for abort acknowledgment from the workers |
| Coordinator | ST_AKN_COMMIT | Wait for commit acknowledgment from the workers |
| Both | ST_COMMIT_LOG | Wait for the COMMIT log entry to be stored |
| Worker | ST_DEC | Expect the decision of the coordinator |
| Coordinator | ST_FABORT | The transaction is aborted due to recovery from failure |
| Coordinator | ST_INIT | Transaction processing just started at the coordinator's site |
| Both | ST_JOB | Wait for a job to complete |
| Both | ST_LOCK | Wait until a lock is acquired |
| Both | ST_LOG_END | The "Finish" log entry is being stored |
| Both | ST_RECOVER | Recovering the objects state after an abort |
| Worker | ST_VOTE | Prepare to vote for the transaction |
| Coordinator | ST_VOTES | Collecting votes from the workers |
| Both | ST_WAIT | Wait for the next job or enter the voting phase |
| Coordinator | ST_WAK | Abort and acknowledgment of the decision are pending |
| Coordinator | ST_WAKR | Abort, recovery and acknowledgment of the decision are pending |
| Worker | ST_WAR | Abort and recovery are pending |
| Coordinator | ST_WKR | Acknowledgment and recovery are pending |
| Both | ST_WRE | Recovery and the "Finish" log entry are pending |

Table 4: States of *FSM* for worker and coordinator