ACID Sim Tools: A Simulation Framework for Distributed Transaction Processing Architectures

Anakreon Mentis
Department of Informatics
Aristotle Un. of Thessaloniki
54124 Thessaloniki, Greece
tel. +30 2310 998236

anakreon@csd.auth.gr

Panagiotis Katsaros
Department of Informatics
Aristotle Un. of Thessaloniki
54124 Thessaloniki, Greece
tel. +30 2310 998532

katsaros@csd.auth.gr

Lefteris Angelis
Department of Informatics
Aristotle Un. of Thessaloniki
54124 Thessaloniki, Greece
tel. +30 2310 998230

lef@csd.auth.gr

ABSTRACT

Modern network centric information systems implement highly distributed architectures that usually include multiple application servers. Application design is mainly based on the fundamental object-oriented principles and the adopted architecture matches the logical decomposition of applications (into several tiers like presentation, logic and data) to their software and hardware structuring. The provided recovery solutions ensure an at-most-once service request processing by an existing transaction processing infrastructure. However, in published works performance evaluation of transaction processing aspects is focused on the computational model of database servers. Also, there are no available tools which enable exploring the performance and availability trade-offs that arise when applying different combinations of concurrency control, atomic commit and recovery protocols. This paper introduces ACID Sim Tools, a publicly available tool and at the same time an open source framework for interactive and batch-mode simulation of transaction processing architectures that adopt the basic assumptions of an object-based computational model.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Design studies, Fault tolerance, Reliability availability and serviceability*. D.4.5 [**Operating Systems**]: Reliability – *Checkpoint/restart, Fault-tolerance*. D.2.12 [**Software Engineering**]: Interoperability – *Distributed Objects*. H.2.4 [**Database Management**]: Systems – *Transaction Processing*.

General Terms

Performance, Design, Experimentation, Measurement.

Keywords

Fault Tolerance, Performance Evaluation, Transaction Processing, Simulation, Atomic Commit, Concurrency Control, Recovery

1. INTRODUCTION

Distributed transaction processing has advanced considerably as an important research subject in the field of database systems. Published performance evaluation results provide comparisons between alternative concurrency control protocols ([1]) or between different atomic commit and recovery protocols ([2]), when applied in centralized or distributed database systems.

The development of ACID Sim Tools was motivated by the following two facts:

- There is no simulation tool, which can provide insight into the collective effects, the interaction effects and the performance and availability trade-offs that arise when applying different combinations of concurrency control, atomic commit and recovery protocols (and protocol parameters).

- The modern trend of partitioning applications into several tiers (presentation, logic and data), the multiplicity of the components and their interdependencies create a new computational environment ([3], [4]). This new environment is no more related to the classic computational model of database servers, where transaction processing is studied in the frame of the so-called page model - that refers to the way in which data pages are read or written at the storage layer of a database system. As a response to this need our toolset (and framework) adopts a minimal set of assumptions that resemble an object-based computational model like for example the so-called OMG Core Object Model ([6]). The OMG OTS standard ([5]) and the Enterprise Java Beans are two well-known cases of transaction processing reference architectures that are build on top of object-based computational models.

ACID Sim Tools implements appropriate abstractions, which allow one to reuse existing functionality for the development of new protocols (beyond the ones that are already implemented). It is also possible for the potential user to alter the basic model of flat transactions (to implement e.g. nested transactions or advanced transaction models [7], [8], [9] with different notions of *serializability*) or to develop models of transactional replication schemes (as for example the one described in [10]).

The analyst debugs the modeled transaction processing architecture by interactive simulation through some basic animation functions (provided by the OMNeT++ based graphical environment [11], [12]). Key events are monitored and the accumulated data is used to calculate performance and availability metrics. The obtained metrics can be used in suitable analyses, which aim to evaluate architectural design options by quantifying their collective effects, as well as their interaction effects in a meaningful way. It is thus possible to compare alternative transaction processing architectures with respect to a given set of performance/availability goals. Similar performance design methods have been already proposed for object replication based fault tolerance architectures ([13]) and independent checkpointing processes for application recovery ([14], [15]).

We provide simulation results for a sample transactional processing architecture, under a given synthetic workload scenario. The presented results reveal some of the most influential performance and availability trade-offs to be taken into account, when a transactional architecture has to fulfill a given set of performance design goals.

Section 2 introduces basic definitions and presents the problem statement. Section 3 provides a description of the framework design. Section 4 introduces the considered synthetic workload scenario and comments the obtained simulation results. The paper concludes with a brief discussion regarding the tool's utility and its future development prospects.

2. BASIC DEFINITIONS AND PROBLEM STATEMENT

2.1 Transactional Objects
In our reference architecture clients send requests to application servers that maintain a repository of encapsulated objects. Application servers constitute the surrounding runtime system of the invoked objects: they manage them by spawning execution threads on behalf of client requests, monitoring executions, managing communication connections, handling generic forms of exceptions and so on. The implementation of the managed objects may itself invoke methods on other objects and issue requests to other servers.

We adopt the common simplification ([16]) that the object is both the unit of operation as well as the unit of disk persistence. Thus, each *transactional object* o_i owns a persistent state: state data live beyond request/reply boundaries and user sessions, for an indefinite time period. The provided methods, $op_l^{o_i}$, $1 \leq l \leq$ #(methods of o_i)[1], is the only way to change an object's state and are invoked by *synchronous messages*: the object requiring the execution of the invoked method (requester) stops executing and waits for the invoked execution to terminate and the reply to return. Upon reception of the reply, the sender resumes.

Each *message msg* is defined as a pair

$$(method, type), \text{ with } type \in \{read, write\}$$

such that

$$method \in \bigcup_i \left\{ op_l^{o_i} \mid 1 \leq l \leq \#(\text{methods of } o_i) \right\}$$

and *read*, *write* indicate whether *method* is read-only or respectively modifies the state of some o_i.

A message sequence specification for $op_l^{o_i}$, $1 \leq l \leq$ #(methods of o_i) is defined as a total order relation $\Rightarrow$ over the set

$$MsgSq(op_l^{o_i}) = \{msg_s \mid 1 \leq s \leq \#(\text{msgs in } MsgSq(op_l^{o_i}))\}$$

of method invocations generated by $op_l^{o_i}$. For every two msg_s, $msg_t \in MsgSeq(op_l^{o_i})$, $msg_s \Rightarrow msg_t$ if and only if msg_s is executed before msg_t.

A transaction is essentially a program execution that reads and/or modifies one or more transactional objects, which are managed by one or more application servers. A transaction comes with certain system-guaranteed properties that simplify the development of distributed applications in that the application programs themselves can safely ignore the complexity of the overall system regarding:

- All effects that may result from concurrent program executions and especially transactional object accesses (ACID Sim Tools currently implements only flat transactions and does not provide support for nested transactions that allow for concurrent atomic execution of operations included in the same transaction).

- All effects that would result from program executions being interrupted because of process or computer failures.

All necessary steps to cope with concurrency and failures are delegated to the runtime system of the underlying transactional servers. Thus, a transaction takes the form of a set of operations $op_l^{o_i}$ that are triggered by a client request and are executed on one or more servers, with the ACID properties (*atomicity*, *consistency*, *isolation*, *durability*) guaranteed by the runtime system of the involved servers. The invoked operations are specified by their associated MsgSq($op_l^{o_i}$) and incur specific CPU and I/O resource consumption requirements. CPU resource consumption is possible to be quantified by appropriate static analysis tools like for example Heptane ([17]).

Having adopted that a transactional object is both the unit of operation as well as the unit of disk persistence, we neglect modeling the existence of a cache, which is usually used in application servers with a large number of transactional objects. This option would complicate the simulated recovery mechanism and would burden the developed simulation models with additional (system specific) buffer management parameters, whose performance implications are not related to the aims of the introduced toolset.

2.2 Concurrency Control and Recovery Control
To assure the required ACID properties an application server includes the following components:

---

[1] #(methods of o_i) denotes the number of methods for o_i

- The *concurrency control component* that guarantees the isolation properties of transactions, for both committed and aborted transactions and

- The *recovery component* that guarantees the atomicity and durability of transactions.

The concurrency control component uses an algorithm[2] like for example aggressive locking, two-phase locking (2PL), timestamp ordering, serialization graph testing, tree locking or an optimistic algorithm that essentially lets newly arriving method invocations simply pass, but validates their output occasionally. The widely used locking protocols require transactions to wait when requested locks cannot be granted immediately. A set of transactions, each holding some locks and requesting an additional one, may end up being mutually blocked. Such cyclic wait situations are commonly known as *deadlocks*. A protocol may either prevent or allow deadlocks. In the second case *deadlock detection and resolution* is performed either in a continuous or in a periodic manner or alternatively, the system employs the use of a *timeout strategy*. In the latter case, the system maintains a timer that is activated upon initiation of each transaction and when the timer expires the transaction is aborted.

The recovery component implements a *distributed transaction coordination protocol* ([18]), like for example the (basic) two-phase commit (2PC) or alternatively one of its optimized variants or the non-blocking three-phase commit protocol (3PC). The server that initiates the transaction is called *coordinator* and the involved servers are called transaction *participants* (*workers*). We note that the same server may be at the same time the coordinator for the transactions issued by itself and a worker for the transactions that involve it.

The recovery component performs *transaction rollbacks* in case of transaction aborts and *crash recovery* in case of server failures. A transaction is aborted upon detecting inconsistencies in an object's state (*surprise aborts*) that would arise from the current transaction's further execution. Also a transaction is aborted when the server "kills" it for internal reasons (e.g. deadlock or overload situations). In these cases, the server continues to operate and is expected to restrict its recovery measures to the actually affected transaction while processing other transactions as usual. The case of server failures is discussed in more detail in the sequel.

Crash recovery, transactions atomicity and durability are assured by the maintenance of a *stable log* in permanent storage[3]. The log stores object state updates and bookkeeping records about the system history and survives non-catastrophic server failures that leave all data on secondary storage intact. It is an explicit representation of the necessary actions for rolling back uncommitted transactions by undoing each transaction's prior updates. We consider that application servers perform *physical logging* of *after-images*, which means that the log stores snapshots of the state of transactional objects after having invoked an operation. Thus, log I/Os depend on the object's *state size* and on the disk performance characteristics of the server.

We assume that failures are characterized by the *fail-stop property* ([19]), in the sense that the server is indeed brought down immediately after detecting an error. Although this assumption is only an idealized behavior, as error detection will not be perfect, it is sufficiently well approximated by intensive self-checking.

While performing crash recovery for a server, the server and its objects are unavailable to the clients. Therefore, *minimizing the recovery cost* is an important performance goal. Over time, the stable log collects a large number of log entries, some of which, we can infer, are no longer relevant regardless of when and how exactly the server crashes. All log entries that are no longer needed, can be periodically removed from the stable log. A periodically invoked log truncation along with saving all uncommitted object updates to the stable log is known as a *checkpoint* and incurs a substantial amount of additional workload during normal operation. Checkpointing results in a new log file through the following steps: (i) appends a mark to the used log file (1 write operation), (ii) copies the after-images of all transactional objects as they are recorded in the old log by the latest committed transactions that affected them (2 read operations and 1 write for each server object) and (iii) copies to the new log file all data appended to the old log after the mark (1 read and 1 write for each log entry). When the checkpoint is complete, the old log becomes obsolete and is then garbage collected.

Rather than minimizing the recovery cost, *a trade-off performance goal is minimizing the additional resource consumption that is required during normal operation of the server to make crash recovery* work when the server fails. Checkpointing incurs additional I/O costs, but does not block transaction processing that takes place in parallel. However, apart from checkpointing costs, another *major overhead arises from forced and thus synchronous I/O on the stable log*. If the overhead caused by checkpointing and by forced logging transactional object updates and protocol messages, becomes too high, then it could adversely affect the performance of object method invocations during normal operation. Architectural solutions often adopt the attractive optimization of batching log I/Os. Trading multiple short I/Os for a single long I/O yields a high benefit because it avoids rotational delays and can therefore utilize the disk bandwidth much better.

Lock contention is another source of resource contention with the important feature of being susceptible to *thrashing phenomena*. The reason for lock contention thrashing is that with increasing concurrency, the probability of a lock request not being granted and the duration for which a transaction needs to wait upon a lock conflict increase superlinearly. When too many transactions run concurrently, we may end up with *a situation where most transactions are blocked because of lock conflicts and only a few transactions are still running*. In this situation, frequent deadlocks are an additive danger. To avoid performance disasters of the above kind, transactional servers limit their *multiprogramming level* (MPL) that represents the maximum number of transactions that are allowed to run concurrently. When this limit is reached,

---

[2] As in all published performance evaluation studies, CPU resource requirements for the mentioned concurrency control algorithms cannot be easily quantified. Thus, we adopt the common approach of not considering resource consumption for concurrency control.

[3] The resource requirements for the applied atomic commit protocol consist of the additional I/O costs for force-writing the required transaction status log entries and the network latency costs when application servers communicate over the network.

newly arriving transactions are held in a *transaction admission queue*.

2.3 Performance – availability considerations

For the described computational setting typical performance requirements are:

- To provide high throughput, that is, a satisfactory number of successfully processed (committed) transactions per time unit.

- To result in short response times, where each transaction's response time is defined as the time span between issuing the transaction and its successful completion as perceived by the client (commit).

Throughput and response times depend on many issues in the implementation, configuration and tuning of application servers. In addition to the high throughput, a trade-off goal is the provision for high availability, which implies that recovery times after failures are short.

We believe that there is a need for a performance engineering perspective and the proposed versatile toolset can support suitable methods that contribute to a systematic performance design approach. More specifically we need to be able

- to provide insight and compare the collective effects, the interaction effects and the performance and availability trade-offs that arise when applying different combinations of concurrency control, atomic commit and recovery protocols and

- to estimate optimal values for multiprogramming levels, transaction timeouts and checkpoint intervals for the servers of a transactional architecture, with respect to a set of given performance goals.

The first aim requires appropriate methods ([20]) that quantify the significance and the relationships of the considered design factors in synthetic workloads with different degrees of distribution (localized to highly distributed transactions), different mixes of read-only and update transactions and different conditions of resource contention (I/O bound or CPU bound system configurations). The second aim will allow tuning distributed transaction processing, in order to maximize transactions throughput up to the servers' thrashing levels and in trading off the resulted resource consumption during normal processing against the achieved speed of recovery.

3. THE ACID SIM TOOLS DESIGN

3.1 A message passing based simulation engine

ACID Sim Tools uses the Objective Modular Network Test-bed (OMNeT++) simulation engine. OMNeT++ is an open-source, component-based and modular framework written in C++. An ACID Sim Tools model inherits the structure of an OMNeT++ simulation model, which in general is defined as a collection of modules.

In an ACID Sim Tools model we include module instances of the following four (4) module types:

- the *Source module* that generates transaction request arrivals;

- the *Atomic Commit Processing (Acp) module* that implements the employed distributed transaction coordination protocol;

- the *Lock Manager module* that provides concurrency control functionality;

- the *Log Manager module* that implements logging in permanent storage.

A new protocol variant for e.g. atomic commit processing is created, by extending the class hierarchy of an existing Acp module instance. This extension results in a new Acp module instance, which can be used in future ACID Sim Tools models.

Modules communicate by exchanging messages and for this reason they are loosely coupled. In effect, the analyst can easily substitute a module instance with a different instance of the same type.

Each message may be a complex data structure. In the course of a simulation run messages are scheduled to occur or they may be exchanged or even canceled.

The ACID Sim Tools architecture:

- Allows the design of modular simulation models, which can be combined and reused in a flexible way.

- Adopts the object-oriented design of OMNeT++ and this allows flexible extension of the currently implemented ACID Sim Tools modules.

- Provides an extensive simulation library that includes support for statistics, data collection, graphical presentation of simulation data, random number generators and a range of data structures.

In ACID Sim Tools, model components are linked with the simulation library, and one of the user interface libraries to form an executable program. One user interface library is optimized for command line and batch-oriented execution, while the other employs a graphical user interface (GUI) that can be used to trace and debug the simulation.

The OMNeT++ based graphical environment utilizes two main windows (Figure 1). The left one provides an *animated view* of the ongoing simulation experiment.

The right window displays the so-called *inspection view* with the current timeline status and simulated time, as well as a tree view of the model parameters and a scrolling log of past events. The icons shown in the animated view represent the model's modules and the moving particles represent sent messages. Regarding the events shown in the timeline of the inspection view, they represent already scheduled messages.

In the shown animation snapshot icons `acp1` and `acp2` represent two transactional servers, the lock icons represent the corresponding lock manager module instances and icons `log1` and `log2` show the log manager module instances for the considered architecture.
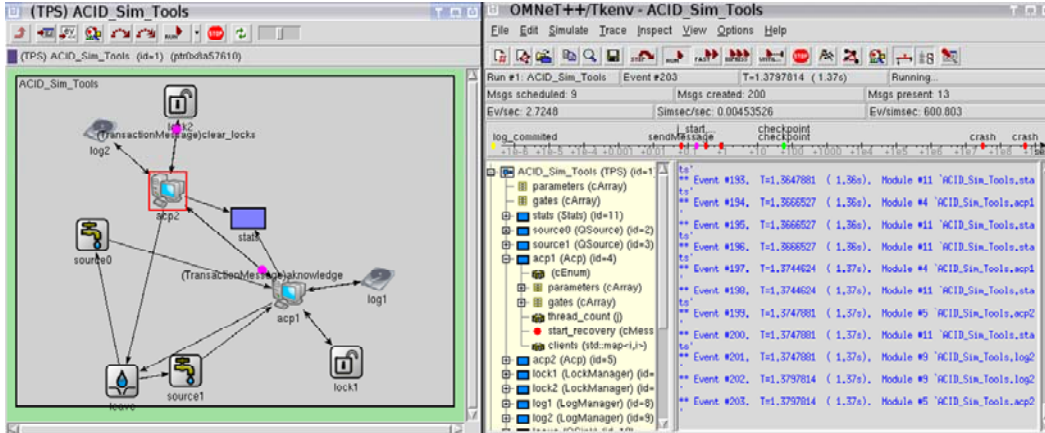
Figure 1. The ACID Sim Tools GUI and the interactive simulation and inspection functions

The source module instances represent the transaction request arrival processes. The *statistics module* instance `stat` receives messages from all other modules, when statistically important events occur. The accumulated event-monitoring information is stored in a file at the end of the simulation run, for further simulation output analysis.

For a detailed description of the provided GUI functionality refer to [11] and [12].

Module instances of a simulation model have a number of parameters (e.g. transactions' request interarrival rates, servers' multiprogramming levels) that altogether are specified in an appropriate initialization file. Furthermore, the model's structure is given in an input XML file that defines the distribution of the transactional objects to the available servers, their methods together with a read/write characterization and the specifications of the considered transaction classes. A more thorough description of the ACID Sim Tools model parameters is provided in the case study introduced in Section 4.

Batch-mode simulation from the command prompt is used for computing certain performance and availability metrics derived from one or more simulation runs. Simulation output from multiple runs is analyzed by employing the method of independent replications, where the number of simulation runs is determined dynamically, based on the targeted level of accuracy (the confidence interval half width threshold defined as a percentage of the computed metric).

3.2 Simulation modules for distributed transaction processing

The provided Source module generates transaction requests that obey the Poisson distribution for a given mean inter-arrival rate that is passed as a module parameter. In an ACID Sim Tools model we consider different classes of transaction requests (corresponding to different message sequence specifications). Each transaction class is generated by a given Source module instance and the generated requests are delivered to the transaction coordinator, which is represented by the Acp module instance corresponding to the server of the first transactional object invocation.

User defined Source module variants may focus on the implementation of alternative request arrival distributions or even

on the delivery of request arrival traces obtained from a real transactional server.

In a given transaction class, a particular Acp module instance plays either the role of the coordinator or the role of a worker. We currently provide implementations of three variants of the two-phase commit (2PC) protocol: (i) the basic 2PC, also called 2PC PResume Nothing (PRN), (ii) the 2PC PResume Commit variant (PRC) and (iii) the 2PC PResume Abort variant (PRA). Figure 2 shows the class hierarchy of the currently implemented Acp module type. PRC and PRA module instances differ from a PRN instance in the way they handle the transaction commit or the transaction abort case. Our experience in extending the Acp module hierarchy suggests that the effort required to implement a new 2PC variant is negligible.
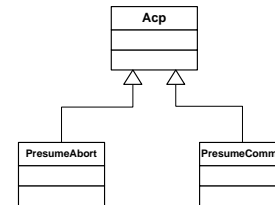

Figure 2. The Acp module class hierarchy

The Log Manager module type simulates storage of forced and non-forced log entries, applies the simulated checkpointing policy and gathers the information required in transaction rollbacks and server crash recovery. In effect, a Log Manager module generates read and write service requests for an I/O queue resource. Service demands are calculated based on the type of the simulated operation (forced or non-forced log entry, checkpointing, read request etc.) and the I/O costs assumed for the primitive read and write operations. Due to the differences in the information expected during recovery in the implemented 2PC variants, we provide an appropriate module instance for each particular 2PC variant (Figure 3).

The LockManager module type is responsible for handling requests for lock acquisition and release. Current module implementation simulates the behavior of the basic two-phase locking (2PL) scheme, but ACID Sim Tools users can always replace it with an alternative concurrency control locking-based module variant.
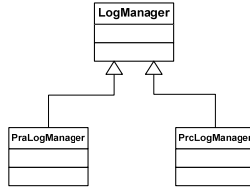
Figure 3. The LogManager module class hierarchy

3.3 Communicated messages for Concurrency Control and Recovery

Table 1 summarizes the messages exchanged between the ACID Sim Tools module types in the course of a simulation run. When a message assumes specific roles for the sender and/or receiver module instances we denote with Acp C the Acp module instance that corresponds to the coordinator of the concerned transaction class and with Acp W an Acp module instance that corresponds to some worker. Scheduled events correspond to messages sent by a module instance to itself.

The described messages are used in transaction processing protocol implementations within new ACID Sim Tools module instances. Figure 4 introduces the ACID Sim Tools message exchange implementing the transaction commit processing in PRN. However, due to space limitations we omit the transaction abort message sequences for PRN, PRC and PRA.

Table 1: Messages exchanged between ACID Sim Tools modules

| Message	Sender	Receiver	Description
2PC Initiation			
CLIENT_SEND_TRANS	*Source*	*Acp C*	Transaction request arrival
Sc TIMEOUT	*Acp C*	*Acp C*	Schedule a timeout event
LOG_INIT_TR	*Acp C*	*LogManager*	Write an "INIT" log entry. Used in **PRN, PRA**.
INIT_LOGGED	*LogManager*	*Acp C*	An "INIT" log entry was written. Used in **PRN, PRA**.
Method invocation (Job) Processing			
NEXT_JOB	*Acp C*	*Acp W*	Transaction worker is requested to execute some job
WORKER_JOB_FINISHED	*Acp W*	*Acp C*	Coordinator is notified for having finished with a job
LOCK_OBJ	*Acp*	*LockManager*	Acquire the locks needed for the ongoing transaction
RELOCK	*Acp*	*LockManager*	Acquire the locks needed (like LOCK_OBJ) after the occurrence of a server crash recovery
OBJ_LOCKED	*LockManager*	*Acp*	A lock was acquired
Sc JOB_ENDED	*Acp*	*Acp*	Schedule the execution of an object method invocation (job)
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
LOG_VOTED	*LogManager*	*Acp W*	Log entry VOTE stored
Sc ASK_OUTCOME	*Acp W*	*Acp W*	Schedule a query to the coordinator for the transaction outcome
VOTE	*Acp W*	*Acp C*	Send the vote to the coordinator
Commit Phase			
COMMIT_TRANSACTION	*Acp C*	*Acp W*	Send commit decision to the transaction worker
COMMIT_LOG	*Acp C*	*LogManager*	Log in simulated stable storage the commit event. Used in **PRN, PRA**.
LOG_COMMITED	*LogManager*	*Acp C*	Commit log entry stored. Used in **PRN, PRA**.
ACKOWLEDGE_COMMIT	*Acp W*	*Acp C*	A worker acknowledges the commit decision. Used in **PRN, PRA**.
Abort Phase			
ABORT_TRANSACTION	*Acp C*	*Acp W*	Send abort decision to the transaction worker
LOG_ABORT	*Acp*	*LogManager*	Log in simulated stable storage the transaction abort
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
LOG_ABORTED	*LogManager*	*Acp*	Abort log entry stored
LOG_OBJ_RECOVERED	*LogManager*	*Acp*	Object states recovered
ACKNOWLEDGE_ABORT	*Acp W*	*Acp C*	A worker acknowledges the abort decision. Used in **PRN, PRC**.
Auxiliary messages			
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
UNLOC_OBJ	*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
Other messages related to the simulated LogManager function			
Sc CHECKPOINT_CONTINUE	*LogManager*	*LogManager*	Create a new log read request for an ongoing checkpoint
CLEAR_LOG	*Acp*	*LogManager*	A server crash failure has occurred. Clear the I/O queue and stop all I/O operations.
INFO_READ	*LogManager*	*Acp*	Information needed for crash recovery was gathered.
LOG_APPEND_ABORT	*Acp*	*LogManager*	Log in simulated stable storage the transaction abort (as the LOG_ABORT message), without replying with a LOG_ABORTED message.
LOG_PREPARE	*Acp*	*LogManager*	Store log entry PREPARE
READ_INFO	*Acp*	*LogManager*	Start gathering information for server recovery
Sc RECOVER_PHASE	*LogManager*	*LogManager*	Create a new log read request, in order to recover the state after the last committed transaction.
Sc I/O	*LogManager*	*LogManager*	Schedule an I/O event for the first request (read/write) in the simulated I/O queue.
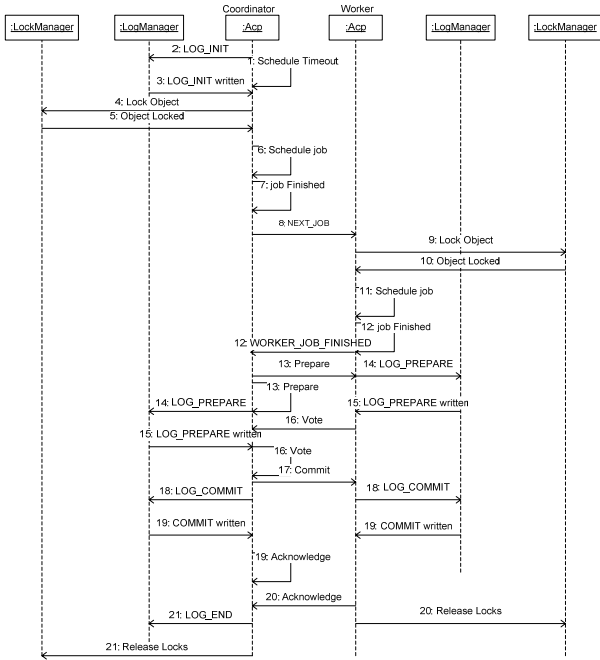
Figure 4. Transaction commit processing in PRN

4. ACID SIM TOOLS RESULTS FOR A SYNTHETIC WORKLOAD SCENARIO

Current section provides sample results from a simulated synthetic workload scenario. We comment on the results obtained for the tested architecture parameters, but we do not present a complete systematic experiment and the subsequent tradeoff analyses, since the basic aim of this article is to present the toolset.

The considered transactional objects are distributed over the two servers of the modeled transaction processing architecture as shown in Table 2.

Table 2: Transactional objects

Server	Object	State Size (Kb) - exp. -	Method Name	CPU Resource Consumption (sec) - exp. -	READ/WRITE
acp1	obj1	5	meth11	0.01	READ ONLY
			meth12	0.05	READ ONLY
	obj2	5	meth21	0.01	WRITE
			meth22	0.01	READ ONLY
			meth23	0.01	READ ONLY
	obj3	5	meth31	0.04	WRITE
			meth32	0.01	READ ONLY
	obj4	5	meth41	0.01	WRITE
			meth42	0.01	READ ONLY
			meth43	0.01	READ ONLY
	obj5	5	meth51	0.01	WRITE
			meth52	0.01	READ ONLY
			meth53	0.01	READ ONLY
acp2	obj6	5	meth61	0.05	WRITE
			meth62	0.05	READ ONLY
	obj7	5	meth71	0.05	WRITE
			meth72	0.01	READ ONLY
	obj8	5	meth81	0.01	READ ONLY
			meth82	0.01	READ ONLY
	obj9	5	meth91	0.05	WRITE
			meth92	0.05	READ ONLY
	obj10	5	metha1	0.05	WRITE
			metha2	0.05	READ ONLY
	obj11	5	methb1	0.01	READ ONLY
			methb2	0.01	READ ONLY
	obj12	5	methc1	0.05	WRITE
			methc2	0.05	READ ONLY
	obj13	5	methd1	0.05	WRITE
			methd2	0.05	READ ONLY

The assumed transaction classes and their associated parameters are introduced in Table 3 and the employed system parameters are summarized in Table 4. We considered fail-stop server failures with mean interarrival time 21600 sec.

Concurrency control adheres to the widely used 2PL scheme. Regarding the atomic commit processing, in our experiments we employed the PRN and PRC protocols. We assume exponentially distributed parameters and we provide the corresponding means for all object state sizes, for the CPU resource demands and for the used transaction interrarival time that utilizes the studied architecture in a considerable degree.

We conducted simulation experiments for two different transaction timeout cases and three different checkpoint interval parameters, where the latter were the same for both servers.

The obtained results are summarized in Figures 5 and 6. They depict how model parameters affect the ratio of committed distributed transactions, as well as their mean response times.

Figure 5 shows that by reducing the checkpoint interval below a certain level - in order to minimize recovery costs - the incurred overhead results in worse ratio of committed distributed transactions without reducing the corresponding mean response time (Figure 6).

As expected, the PRC protocol improves the mean response times of all distributed transactions. Local transactions are not shown, because their performance is not affected significantly by the changes to the studied model parameters. Finally, the smaller timeout parameter improved the observed mean response time, but at the same time increased the number of aborted transactions resulting in a lower ratio of committed distributed transactions.

5. CONCLUSION

This article presented a publicly available interactive and batch-mode simulation tool (and framework) that provides insight into the most influential performance and availability tradeoffs that arise in distributed transaction processing architectures. Detailed documentation for the basic simulation algorithm and source code is available in the ACID Sim Tools web site [21].

We believe that the proposed toolset could support suitable analyses that will contribute to a systematic performance design approach for distributed transaction processing architectures.

Table 3: Transaction classes

Transaction Class	Methods invoked	Mean interarrival times (sec) - exp. -	Timeouts (sec) (experiment 1)	Timeouts (sec) (experiment 2)
tr1	meth11, meth22, meth32	0.6	0.9	0.7
tr2	meth11, meth72, meth12	0.6	0.9	0.7
tr3	meth12, meth61, meth21	0.6	0.9	0.7
tr4	meth92, metha2, meth82	0.6	0.9	0.7
tr5	meth92, meth42, meth92	0.6	0.9	0.7
tr6	meth92, meth41, metha1	0.6	0.9	0.7
tr7	methb2, methc2, methd2	0.6	0.9	0.7
tr8	methb2, meth52, methb2	0.6	0.9	0.7
tr9	methb2, meth51, methc1	0.6	0.9	0.7

Table 4: System parameters

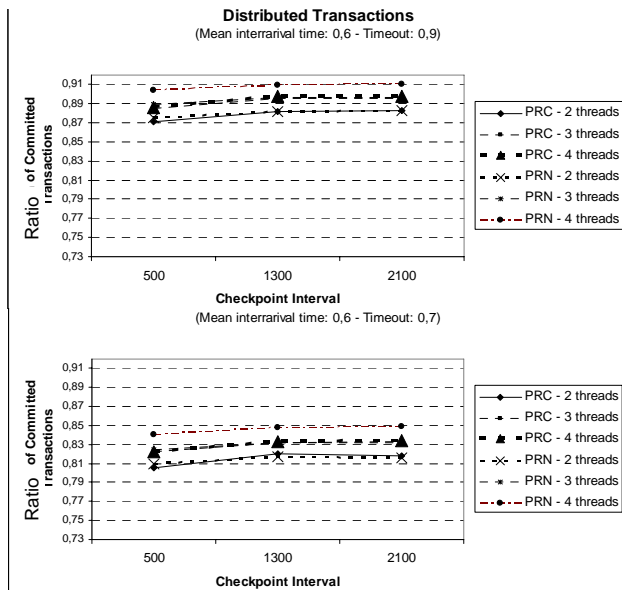| Network Latency / message:	0.06 sec			
Server	Disk Read Latency	Disk Write Latency	Multiprogramming Level (MPL)	Checkpoint intervals (sec) Periodic
acp1	4.271e-5 sec/Kb	51.252e-5 sec/kb	2, 3, 4	500 sec, 1300 sec, 2100 sec
acp2	4.271e-5 sec/Kb	51.252e-5 sec/kb	2, 3, 4	500 sec, 1300 sec, 2100 sec

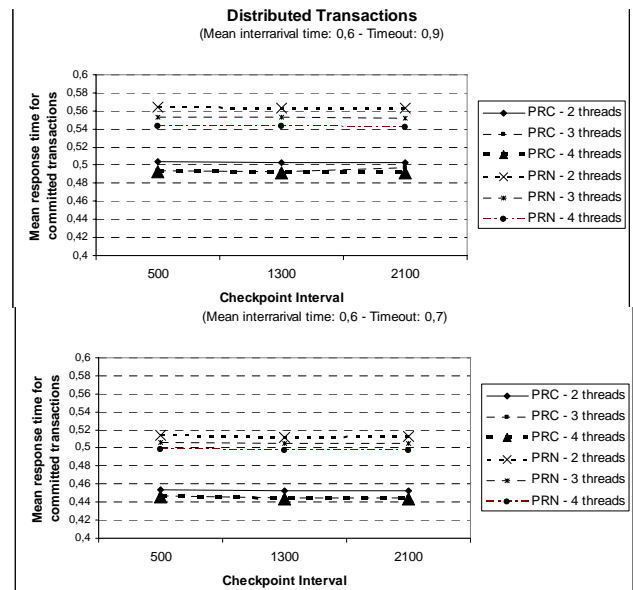**Figure 5. Ratio of distributed transactions that commit**

Figure 6. Mean response times for committed dist. transactions

6. REFERENCES

[1] Agrawal, R., Carey, M., and Livny, M. 1987. Concurrency control performance modeling: Alternatives and implications. ACM Trans. on Database Systems 12, 4, 609-654.

[2] Chrysanthis, P. K., Samaras, G. and Al-Houmaily, Y. J. 1998. Recovery and performance of atomic commit processing in distributed database systems. In Recovery Mechanisms in Database Systems, V. Kumar and M. Hsu, Ed. Prentice-Hall. 370-416.

[3] Wheater, S. M. and Shrivastava, S. K. 1997. A framework for configurable distributed transactions, In Proc. 7th High Performance Transaction Systems Workshop (California, USA, 1997).

[4] Weikum, G. and Vossen, G. 2002. Transactional Information Systems. Morgan Kaufmann Publishers. San Francisco.

[5] Object Management Group. 2003. Transaction Service Specification, version 1.3. OMG Technical Committee Document ptc/2003-03-08.

[6] Object Management Group. 1995. Object Management Architecture Guide, revision 3.0, OMG Technical Committee Document ab/97-05-05.

[7] Elmagarmid, A. K. Ed.1990. Database Transaction Models For Advanced Applications. Morgan Kaufmann. California.

[8] Gallina, B., Guelfi, N. and Romanovsky, A. 2007. Coordinated Atomic Actions for dependable distributed systems: the current state in concepts, semantics and verification means. In Proc. 18th IEEE Int. Symposium on Software Reliability (Trollhattan, Sweden, 2007). 29-38.

[9] Georgiadis, C. K. and Pimenidis, E. 2007. Proposing an evaluation framework for B2B web services-based transactions. In Proc. IASK Int. Conf. on e-activity and leading technologies (Porto, Portugal). 164-171.

[10] Wu, H., Kemme, B. and Maverick V. 2004. Eager replication for stateful J2EE servers. In: On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE (Cyprus, 2004). LNCS 3291 Springer-Verlag. 1376-1394.

[11] Varga, A. 2001. The OMNeT++ Discrete Event Simulation Environment. In Proc. European Simulation Multiconference (Prague, Czech Republic, 2001). Society for Computer Simulation. 319-325.

[12] OMNeT++ Community Site, http://www.omnetpp.org/ (last access: 29th of November 2006)

[13] Katsaros, P., Iakovidou, N., and Soldatos, T. 2006. Evaluation of composite object replication schemes for dependable server applications. Information and Software Technology 48, 9. Elsevier. 795-806.

[14] Katsaros, P., and Lazos, C. 2004. Optimal object state transfer - recovery policies for fault tolerant distributed systems. In Proc. IEEE/IFIP Int. Conference on Dependable Systems and Networks (Florence, Italy, 2004). 762-771.

[15] Katsaros, P. Angelis, L. and Lazos, C. 2007. Performance and effectiveness trade-off for checkpointing in fault tolerant distributed systems. Concurrency and Computation: Practice and Experience 19, 1. John Wiley & Sons, 37-63.

[16] Martin, C. P. and Ramamritham, K. 1997. Toward formalizing recovery of (advanced) transactions. In Advanced Transaction Models and Architectures, S. Jajodia and L. Kerschberg Ed. Kluwer, Boston.

[17] Heptane Site, http://www.irisa.fr/aces/work/heptane-demo/heptane.html (last access: 4th of December 2007)

[18] Thanisch, P. 2000. Atomic commit in concurrent computing. IEEE Concurrency (October – December 2000). 34-41.

[19] Schlichting, R. D. and Schneider, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems, ACM Transactions on Computer Systems 1, 3. 222-238.

[20] Katsaros, P., Angelis, E. and Lazos, C. 2001. Applied multiresponse metamodeling for queuing network simulation experiments: problems and perspectives, In Proc. EUROSIM 2001 Congress (Delfts, The Netherlands, 2001).

[21] ACID Sim Tools Site, http://mathind.csd.auth.gr/acid/html/index.html (last access: 11th of December 2007)