A Framework for High Performance Simulation of Transactional Data Grid Platforms*

Pierangelo Di Sanzo, Francesco Antonacci, Bruno Ciciani, Roberto Palmieri, Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, Diego Rughetti, Roberto Vitali

HPDCS Research Group, DIAG - Sapienza, Università di Roma

ABSTRACT

One reason for the success of in-memory (transactional) data grids lies on their ability to fit elasticity requirements imposed by the cloud oriented pay-as-you-go cost model. In fact, by relying on in-memory data maintenance, these platforms can be dynamically resized by simply setting up (or shutting down) instances of so called *data cache* servers. However, defining the well suited amount of cache servers to be deployed, and the degree of in-memory replication of slices of data, in order to optimize reliability/availability and performance tradeoffs, is far from being a trivial task. To cope with this issue, in this article we present a framework for high performance simulation of in-memory data grid systems, which can be employed as a support for timely what-if analysis and exploration of the effects of reconfiguration strategies. The framework consists of a discrete event simulation library modeling differentiated data grid components in a modular fashion, which allows easy (re)-modeling of different data grid architectures (e.g. characterized by different concurrency control schemes). Also, the library has been designed to be layered on top of the open source ROOT-Sim parallel simulation engine, natively offering facilities for optimized resource usage in the context of model execution on top of multi-core and cluster based architectures. Finally, instances of data-grid models supported by the framework have been validated against real measurements obtained by deploying the Infinispan data grid onto Amazon EC2 virtual clusters, and running the well known TPC-C benchmark. By the experiments we demonstrate closeness of simulation outputs and real measurements, while jointly showing extreme scalability of the framework, in terms of speedup and ability to manage extremely large data grid models.

*This work has been partially supported by the Cloud-TM project (co-financed by the European Commission through the contract no. 57784). The package has been released as part of this project deliverables, and is accessible at the URL https://github.com/cloudtm/cloudtm-autonomic-manager/tree/master/src/dags.

Simutools 2013, March 05-07, Cannes, France

DOI 10.4108/icst.simutools.2013.251737

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed Applications*; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*Discrete Event, Parallel*

General Terms

Algorithms, Performance, Measurement

Keywords

Transactional data platforms, Parallel discrete event simulation

1. INTRODUCTION

With the advent of cloud computing, we have experienced the proliferation of a new generation of in-memory, transactional data platforms, often referred to as NoSQL data grids, among which we can find products such as Red Hat's Infinispan [12], VMware vFabric GemFire [19], Oracle Coherence [15] and Apache Cassandra [14]. These platforms well meet the elasticity requirements imposed by the pay-as-you-go cost model since they (a) rely on a simplified key-value data model (as opposed to the traditional relational model), (b) employ efficient in-memory replication mechanisms to achieve data durability (as opposed to disk-based logging) and (c) natively offer facilities for dynamically resizing the amount of hosts within the platform.

However, one aspect that still represents a core issue to cope with is related to how to (dynamically) dimension and configure the system in order to, e.g., match a predetermined Service Level Agreement (SLA), while also minimizing operating costs related to, e.g., renting the underlying virtualized infrastructure. In fact, forecasting the scalability trends of real-life, complex applications deployed on distributed in-memory transactional platforms is an extremely challenging task. Specifically, as also shown in [8], when the number of nodes in the system grows and/or the workload intensity/profile changes, the performance of these platforms may exhibit strong non-linear behaviors, which are imputable to the simultaneous, and often inter-dependent, effects of contention affecting both physical (CPU, memory, network) and logical (conflicting data accesses by concurrent transactions) resources.

Recent approaches have tackled the issue of dynamically reconfiguring these platforms via the reliance on analytical modeling, machine-learning techniques or a combination of the two approaches (see, e.g., [8]). In this article we provide

an orthogonal solution which is based on high performance simulation techniques. Specifically, we provide a framework for simulating data grid platforms on top of high performance parallel discrete event simulation (PDES) engines, particularly the open source ROOT-Sim engine [11]. The framework can be exploited for timely what-if analysis in order to determine what would be the effect of reconfiguring various parameters, like:

- number of cache servers within the platform;
- degree of replication of the data-objects;
- placement of data-copies across the platform.

Hence, it can be exploited in order to timely determine well suited configurations (e.g. minimizing the cost for the underlying virtualized infrastructure) vs variations of the volume of client requests, the actual data conflict and the locality of data access. It can also be used for long term planning within SLAs in order to determine whether to accept some scale-up in the maximum sustainable volume of requests, and at what cost for the customer (as a reflection of the planned costs for the underlying infrastructure).

The framework has been developed as a library implementing data grid models developed according to the traditional event-driven approach, where the evolution of each individual entity to be simulated within the model is expressed by a specific event-handler. These event handlers have been implemented in order to make them compliant with the programming model offered by the ROOT-Sim parallel/distributed simulation environment. Hence the library can be natively hosted on top of a run-time environment allowing high performance model execution via transparent parallelization/synchronization of the running code on multi-core machines and clusters.

On the other hand, the library has been structured in order to provide a means for easy development of data grid models offering specific facilities and supporting proper algorithms (e.g. in terms of management of the consistency of replicated data). In particular, distributed data grids relying on two-phase-commit (2PC) as the native scheme for cache server coordination (as typical of most of the mainstream implementations [12]), have an execution pattern already captured by the the skeleton model offered by the library. Hence, differentiated data management protocols could be easily implemented via the framework. Models natively offered within the framework include data grids ensuring repeatable read semantics, which are based on lazy locking. Primary data ownership vs multi-master schemes are also natively offered.

Fidelity of the provided skeleton and models has been demonstrated by comparing simulation outputs with real measurements achieved by running the TPC-C benchmark [18] on top of the Infinispan data grid system by JBoss/RedHat, namely the mainstream data layer for JBoss applications, on virtualized infrastructures hosted by Amazon EC2. On the other hand, from the experimental study we also show how the framework can provide speedup and fast delivery of simulation outputs for largely scaled up models.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. The framework architecture is presented in Section 3. Experimental data are reported in Section 4.

2. RELATED WORK

The issue of optimizing the configuration of data grids has been addressed in literature according to differentiated methodologies. The work in [8] provides an approach where analytical modeling and machine learning are jointly exploited in the context of performance prediction of data grid systems hosted on top of cloud based infrastructures. The analytical approach is mainly focused on capturing dynamics related to the specific concurrency control algorithm adopted by the data grid system, while machine learning is targeted at capturing contention effects on infrastructure-level resources. Differently from our approach, this work copes with a specific data grid configuration (hence a specific data management algorithm) to which the analytical model is targeted. Instead, we offer a framework allowing the modeling of differentiated data management schemes, which does not even impose specific assumptions on the workload profile (in fact real execution traces can be used to drive the simulated data access). Also, the employment of machine learning, which requires the actuation of training, does not always allow reliable extrapolation of performance values related to configurations that significantly diverge from those that have been observed in the training phase. This problem is avoided via the approach we propose since we purely rely on simulation. The latter consideration also applies to the proposal in [6], which provides data grid reconfiguration schemes based on pure machine learning.

One approach close to our proposal has been presented in [17], where a simulation layer is provided entailing capabilities of simulating data grid systems. Differently from the present proposal, the architecture of the simulation system is based on a data grid simulation layer (as opposed to our lower layer, which is a general purpose parallel simulation engine). Further, the whole system architecture is based on Java technology, while we target C technology and parallel processing, hence endemically increasing the relevance of (and focus on) performance aspects of the simulation environment, which is especially important in the case of real-time re-configuration strategies based on what-if analysis.

Simulation of data grid systems has also been addressed in [13]. In this proposal, the modeling scheme of the data grid is based on Petri nets, which are then solved via simulation. Instead, we proposed a functional model not explicitly relying on modeling formalisms, except for the case of some specific component (such as the CPU). These are anyway modeled via queuing approaches, rather than Petri nets. Further, one relevant difference between the work in [13] and our proposal lies on that our simulation models are able to simulate complex transactional interactions entailing multiple read/write (namely get/put) operations within a same transaction. Instead, the work in [13] only models single get/put interactions to be issued by the clients. As for this aspect, our approach can be considered as more general.

Finally, a less closely related work to our proposal can be found in [5], where simulation supports for backup data storage systems in peer-to-peer networks are presented. Compared to our present proposal, this work is focused on lower level data management aspects, such as the explicit modeling of actual stable storage devices. Instead, our focus is on distributed dynamics at the level of in-memory data storing schemes, which are independent of stable storage technologies.

3. THE SIMULATION FRAMEWORK

As hinted, our framework has been designed in order to be layered on top of the ROOT-Sim parallel/distributed simulation engine, which offers supports for optimistic synchronization of the simulation objects included within the model. Hence, some of the framework design choices have been taken in order to provide a final structure that is aimed at not hampering the exploitation of parallelism while model execution is in progress. One major choice along this direction has been targeted at reducing to a minimum the presence of simulation objects that may figure out as synchronization bottlenecks, which, in the context of optimistic synchronization, maps onto avoiding the presence of simulation objects whose rollbacks can lead to spreading effects where cascading rollback is induced on many other objects within the model. Particularly, network delays for messages sent across the different components within the simulation model are not simulated by explicitly including a network simulation-object. Instead, each simulation object within the model has the ability to compute network delays associated with simulating message-send operations towards other simulation objects, and to schedule the corresponding message-arrival events onto the destination object.

Overall, our simulation models of data grid systems are mapped onto a system representation only entailing two types of simulation objects: (A) client objects and (B) cache server objects. Having multiple client objects running concurrently on top of ROOT-Sim leads to great exploitation of parallelism while generating the simulated workload of requests towards the data grid layer. On the other hand, having multiple cache server objects running concurrently leads to great exploitation of parallelism while simulating actual data access operations, especially when the simulated configuration entails partial data replication schemes (as typical of when high scalability needs to be provided within the data grid [16]).

As for the latter point, given that our base simulated protocol for cache server coordination is 2PC, upon any 2PC-based coordination action, only those cache servers maintaining copies of the data to be locked need to mutually exchange simulation events. If the degree of replication is limited, as typical of partial replication schemes, the set of servers that coordinate with each other while handling client requests gets reduced, hence leading to simulate the evolution of groups of cache servers along different critical paths in different phases of the simulation run. Again, this likely favors parallelism in the model execution.

In the next subsections we focus on the structure and event patterns of cache server and client simulation objects. Particularly, we focus on their associated skeleton and on the offered supports for easy modifiability of their simulated logic so to allow easy re-implementation of differentiated data grid simulation models, particularly on the side of differentiated concurrency control schemes and supports for distributed transaction atomicity.

3.1 The Cache Server Simulation-Object

A cache-sever simulation object within our framework can be schematized as shown in Figure 1. By the scheme we can identify four main software components:

- the transaction manager (TM);
- the distribution manager (DM);

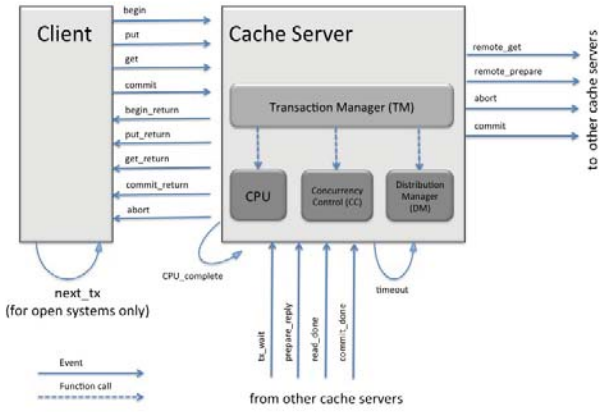

Figure 1: Client and Cache-Server Simulation Objects.

- the concurrency control (CC); and
- the CPU.

Any simulation event destined to the cache server is eventually passed in input to TM, which acts therefore as a front-end for event processing. Once scheduled whichever event, TM determines the amount of CPU time required for actual processing of the requested activity, which depends on the type of the scheduled event, and on the current CPU load. Hence, the CPU load is updated on the basis of the newly scheduled activity. Then the completion time for the activity is determined, which depends on the current CPU load, and a CPU-complete event is scheduled, at the corresponding simulation time. To determine the CPU delay, the CPU resource has been modeled as a G/M/K queue, which allows capturing scenarios entailing multiple CPU-cores. Although more sophisticated models could be employed (see, e.g., [9]), we relied on G/M/K queues since, in our target simulation scenarios, the core dynamics are associated with contention on logical resources, namely data-objects, rather than physical resources, and on distributed (locking) strategies for the management of atomicity of the updates of distributed/replicated data copies. Hence, communication delays play a major role in the determination of the achievable performance, as compared to CPU delays for processing local activities. As a consequence, the G/M/K queue is expected to reveal adequate for the objectives of the framework. For the same reason depicted above, effects of virtual memory on the latency of operations provided within the data grid simulation model are not explicitly considered.

When a local processing activity gets completed, TM takes again control (via the aforementioned CPU-complete event) and performs the actual updates related to the activity. These updates are different depending on the exact type of event that triggered CPU work.

As for simulation events scheduled by client simulation objects towards the cache servers, the corresponding event-types within our skeleton are listed below:

- begin, used to notify to TM that a new transactional interaction has been issued by some client, which must be processed by the cache server;
- get, used to notify that a read operation on some data

object has been issued by the client within a transaction;

- put, used to notify that a write operation on some data object has been issued by the client within a transaction;

- commit, used to indicate that the client ended issuing operations within a transaction, which can therefore attempt to commit.

The handling of the begin event at the side of TM in our framework automatically triggers the internal function `setupTransaction`, which simply takes as input the current simulation time and pointers to two records of type `TxInfo` and `TxStatistics`, which are both automatically allocated by the framework and linked to the corresponding records for already active transactions. The actual internal structure of both the `TxInfo` and `TxStatistics` records is simulation-modeler defined, in fact the framework provides a proper header file, named `transaction.h`, where the modeler can specify the structure. The only constraint is that the top standing field of `TxInfo`, must be of type `TxId`. It keeps the transaction unique identifier, automatically generated by the cache server just to facilitate the actual management within model execution.

This is one of the core modules on top of which flexibility of actual model implementation within the framework lies. In fact, with this organization, the modeler can keep track of management information (i.e. `TxInfo`) and statistics information (i.e. `TxStatistics`) associated with active transactions within whichever modeler-defined data structure, which gets automatically allocated and managed by the framework into the heap. The reason for allowing the modeler to exploit two different data types lies on that the content of `TxInfo` is made valid according to a cross cache-server scheme. In fact, it is automatically transferred to remote cache servers when cross scheduling of events is actuated, as we shall discuss. This is relevant in any simulated scenario where some transaction set-up information needs to be made available to remote cache servers for, e.g., distributed contention management purposes. On the other hand, the content of `TxStatistics` is not transferred across different simulation objects, being instead locally handled by the cache server acting as the coordinator of the transaction. In particular, upon finalization of a transaction, TM automatically invokes the module `finalizeTransaction`, which receives in input the current simulation time, and again pointers to both `TxInfo` and `TxStatistics` records so to allow to update them (particularly the statistics). The release of these buffers within the framework is again handled automatically. However, before releasing any of them, a special module `statisticsLog` is called, passing in input pointers to both of them, hence the modeler can finally log, e.g. onto the file system, any provided statistical data. We note that the identification of output operations related to the committed portion of the simulation is done within the framework by exploiting proper ROOT-Sim facilities. Hence, only committed output calls area actually reflected within output files.

As for get and put simulation events, they cause the TM module to simply query (via synchronous procedure invocation) the DM module. This is done in order to get information about what cache servers figure as the owners of the data object to be accessed. In our architecture, the DM module provides this information back in the form of a pointer to a list of cache server identifiers (hence simulation object identifiers), where each record also keeps additional information specifying whether a given cache sever is (or is not) the primary owner of a copy of the data object to be accessed. Once TM gets this information, it then determines the pattern of additional simulation events to be scheduled. More in detail, primary ownership has relevance for put (namely write) operations on data objects. Instead, get operations are not affected by the presence of a primary owner, if any. Let us discuss this aspect in detail.

In case of get simulated events, the cache sever determines whether it is the owner of a copy of the data object. In the positive case, the read operation on the data object will simply result in an invocation of the CC module on this same cache server instance. Otherwise, remote_get simulation events are scheduled (at later time, which models the corresponding request transmission delay) for all the cache servers figuring out as owners of a copy of the data object. Upon their execution at the destination cache severs, which will still entail passing through the simulated CPU processing stage, these events will trigger CC invocations on those cache server simulation objects.

One important aspect associated with the above scheme is that the get operation may be blocked at the level of CC, depending on the actual policy for controlling concurrency. On the other hand, even in case of CC simulated algorithms implementing non-blocking read access to data (as is the case for most data grid products guaranteeing weak data consistency, such as read committed or repeatable read semantics [12], as well as for some optimized data grid architectures providing more strict consistency levels, like update-serializability [16]), the read operation may anyway be blocked in case no local copy exists and needs to be fetched by some remote cache sever. This is automatically handled by our framework since the TM module records information on any pending simulated read operation within a proper data structure. When setting up the record for a given operation, information on the remotely contacted cache servers, if any, is also installed. That record will be removed only after processing the corresponding reply simulation events from all those cache servers, which is done for allowing an optimized execution flow for those reply events. On the other hand, the operation is unlocked (and a reply event is scheduled towards the corresponding client) upon the first copy of the data becomes available from whichever cache server, hence after processing the first simulation event associated with a read-reply. Note that this architectural organization automatically covers the case where the transaction operation is blocked locally, due to the current state of CC. In such a case, the contacted-server list will be filled with the identifier of the local cache server, and a read-reply event from this same server (which will be scheduled by the local CC module, as we shall discuss) will be used to unlock the request and schedule the reply towards the client simulation object.

In case of put operations (namely data object updates) the corresponding simulation events only trigger the update of some meta-data locally hosted by the cache server, which are embedded into records treated at the same manner as the above mentioned modeler definable `TxInfo` record. These data include the operation identifier, and the key associated

with the data object to be updated. This behavior simulates a simple local update of the transaction write set, which is again reflected into a cross cache sever valid record (in case cross server events for that transactions are scheduled) which we name `TxWriteSet`. On the other hand, these meta-data are queried upon simulating a get operation to determine whether the data object to be read already belongs to the transaction write set (hence whether the get operation can be served immediately via information within the write set). In such a case, the simulation-event pattern for handling the get is different from the general one depicted above since it only entails simulating local CPU usage required for providing the value extracted from the transaction write set to the client.

More complex treatments are actuated when handling commit simulation events incoming from the clients. In particular, differentiated simulation event patterns are put in place by TM depending on whether the simulated scheme entails a primary owner for each data object or not. For primary ownership scenarios, the prepare will result in scheduling remote_prepare events towards all the primary cache servers that keep copies of the data to be updated (each event carries the keys associated with the data objects to be updated, which are again retrieved via the `TxWriteSet` data structure maintained by the cache server acting as transaction coordinator). TM can determine this set of cache servers by exploiting the keys associated with the written data objects (which are kept within the transaction write set). If one of these cache servers corresponds to the server currently processing the prepare request, then, after passing through the CPU processing stage, the local CC module is immediately invoked. At this point we are in a similar situation as the one depicted above for the case of read access to remote data. In particular, for the preparing transaction, the framework logs the identities of the contacted servers, and then waits for the occurrence of prepare_reply simulation events scheduled by any of these servers. For homogeneity, even in case one of the contacted CC module is the local one, the reply from this module occurs via the scheduling of such a `prepare_reply` event, thus giving rise to the situation where the CC module exhibits the same simulated behavior (in terms of notification of its decisions) independently of whether the prepare phase for the transaction needs to run local tasks on the same cache server, or remote tasks. Hence, the CC module operates seamless of any simulated data distribution/replication scheme. The above simulation-event pattern is only slightly varied in case of non-primary ownership of data objects since the framework will schedule these prepare events for all the servers keeping copies of the data to be updated. This again allows the CC module to operate transparently to the ownership scheme. On the other hand, for both the schemes, in case the prepare_reply events are positive from all the contacted servers, final commit events are scheduled for all of them, which will ultimately result in invocations of the CC module. On the other hand, abort events are scheduled in case of negative prepare outcome. Further, for the case of primary ownership, the commit events are propagated to the non-primary owners, in order to let them reflect data update operations.

Let us now come to details related to the CC simulation module, which represents one core component for our framework architecture. By the above description, this module gets invoked upon the occurrence of get or remote_get events, remote_prepare events, and commit events. However, all the above events are actually intercepted and initially processed by the TM module, which, as said, is the front end simulation-handler within the cache server simulation object. Hence, ultimately, the CC module does even not know whether a requested action is associated with some local or remotely executed transaction. It only acquires as input parameters:

- a pointer to the `TxInfo` record (recall that, in the simulation flow, the field `TxId` within this record has been automatically setup by TM upon processing the begin event, while additional transaction information can be defined by the modeler by setting it up via the `setupTransaction` module);
- a pointer to `TxStatistics` (or null if the cache server is not the transaction coordinator);
- the `type` of the operation to be actuated (read, prepare or commit);
- the `key` of the data object to be involved in the operation (this is for read operations);
- the `TxWriteSet` to be used for CC purposes (this is for the prepare case).

On the other hand, CC can reply to invocations by raising to TM the generation of one or more of the below listed events:

- `TX_WAIT`, indicating that the currently requested operation leads to temporary block the transaction execution;
- `READ_DONE`, indicating that the data object can be returned to the reading transaction;
- `PREPARE_DONE`, indicating that the transaction is successfully prepared;
- `PREPARE_FAIL`, indicating that the transaction prepare stage has not been correctly completed;
- `COMMIT_DONE`, indicating that the transaction commit request has been processed.

Each of the above events is not directly routed towards the destination simulation object (hence these events are not proper simulation events, but only event generation indications), right because CC is not aware of whether they must represent replies for the local cache server or remote cache servers, or even the client. Hence, within the framework they are intercepted by a proper layer, which buffers these CC triggered event-generation requests so to make them available for actual scheduling (towards the correct destinations), which is actuated by the TM module once it takes control back upon the return of CC. As such, the events triggered by CC can be re-mapped onto actual simulation events to be exchanged across different simulation objects. As an example, `PREPARE_DONE` and `PREPARE_FAIL` events are re-mapped and actually scheduled as the aforementioned `prepare_reply` events, with proper payload (indicating positive or negative prepare outcomes). Further, the CC module can raise the request for issuing `TIMEOUT`

events, which can be useful in scenarios where CC actions are also triggered on the basis of passage of time.

Overall, the simulation modeler can implement different concurrency control algorithms by completely ignoring data distribution and replication schemes. He only needs to deal with transaction identifiers, basic transaction setup information and relations across different transactions, on the basis of the actual data objects locally hosted. To determine what are the locally hosted data objects, hence the locally hosted keys, CC can access a hash table, that gets automatically setup upon simulation startup. On the other hand, the meta-data required to keep relations across active transactions, (e.g. wait-for relations), and the corresponding data structure is completely left to programming by the simulation-modeler. It can be again defined, in terms of types, within the `transaction.h` header file. On the other hand the actual instance of this data structure, can be accessed via a special pointer which gets passed to the CC module by the framework as an additional input parameter. We note that if the pointer value is `NULL`, then CC has not yet allocated and initialized the structure, hence this must be done, and the actual pointer to be used in subsequent calls to CC can be setup and returned upon completion of the current CC execution.

Let us go bach to the TxInfo record. As we have said, this is modeler defined and can keep track of per-transaction meta-data, which can be exploited by the CC module in order to support the actual concurrency control logic. Let us consider two different examples of how to model via the framework different CC algorithms. One is a classical 2PC based data-grid CC algorithm where every transaction is successfully prepared at any site in case the target data object to be updated is not currently locked upon the prepare request. On the other hand, the second scenario shows how to model cases where the transaction is prepared only in case the target data has a timestamp lower than the transaction timestamp. The examples are presented via pseudo-code for simplicity.

Example One. In Figure 2 we show the pseudo-code defining the entries of `TxInfo` and some part of the core logic at the level of CC. In this case, `TxInfo` is not required to keep transaction control information targeted at contention management. Basically it keeps transaction identification information. On the other hand, the base setup for concurrency management can be actuated by simply setting up a wait-for table where transaction identifiers are queued in different rows depending on what other transaction holds the lock they would like to get on a given data object (the top standing transaction is therefore the one to which the lock has been granted). In the pseudo-code we show a scheme where, upon the handling of any prepare request, the associated transaction always gets queued. On the other and upon commit or abort events for a pending transaction, the subsequent one in the wait-for list gets reactivated, with positive reply to the original prepare request.

Example Two. In Figure 3 we show the pseudo-code defining the entries of TxInfo and some parts of the core logic at the level of CC, where this time we have a variation that leads the `TxInfo` record to keep cross-server control information specifically targeted at data contention management, namely a timestamp value. In this case, differently from the previous scenario, a transaction for which a prepare event has been issued can get successfully prepared only in case its timestamp is greater than the timestamp of any data object accessed in write mode. We note that this time, the CC module, upon setting up the `CC-Table` needs to take care of setting up meta-data for the explicit maintenance of data object timestamp values.

```
record TxInfo{
    TxId
    ...
} //end record

CC-logic(input: task T, pointer CC-Table){

if (CC-table == NULL)
    allocate and initialize [wait-for,active-tx] table;
    // keys are data object identifiers or TxId values
    // entries are lists of TxInfo records or TxId values
    set CC-table point to the allocated table

case T.type
    prepare:
       link T.TxInfo.TxId to CC-Table.active-tx
       AllPrepareKeys = T.TxWriteSet
       link T.TxInfo to CC-Table.wait-for[AllPrepareKeys]
       if T.TxInfo not top standing for some key
          generate event TX_WAIT[T.TxInfo]
          generate event TIMEOUT[T.TxInfo]
       else generate event PREPARE_DONE[T.TxInfo]
    ....
    timeout:
    commit:
       unlink T.TxInfo.TxId from CC-Table.active-tx
       unlink T.TxInfo from CC-Table[AllOccurrences]
       if (T.type == commit) generate COMMIT_DONE[T.TxInfo]
       else generate PREPARE_FAIL[T.TxInfo]
       for all TxInfo top standing in CC-Table[AnyPresenceRow]
           generate event PREPARE_DONE[TxInfo]
    ....

return CC-Table
} //end CC
```

Figure 2: Example One.

3.2 The Client Simulation-Object

Client simulation objects have an internal structure that does not need to be changed by the simulation-modeler. In fact, the modeler only needs to specify, via proper configuration files within the framework, what type of distribution must be used for determining the data to be accessed, and what distribution needs to be used for determining the number of operations to be executed within a transaction and the type (read or write) of each operation.

As for this aspect, the framework already offers the possibility to use differentiated access distributions, some of which are analytical, while others have been determined by relying on traces of known benchmarks. Further, the clients can be configured in order to simulate either an open or a closed system. Fort the former case, the simulation modeler needs to specify the rate of generation of transactions at the client side.

As a final note, our clients also embed the possibility to generate the workload by directly relying on traces (rather than on distributions derived from the traces). This was not straightforward given the optimistic nature of the underlying simulation engine. Specifically, the ROOT-Sim optimistic engine manages transparent rollback operations for any in-memory data structure (even if allocated via `malloc` services), and supports rollback operations for I/O interac-

```
record TxInfo{
    TxId
    timestamp
    ...
} //end record

CC-module(input: task T, pointer CC-Table){

if (CC-Table == NULL)
    allocate and initialize [wait-for,DOT] table;
    // DOT stands for data-object-timestamp
    // table access keys are data object identifiers
    // entries are lists of TxInfo records or DOT values
    set CC-Table point to the allocated table

case T.type
    prepare:
        AllPrepareKeys = T.TxWriteSet
        if T.TxInfo.timestamp > CC-Table.DOT[AllPrepareKeys]
           link T.TxInfo to CC-Table[AllPrepareKeys]
        else generate event PREPARE_FAIL[T.TxInfo]
            goto out
        if T.TxInfo not top standing for some key
          generate event TX_WAIT[T.TxInfo]
          generate event TIMEOUT[T.TxInfo]
       else generate event PREPARE_DONE[T.TxInfo]
    ....

out:
 return CC-Table
} //end CC
```

Figure 3: Example Two.

tions only in case of output. Instead, input operations are not automatically rollbackable (since this would entail, in principle, facilities for management of input data from, e.g., an interactive user, which are still not present in ROOT-Sim). This would lead to problems when subsequent portions of the trace on file are dynamically acquired during the simulation run. To bypass these problems, we have adopted a scheme, that operated transparently to the modeler, which is based on logging onto an apposite temporary file information on the amount of bytes read from the trace, which is also logged into a volatile memory variable. If upon the need for reading the next fraction of the trace the two values are not identical, it means that some previous read from the trace file has been undone by a rollback. In this case the client simulation object seeks the file pointer to the trace to the correct position in order to re-execute the correct read operation for the given portion of the trace, and then updates the two counters to make them coherent again. Again we note that this mechanism is completely transparent to the modeler, and hence to the final user.

4. EXPERIMENTAL RESULTS

4.1 Validation

Validating a whole framework is not an easy task since it would entail validating any possible model that is feasible to be implemented via the framework. On the other hand, for our framework, feasible models need to rely on specific skeleton operations, such as 2PC for coordinating the distributed caches servers. Hence, validating at least one model that exploits such a skeleton would anyhow represent a significant step. This is exactly the choice we have taken, since we present validation data obtained by comparing simulation results with the corresponding results achieved when running a real-world data grid system, exploiting 2PC, namely Infinispan by JBoss/Red-Hat [12], on top of a cloud based infrastructure hosted by Amazon EC2. The used benchmark for validation is a port of TPC-C [18] (already exploited in, e.g., [16]) that has been performed in order to allow this benchmark to be adapted to the $< key, value >$ data model (rather than the original relation model used in the benchmark specification).

As for Infinispan, this is a popular open source in-memory data grid currently representing both the reference data platform and the clustering technology for JBoss, which is the mainstream open source J2EE application server. Infinispan exposes a pure key-value data model (NoSQL), and maintains data entirely in-memory relying on replication as its primary mechanism to ensure fault-tolerance and data durability. As other recent NoSQL platforms, Infinispan opts for weakening consistency in order to maximize performance. Specifically, it does not ensure serializability [3], but only guarantees the Repeatable Read ANSI/ISO isolation level [2]. At the same time, atomicity of distributed updates is achieved via 2PC. Particularly, in the version selected for the experiments, namely V5.1, the 2PC protocol operates according to a primary-owner scheme. Hence, during the prepare phase, lock acquisition is attempted at all the primary-owner cache servers keeping copies of the data-objects to be updated. If the lock acquisition phase is successful, the transaction originator broadcasts a commit message, in order to apply the modifications on these remote cache servers, which are then propagated to the non-primary owners.

In presence of conflicting concurrent transactions, the lock acquisition phase may fail due to the occurrence of (possibly distributed) deadlocks. Deadlocks are detected using a simple, user-tunable, timeout based approach. In our experimental assessment, we consider the scenario in which the deadlock detection timeout is set to few milliseconds (ranging from 2 to 5 depending on the actual size of the infrastructure, namely number of virtualized hosts, on top of which cache servers run). Very small timeout values, as the one we have selected, are typical for state of the art in-memory transactional platforms [7] since distributed deadlocks represent a major threat to system scalability, as highlighted by the seminal work in [10].

The whole test-bed architecture has been deployed onto an Amazon EC2 environment where both clients and cache servers run on top of *small EC2 instances* equipped with 1.7 GB of memory and one virtual core providing the equivalent CPU capacity of 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processors. Each machine runs Linux Ubuntu 10.04 with kernel 2.6.32-316-ec2. We have varied the number of cache servers between 2 and 4. Also, the number of clients deployed on the infrastructure has been set up to 64. We note that, although one might think that this is a configuration with a reduced number of clients, our clients issue TPC-C transactions towards the cache servers with zero-think time, which gives rise to sustained workload. Hence we are emulating the scenario where the clients we deployed onto the EC2 platform mimic the behavior of front-end servers, which access the Infinispan in-memory data layer ultimately hosted by the cache servers (acting as back-end servers). In other words, we mimic a scenario where they would be handling multiple non-zero think time interactions by actual end-clients. As for the transaction access pattern issued by the clients, we have used a model expressing the

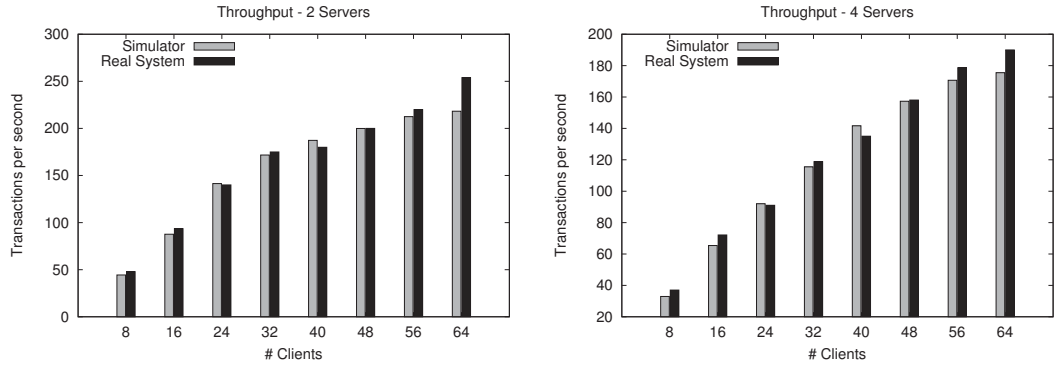

Figure 4: Comparison Between Simulation and Real Data.

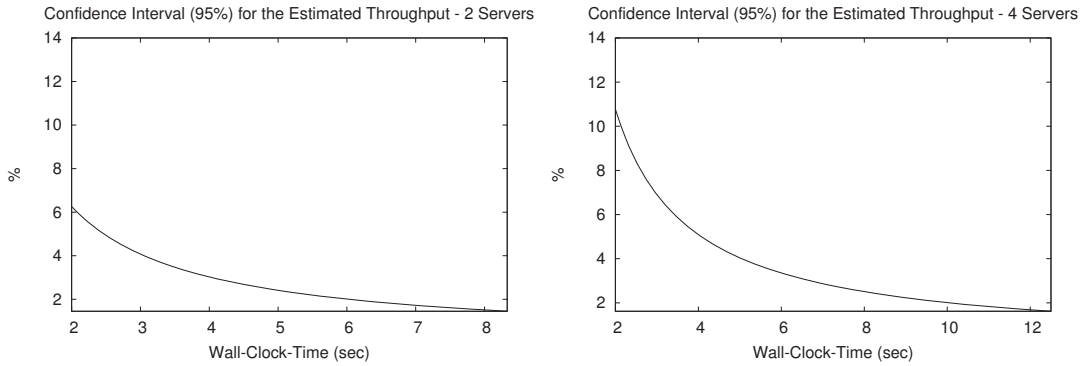

Figure 5: Confidence Interval for the Estimated Throughput with Respect to the Wall-Clock-Time with 64 Clients.

corresponding data distribution access specified within the TPC-C benchmark. As a final preliminary note, in these experiments we have considered replication degree of the data objects set to the value 2, which is a typical settings [1].

We have traced the execution of the benchmark in order to determine both (a) the parameters to be used within the simulation model (such as the CPU demand for specific operations, and the average transmission delay between the hosts) and (b) the actual statistics used for validating the simulator. As for point (a), the exact list of measured parameters is reported in Table 1. They all refer to the CPU demand for specific operations at the cache servers, except for the transmission latency. In relation to the latter parameter, in the simulation framework we have adopted a message delay model based on an exponential distribution where the average transmission delay reported in Table 1 expresses the corresponding mean value. We note that such a delay refers to what is observable at the application (e.g. data platform) level, not at the level of actual host-to-host delay (as typical of when pinging the hosts). Hence, it includes, e.g., marshalling/unmarshalling costs at the level of the JGroups group communication layer used by Infinispan. As for point (b) the validation has been based on the system throughput, for which we report in the plots in Figure 4 both the simulation results and the real measurements taken on the test-bed platform.

By the results we observe good matching between simulated values of the throughput and real ones, with a discrepancy that is bounded by 16%. Such a bound is reached only for the configuration with 2 cache servers and 64 clients. Instead, for all the other considered configurations, the actual discrepancy between real and simulated data is even lower (typically on the order of no more than 10%).

Table 1: Measured Parameter Values.

Parameter Name	Value (msec)
local_tx_get_cpu_service_demand	0.027
local_tx_put_cpu_service_demand	0.022
local_tx_get_from_remote_cpu_service_demand	0.015
tx_send_remote_tx_get_cpu_service_demand	0.022
tx_begin_cpu_service_demand	0.004
tx_abort_cpu_service_demand	0.369
tx_prepare_cpu_service_demand	0.129
distributed_final_tx_commit_cpu_service_demand	0.077
cross_node_transmission_latency	36

In Figure 5 we provide data related to how the statistical significance of the throughput values computed by the simulator varies vs the wall-clock-time of the simulation run. These data refer to serial executions of the simulation models, carried out on top of the same platform we have employed for demonstrating efficiency and scalability of the parallel runs, whose details will be provided in the next section. The reported data refer to 64 clients, and show how the wall-clock-time requested for achieving high confidence of the produced statistics is on the order of 8 to 12 seconds (depending on the amount of simulated cache servers). For

(much) scaled up model sizes, this requirement would likely be significantly higher thus demanding for parallel computation. This point is the objective of the study in the next section.

4.2 Simulation Performace and Scalability

Another aspect of relevance for the presented framework is the actual performance and scalability it can provide while carrying out model resolution. To address this issue and provide quantitative data in relation to this aspect, we have carried out an experimentation based on simulation models with scaled up size. In particular, we have run experiments with up to 1024 simulation objects where 1/8 of them are cache servers, and the remaining ones are clients.

The simulation runs have been carried out on a clustered architecture relying on a couple of HP Proliant servers. Each server is a 64-bit NUMA machine, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 CPU-cores (for a total of 32 CPU-cores) that share a 10MB L3 cache (5118KB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux kernel version 2.6.32.5. Overall, across the two servers a total number of 64 CPU-cores have been used.

We have performed experiments where we have measured the event rate, namely the cumulated amount of committed events per wall-clock-time unit, which is a typical indicator for the speed of model execution in the context of optimistic parallel/distributed simulation engines. We have measured this parameter while increasing the size of the simulation model (as said up to 1024 simulation objects) and while adopting two different deploy strategies of the simulation objects across the cluster. Particularly, in one strategy we hosted the simulation objects on different simulation kernels that are deployed as much as possible on the same machine. When no more CPU-cores on this machine are available for additional simulation kernel instances, then the second machine starts to be used (we refer to this deploy as block-deploy). In the other hand, in the second strategy the different kernel instances that are added while the simulation platform is resized are deployed onto the two machines according to a round robin scheme. The two different deploys allow us to observe the simulation system performance while varying the ratio between local (inter-machine) event scheduling and remote one.

The achieved results are shown in Figure 6, where the symbol K indicates the number of used simulation kernel instances (hence the number of CPU-cores exploited in the run). Also, we have reported the data by providing different curves that are representative of iso-scaling in terms of both model complexity (total number of simulation objects) and underlying computing power (number of used CPU-cores). By the results we see how while the iso-scaling factor grows, the performance delivered by the simulator tends to stay stable or to get reduced by no more than 25%, which is an indication of how the simulation framework tends to scale well while hosting larger models onto larger computing platforms. This is true for both the considered deploy strategies, which further supports robustness of the deliverable performance in differentiated architectural setting.

Beyond the above results, indicating absolute execution speed and its variations (vs variations of the iso-scaling factor), we also report results for a comparison between parallel execution speed and serial execution speed. The latter parameter has been evaluated when running the same application level software (that was originally run on top of ROOT-Sim) on top of a calendar queue based [4] serial engine. The achieved speedup results for the case of block-deploy are shown in Figure 7, where the x-axis indicates the number of simulation kernel instances used (hence the number of CPU-cores exploited in the run). As for the above data, these speedup values still refer to iso-scaling in terms of both model complexity (total number of simulation objects) and underlying computing power (number of used CPU-cores). By the results we see how while the iso-scaling factor grows, the speedup delivered by the simulator increases linearly, which is an additional indication of how the simulation framework tends to scale well while hosting larger models onto larger computing platforms. Although not explicitly plotted, quite similar speedup values have been observed with the less favorable round-robin deploy.

Figure 7: Speedup wrt Serial Execution (Block-Deploy).

5. CONCLUSIONS

In this paper we have addressed the issue of simulating in-memory data grid platforms on high performance simulation engines. This is relevant when considering that these platforms are commonly adopted in cloud based environments, thus being good candidates for integration with dynamic reconfigurations strategies. In this context, our framework can provide supports for timely what-if analysis and validation of specific reconfiguration strategies. Also, the framework is flexible, in terms of ability to model differentiated data grid systems (e.g. characterized by different concurrency control schemes). Experimental data for validating the framework skeleton and for assessing the actual performance while supporting model execution are also reported.

6. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Operating Systems Review*, 41:159–174, 2007.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM*

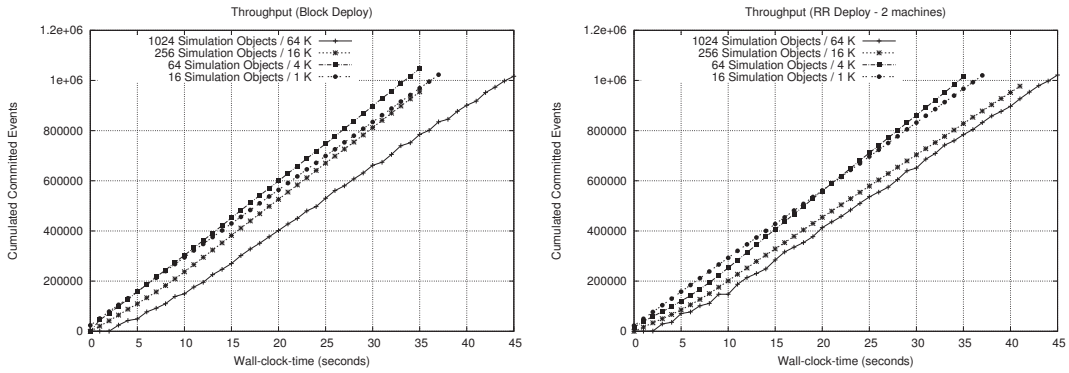

Figure 6: Event Rate Achieved by the Parallel Runs.

SIGMOD International Conference on Management of Data, SIGMOD '95, 1995.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[4] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31:1220–1227, October 1988.

[5] O. Dalle and E. Mancini. Integrated tools for the simulation analysis of peer-to-peer backup systems. In *5th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, pages 178–183, 2012.

[6] P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia. Auto-tuning of cloud-based in-memory transactional data grids via machine learning. In *Proc. 2nd IEEE International Symposium on Network Cloud Computing and Applications (NCCA)*, NCCA '12. IEEE Computer Society, 2012.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[8] D. Didona, P. Romano, S. Peluso, and F. Quaglia. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[9] R. M. Fujimoto and W. B. Campbel. Direct execution models of processor behavior and performance. In *Winter Simulation Conference*, pages 751–758, 1987.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, 1996.

[11] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, Oct. 2012.

[12] JBoss Infinispan. Infinispan Cache Mode. http://www.jboss.org/infinispan, 2011.

[13] S. Kounev, K. Bender, F. Brosig, N. Huber, and R. Okamoto. Automated simulation-based capacity planning for enterprise data fabrics. In *4th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, pages 27–36, 2011.

[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44, 2010.

[15] Oracle. Orache Coherence. http://www.oracle.com/technetwork/middleware/coherence/overview/index.html, 2011.

[16] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 0:455–465, 2012.

[17] A. Sulistio, U. Cibej, S. Venugopal, B. Robic, and R. Buyya. A toolkit for modelling and simulating data grids: an extension to gridsim. *Concurrency and Computation: Practice and Experience*, 20(13):1591–1609, 2008.

[18] TPC Council. TPC-C Benchmark, Revision 5.11. Feb. 2010.

[19] WMware. VMware vFabric GemFire. http://www.vmware.com/products/application-platform/vfabric-gemfire/overview.html.