A Middleware Level Active Replication Manager for High Performance HLA-based Simulations on SMP Systems

Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”

Abstract

In this paper we explore active replication in the context of advanced simulation systems, with the aim of improving the timeliness for the production of simulation output. Our proposal is framed by the High-Level-Architecture (HLA), i.e. the middleware based standard for interoperability of simulation packages. It results in the design and implementation of an Active Replication Management Layer (ARML) targeted to SMP computing systems, which supports the execution of (diversity-based) active replicas of a same simulation package in a totally transparent manner.

1 Introduction

In the recent years, several works have addressed the issue of enhancing the run-time effectiveness of simulation systems relying on the High-Level-Architecture (HLA) [10, 11], namely the middleware based standard for interoperability and integration (federation) of (heterogenous) simulation packages and applications. Among them, numerous solutions (see, e.g., [3, 15, 16, 18]) have been oriented to improving the run-time behavior of the underlying middleware component, i.e. the so called Run-Time-Infrastructure (RTI). Orthogonally to these solutions, in a previous work [17] we have provided a framework relying on software diversity and active replication of application level simulation components, which is aimed at tackling the impact of application level software on the timeliness of the output production. This framework identifies a middleware based architecture aimed at taking performance advantages from the “best instant responsiveness” among active replicas of a same component, implemented according to software diversity criteria.

Along the outlines provided by that replication framework, we propose in this paper the design and implementation of an Active Replication Management Layer (ARML), which transparently supports active replicas of a federate simulator, by showing them as a single logical entity. The implementation is targeted to SMP computing systems, namely COTS reference platforms for high performance computing applications, and has been based on C technology and standard POSIX APIs. Hence it results portable across any kind of POSIX compliant operating system (e.g. UNIX systems). Also, such an implementation has been tailored for being integrated with the well known Georgia Tech B-RTI package [7], even though the design principles underlying the implementation remain valid independently of the specific RTI to which replication handling facilities should be added. We also report the results of an experimental study in the context of HLA-based simulation of a mobile communication system, outlining the performance benefits from the active replication facilities offered by ARML.

The remainder of this paper is structured as follows. In Section 2 the replication framework in [17] is recalled. The design and implementation of the replication layer are proposed in Section 3. Related work is discussed in Section 4. Experimental results quantifying of the benefits from the proposed solution are reported in Section 5.

2 Recall on the Replication Framework

The active replication framework in [17] copes with an advanced, HLA-based simulation scenario, where instances of different simulators, namely federates in the HLA terminology, cooperate with each other through standard (call/callback) services provided by the underlying RTI. A federate can be seen as composed of (i) application specific simulation software, which includes all the modules and data structures implementing the specific simulation model associated with the federate itself, and (ii) general purpose simulation software, which instead includes all the modules and data structures used for typical, general purpose tasks (e.g. housekeeping tasks) in support of simulation applications. A federate is present within the federation as multiple diversity-based replicas, where software diversity can involve both application specific software (e.g. via multiple implementations of the same simulation model relying on different data structure organizations and/or different algorithms) and general purpose software (e.g. via different or differently parameterized libraries for housekeeping tasks). These replicas interact with a so called Active Replication Management Layer (ARML) - see Figure 1 - which exposes the same interface exposed by the RTI. At the same time, ARML interacts with the underlying RTI via that same interface, and has the following tasks to perform:

A. It intercepts all the instances of calls to a given RTI ser-
service performed by the different federate replicas, and forwards a single one of those calls to the underlying RTI. The forwarded call is the fastest one issued by the overlying replicas.

B. As soon as the RTI returns to ARML for a previously issued call, ARML delivers the return statement (and the return value, if any) to all the overlying replicas. If some federate replica has not yet executed the call to the corresponding RTI service (this might happen because the replicas execute asynchronously with each other within the active replication scheme), ARML needs to keep the return value buffered until that call is issued, and then immediately returns with the established return value to that federate replica.

In other words, ARML takes all the streams of calls to RTI services, each from a different replica, and builds a single stream including, for each specific service call, the corresponding instance coming for first among all the streams. In this way, the calls to RTI services associated with the stream provided by ARML follow a timing faster than what would be obtained in case of a single federate instance. This can provide advantages for the run-time effectiveness of the whole federation, whose speed of advancement in simulation time is affected by the timeliness in the invocation of specific RTI services, e.g. Time Management services, by the overlying federate simulators. A scheme similar to the one described in points A and B is adopted by ARML for handling the callbacks possibly issued by the underlying RTI to the overlying application. In particular:

C. ARML intercepts each callback from the RTI and delivers it to all the overlying replicas.

D. In case one of those replicas cannot yet accept the callback (e.g. because, acting asynchronously, it has not yet executed all the RTI calls preceding the delivery of that callback), ARML simply delays the callback execution on that replica until it is ready to accept it.

As pointed out in [17], there are some requirements which must be satisfied for both the effectiveness and the correctness of the whole approach. Concerning the effectiveness, each replica and the RTI need to be executed in real parallelism. This is because the RTI must be able to process requests according to an interleaved stream determined by ARML via the selection of requests from one or another replica, depending on instant responsiveness of each of those replicas. Hence it must be able to proceed in parallel with all of the involved replicas. At the same time, each replica must not affect the execution speed of the other replicas and of the RTI due to resource (e.g. CPU) contention. Concerning the correctness of the approach, all the federate replicas need to be Piece-Wise-Deterministic (PWD), with the meaning that they must exhibit the same trajectory for what concerns the state of the simulated component, and the same external interactions, under the same input conditions (e.g. the same callbacks from the RTI).

Anyway, it is important to remark that the PWD assumption is not a relevant limitation in the context of simulation systems, especially when considering that simulation software mostly rely on (i) pseudo-randomization, which, once fixed the corresponding seeds, determines well established computation paths, or (ii) pre-sampling, according to which random number generators are sampled in advance (see, e.g., [14]), hence all the replicas can use a same pre-sampled sequence along a computation path, or even (iii) traces collected from, e.g., logs, which can be made available to all the replicas. Additionally, it is usual that simulation software implements the representation of the state of the simulated component in a way semantically independent of any non-deterministic behavior of the underlying computing platform, i.e. the underlying Hardware and Operating System. In particular, the state representation is, in general, semantically independent of, e.g., specific memory addresses reserved for the corresponding data structures.

3 An Implementation for SMP Systems

In this section we present an implementation of ARML we have designed and developed for SMP computing systems. Although most of the design concepts we have used are independent of the specific underlying RTI package, the implementation itself is tailored for integration with the Georgia Tech B-RTI package [7]. For this reason, we initially propose a brief overview of B-RTI. Then we enter the details of the main targets and issues involved in the development of ARML, and provide relevant information related to the implementation.

3.1 Overview of B-RTI

B-RTI offers three classes of basic services (see Table 1). Declaration Management Services support the creation and the publication/subscription of classes of objects within the federation. The types of the parameters used by these services are mostly integer values (as an example, RTI_ObjInstanceDesignator is a redefinition of long) or memory addresses of either strings (class names) or other data structures. Specifically, RTI_ObjClassDesignator is a pointer to a data structure maintained by B-RTI which records information related
to a specific object class. Also, MCASTWhereProc is a pointer to an application level (i.e. federate level) function which must be used by B-RTI for reserving memory space for buffering the incoming messages associated with the updates of objects of a class subscribed by the federate (these messages are multicasted to all the federates subscribing that class). Object Management Services support the update of attributes of an object instance. RTIUpdateAttributeValue() can be invoked by the owner of an object instance, while ReflectAttributeValue() is a callback which can be issued by B-RTI to the federates which subscribed the corresponding class in order to make them reflect changes in the object state. RTI_Retract() is a service for retracting (i.e. undoing) the delivery of an issued object update and RequestRetraction() is the corresponding callback for finalizing the undoing at the application level. The types of the parameters used by these services are int or double (EventRetractionHandle is a numerical code associated with the message used for communicating the object update at the destination federate, and TM_Time is a redefinition of double) or memory addresses of either object related data structures maintained by B-RTI (i.e. RTI_ObjInstanceDesignator) or B-RTI managed messages (i.e. struct MsgS *). Time Management Services support synchronization among the federates. TimeAdvanceGrant() is a callback used by B-RTI to notify a safe simulation time for the federate. The other services are used by the federate for setting/getting the current lookahead value and for asking both the delivery of incoming messages up to a given simulation time and the possibility to advance the local simulation clock to that time. The parameters/return-values used by these services are all of type double (i.e. of type TM_Time).

There is a final observation, the tick service triggering the delivery of all the pending callbacks is supported via void BRTI_Tick(void), which takes no parameter and returns no value. Also, the function void RTI_Init(int, char **), with classical argument number and argument vector parameters, is used to set up the B-RTI for federated execution.

### 3.2 ARML Design Concepts

As pointed out in Section 2, the requirement for the effectiveness of the active replication framework depicted in [17] is that the replicas of the federate simulator and the underlying RTI must be executed in real parallelism. In the design of ARML we present in this paper, we satisfy this requirement by maintaining complete transparency for what concerns the presence of the ARML layer. This means that both the federate and the B-RTI must undergo no modification in order to be integrated with the ARML layer (e.g. no renaming of the federate main function and no addition of mechanisms for global data protection against critical races within the replication scheme, such as encapsulation). To achieve this objective, we have decided to organize ARML into two independent C software modules, namely federate_replication_manager and RTI_replication_manager, to be linked to the federate and to the B-RTI, respectively, and to let the federate replicas and the B-RTI run as separate UNIX processes on the SMP system. federate_replication_manager provides to the federate the same service interface as the one offered by B-RTI, and requires from the federate the corresponding callback interface, thus simulating the presence of the B-RTI code within that UNIX process. RTI_replication_manager uses that same B-RTI service interface and offers to the B-RTI the corresponding callback interface. It also contains a main function allowing the independent activation of B-RTI within a separate UNIX process. With such an organization, the invocation of calls to B-RTI services, or callbacks to the federate code, cannot take place by passing parameters within the stack or within CPU registers. This is due to separation of the contexts of the different processes. To solve this problem, we have decided to support the interaction between federate_replication_manager and RTI_replication_manager via an ad-hoc mechanism based on a shared memory buffer (see Figure 2).

Specifically, a service call issued by the federate is intercepted by federate_replication_manager and is then translated to a write operation into the shared memory buffer so that the corresponding information can be read by RTI_replication_manager and the corresponding B-RTI service can be correctly invoked. At the same time, when the service call returns to RTI_replication_manager, this module provides the service output to federate_replication_manager (again via the shared memory buffer), which then provides it back to the federate. Similarly, each callback issued by B-RTI towards the federate is intercepted by RTI_replication_manager and results in a write operation into the shared memory buffer so to communicate the corresponding information to federate_replication_manager, which then really invokes the corresponding callback and returns the callback output to RTI_replication_manager again via
shared memory. This output is finally returned by $RTI_{replication\_manager}$ to B-RTI.

Keeping in mind that, in general settings, the granularity of simulation events might be on the order of few microseconds (or even less), there are two fundamental objectives to be pursued in order to keep the overhead for the interaction of the different UNIX processes via the shared memory buffer at minimal (or almost null) levels:

**Minimal Number of Data Copies.** We should allow the interaction through the minimal amount of data write operations within the shared memory buffer. This is important especially when parameters/return-values associated with call/callback to/from B-RTI involve a non-minimal amount of bytes. This objective means in practice that each call to B-RTI services from the different $federate_{replication\_manager}$ instances should be issued to $RTI_{replication\_manager}$ by having the corresponding information written only once into the shared buffer, and the return value should be made available to all the replicas by writing it only once into the shared buffer. A similar reasoning can be made for the treatment of callbacks issued by B-RTI. Specifically, data associated with a callback should be made available to all the federate replicas by writing them only once into the shared buffer. The same should happen for the callback return value, i.e. only one among the different $federate_{replication\_manager}$ instances should perform the write operation of the callback return value into the shared buffer.

**No Kernel Level Involvement.** Write/read operations into the shared memory buffer, and the related supports for critical sections, should avoid the execution of kernel level modules. This would allow avoiding the overhead due to system calls (e.g. the cost of saving user level CPU context into kernel level stack plus the cost of system call dispatching), and would allow avoiding any secondary overhead effect associated with process dispatching operations (e.g. in case of a previous event of block while in kernel mode) (1).

1 An early implementation [17] relying on a socket based approach to support the interaction between the replication management modules, has shown that the kernel mode overhead can be affordable only for coarse grain applications.

2 Higher resource consumption can be justified by scenarios for which the timeliness in the production of simulation output (e.g. real-time response from the simulation system) is the dominating factor, and also by considering that, nowadays, high power computing systems, like SMP architectures, have become relatively cheap COTS platforms.

### 3.3 Shared Buffer Data Structures

We have developed an implementation of the data structures maintained into the shared memory buffer based on:

(i) Critical sections protected by application level spinlocks to address the requirement of no kernel level involvement while handling the interaction between $federate_{replication\_manager}$ and $RTI_{replication\_manager}$. To this purpose we have used the facilities (e.g. atomic test and set) offered by the `asm/atomic.h` header.

(ii) Sequence number based mechanisms to address the issue of minimal number of data copies for the interaction between those same modules.

Although the spinlocking approach might originate higher CPU consumption compared to a blocking approach based on kernel level mutexes/semaphores, spinlocking is in practice a viable solution each time we do not have particular constraints on resource availability. This is in the spirit of our approach, which is oriented to increasing the simulation system execution speed possibly at the expense of a higher amount of resource usage due to the active replication scheme (2). Additionally, as also recently shown in [5], application level spinlocking reveals an effective alternative to system level blocking approaches while handling critical sections when the target is application level execution speed on SMP systems.

Actually, some call/callback parameters identify pointed objects to be accessed by both the B-RTI and the federate. This is the case of `struct Mag5*` and `char*` parameters. Such a sharing of memory areas needs to be supported within ARML via the shared memory buffer used for the interaction between $federate_{replication\_manager}$ and $RTI_{replication\_manager}$. To cope with this issue, we have adopted within ARML a classical pack/unpack technique, based on linearizing all the call/callback parameters, including the pointed objects (i.e. pointed messages and strings), and packing them into a memory region within the shared memory buffer. The basic data structure used for the packing operation is represented in Figure 3. The `callback` field indicates whether the packed information is associated with a call to B-RTI services or a callback from B-RTI. The `code` field identifies the numerical fields.

---

![Figure 2. Interaction via Shared Memory.](image)

---

![Figure 3. Call/Callback Descriptors.](image)
cal code of the service or callback (this is used to perform correct binding of calls and callbacks in the different address spaces). Finally, the arguments field stores the linearized parameters associated with the call or callback. The same call_callback_descriptor data structure is used also for packing the return value of a service call, notified from RTI_replication_manager to federate_replication_manager and the return value of a callback notified from federate_replication_manager to RTI_replication_manager. In such a case, the arguments field is used to pack that return value.

By exploiting the call_callback_descriptor data structure we have built within the shared memory buffer three different communication channels. A first channel, which we refer to as Service-Call-Channel (SCC) simply consists of a data structure formed by a call_callback_descriptor plus an additional field of type long named sequence_number. A graphical representation of SCC is provided in Figure 4.A. This channel is used by the different replicas of federate_replication_manager to notify to RTI_replication_manager calls to B-RTI services. The field sequence_number is used to implement the minimal number of data copies requirement as follows. When whichever instance of federate_replication_manager wants to notify to RTI_replication_manager a new call to a B-RTI service, it associates with the corresponding call_callback_descriptor a monotonically increasing sequence number x, which indicates the ordering position of the service call within the stream of calls from each federate replica. Then, the call_callback_descriptor is really written on SCC, i.e. it is posted on the channel, only in case its sequence number x is greater than the value of the sequence_number field within the channel. In the positive case, the sequence_number field is updated accordingly. If the sequence number x is lower than or equal to the value of the sequence_number field already stored within the channel, it means that some other replica has been faster in issuing that same service call (i.e. the service call with that same sequence number) to the underlying B-RTI. Hence the present call needs to be filtered out from the stream of calls really issued by ARML to the B-RTI. Filtering out this call simply means that we avoid copying the corresponding descriptor within SCC. In this way, each service call is issued by having the corresponding information written only once into the shared memory buffer.

We note that the implementation of SCC based on a single slot for call descriptors is correct since only one service request to the underlying B-RTI can be standing at any time (no federate can issue a service call before the last issued one has already returned). The same is not true when considering the return value of a service call and also callbacks issued by the B-RTI via RTI_replication_manager towards the overlying replicas. Specifically, when the return value of a service call is provided by B-RTI, some overlying replicas might be not yet ready to accept that return value, which needs to be kept buffered by ARML until all the replicas have received it (see Section 2). Similarly, the information written into the shared memory buffer, and associated with each call (e.g. callback parameters), needs to be buffered until all the replicas have read it. To achieve such a buffering objective we have implemented a second communication channel within the shared memory buffer, which we refer to as Call-Return-and-Callback-Channel (CRCC). A graphical representation of CRCC is provided in Figure 4.B. This channel is used by RTI_replication_manager to notify the return value of each service call and also the information associated with each callback to all the instances of federate_replication_manager. CRCC is implemented as a circular buffer of data structures maintaining a call_callback_descriptor field, and an additional field of type int named pending_replicas, which is treated as a bit-mask indicating which replicas have not yet read the service call return value or the callback information. Each time RTI_replication_manager needs to notify the return value of a service call, or needs to issue a callback to the overlying replicas, it writes the corresponding descriptor into the first available slot of CRCC and then sets the bits corresponding to the currently active replicas into the pending_replicas bit-mask. This indicates that upon the insertion of the information into that slot, no one among the overlying replicas has yet read it. (Note that in this way, the descriptor is made available to all the replicas by writing it only once into CRCC, thus matching the minimal number of data copies requirement.) When an instance of federate_replication_manager reads the service call return value or the callback information from that slot, it resets its corresponding bit into the pending_replicas bit-mask. When all the replicas have read the information, the pending_replicas bit-mask assumes the value zero, and the corresponding slot within CRCC gets available again according to the circular buffer policy. In order to allow all the replicas to read the information within all the slots in correct order, each instance of federate_replication_manager implements the circular buffer policy by maintaining its own read index. We note that the read/write policy adopted for the slots within CRCC ensures that the delivery order of service returns and callbacks at each replica matches the order in which service returns and callbacks are issued by the underlying B-RTI. This allows eventual consistency of the states of all the replicas under the PWD assumption (see Section 2).

The third channel we have implemented within the shared buffer, namely Callback-Return-Channel (CRC), is used by the federate replicas to provide the return values of callbacks to the B-RTI. This channel consists of a data structure identical to the one implementing SCC. Specifically, it consists of a record formed by a call_callback_descriptor field used for packing
the callback return value, and of a sequence_number field, which is used according to the same policy adopted for the SCC channel. More precisely, each return to a callback is associated by whichever instance of federate_replication_manager with a sequence number, again indicating the ordering position of the callback return within the stream of return values from each replica. This sequence number is used to determine whether a callback return is “on time”, or some other replica has already returned to B-RTI for that callback. In particular, the callback return value is really written by federate_replication_manager within CRC only in case the associated sequence number is greater than the value of the sequence_number field kept by CRC (in this case this field is updated to the callback return sequence number). Otherwise, federate_replication_manager simply skips performing the copy of the callback return value within CRC. According to this solution, each callback issued by B-RTI will result in a single write operation of the callback return value within shared memory, which matches the minimal number of data copies requirement.

Wait statements on any channel (e.g. when federate_replication_manager needs to wait for the return value of a service call on CRCC, or when RTI_replication_manager needs to wait for a service call on SCC or for the return value of a callback on CRC) have been implemented by exploiting the same application level spinlocks used for solving critical sections on data structures within the shared buffer.

Looking back at the call/callback interface presented in Section 3.1, there is a single type of parameter whose passage cannot be straightforwardly solved by using the packing/unpacking approach previously described, based on the call_callback_descriptor data structure. This type is MCAST_whereProc, namely the function pointer indicating to the B-RTI which is the “federate level” procedure for identifying memory buffers for incoming messages associated with the updates of subscribed objects. Specifically, according to the B-RTI interface, the federate must specify the value of this function pointer in order to identify the function address in the address space of the whole application (federate plus B-RTI). However, when ARML is used, the corresponding function should be executed by B-RTI in a separate address space. To cope with this issue without the need for managing linker symbols (and binding them on B-RTI service parameters), we have decided to let RTI_replication_manager overrule the MCAST_whereProc parameter with a function pointer value defined by RTI_replication_manager itself, which identifies a memory reserving procedure valid in the address space of B-RTI (recall that RTI_replication_manager and the B-RTI are linked together to generate a same executable).

4 Related Work

Concerning solutions specifically oriented to enhancing the performance of simulation systems, a kind of replication approach known as cloning has been proposed in [4, 8, 9]. Its aim is to allow fast exploration of multiple execution paths due to sharing of portions of the computation on different paths. Our approach differs in nature from (and hence reveals orthogonal to) these proposals since we aim at increasing the execution speed on each single execution path thanks to the presence of several software diversity-based replicas of a same “simulation entity” executing that same path in parallel, according to an active replication scheme. Always in the context of simulation systems, replication has been exploited by running multiple copies of a same simulation program with different input parameters (see, e.g., [1, 13]), sometimes even in interleaved mode on the same hardware [2] so to further improve resource usage in the presence of interleaving between computing and communication phases. Differently from (and orthogonally to) our approach, this type of replication is not aimed at accelerating the execution speed on each single run, but is aimed at efficiently providing a set of output samples (from differently parameterized runs) for statistical inference.

Our replication approach is also related to classical Parallel Discrete Event Simulation (PDES) techniques [6] since, like PDES, it attempts to exploit an increased amount of computing resources to speedup the simulation execution. However, a key difference between our approach and conventional PDES techniques is that they require explicit ad-hoc (re-)programming of the simulation package in order to embed within it either space or time partitioning of the simulation tasks across multiple CPUs. Instead, our replication approach is oriented to transparency in the exploitation of increased computing power (i.e. multiple CPUs on an SMP system) in the context of integration of already existing (legacy) simulation packages via a middleware approach relying on HLA.

5 Experimental Evaluation

5.1 The Case Study

The application level code we have used in this experimental study is a parameterizable simulation software of
mobile Personal Communication Systems (PCS). The simulator can be parameterized so to simulate the PCS either at the channel abstraction level, in this case power regulation is not explicitly simulated, or at a higher level of details, in this case fading effects and channel interference are explicitly modeled so to perform statistical inferences on the signal strength (or signal quality) based on the Signal-to-Interference Ratio (SIR) [12]. Actually, the case with no power regulation represents a finer grain configuration, where simulation events (e.g., start call or handoff events) only need to manage channel availability at the destination cell to determine whether the call is accepted, blocked or dropped. Instead, the case with power regulation represents a coarser grain configuration, where simulation events require costly (re-)calculation of fading and interference on active channels. In the experiments we have simulated a coverage area with 100 cells, each managing 100 channels, and we have considered both the previous fine and coarse grain configurations. The supported mobility model is random way-point, and the mobile devices involved in on-going calls belong to three different classes simulating, respectively, users residing in some buildings, users walking along the streets and users travelling by some vehicle.

We have federated this simulator with an external workload generator based on traces, which simply accesses a log of information to originate TimestampOrdered (TSO) interactions to be delivered at the PCS simulator side. Each interaction schedules the arrival of a call, and the interaction message includes any information required by the PCS simulator to simulate the call itself (e.g. the call duration).

The PCS simulator uses the optimistic Time Management interface offered by the underlying RTI to synchronize with the external workload generator, and the software diversity approach we employ deals with obtaining two diversity-based replicas of the PCS simulator thanks to different parameterization of the library used for checkpointing/recovery of the simulation model state (in other words we exploit diversity at the level of general purpose software supporting housekeeping tasks). Specifically, the optimistic Time Management interface specified by HLA requires state checkpointing/recovery modules to be included at the level of the federate. Hence, checkpointing/recovery actions of the federate state need to be controlled at the federate level via proper software modules. In our case, these modules adopt a periodic approach for logging state information required to perform correct state recovery in case of violations on TSO interactions with the workload generator, which requires the reply (namely coasting forward) of the events in between a checkpoint and the causality violation, if any, when a recovery phase occurs. Also, the diversity is achieved by parameterizing these modules in order to have the two PCS simulator replicas take state log each \( K \) events, with an out-of-phase of \( K/2 \) events. If \( K = 1 \), the two replicas exhibit the same behavior by taking state log at each event. Instead, for any value of \( K \) larger than one, they behave differently by logging the state at different points in simulation time. This allows for different instant responsiveness of the two replicas during both forward computation and also in a rollback phase (due to different coasting forward length when a TSO violation occurs at a given simulation time point). In other words, the checkpointing/recovery modules are parameterized in order to provide a speed inversion on the two PCS simulator replicas while handling state record and state recovery operations. Note that this is not an artifact of our experimental settings, but is exactly in the spirit of software diversity concepts related to the active replication framework (e.g., different parameterization for different instant responsiveness while executing some housekeeping tasks).

In order to evaluate the benefits from replication, such configuration has been compared with a standard configuration employing no replication, formed by a single instance of the PCS simulator and by the workload generator, both directly interacting with the underlying B-RTI. However, in order to also evaluate the overhead by ARML, we have considered an additional configuration in which there is a single instance of the PCS simulator (i.e. replication is not employed), which interacts with the underlying B-RTI via ARML. In other words, with this configuration we introduce the overhead due to ARML but do not really exploit active replication for performance purposes.

5.2 Test Settings

All the runs have been carried-out on an SMP machine equipped with 4 Xeon CPUs (2.0 GHz) and 4 GB of RAM memory, running LINUX (kernel 2.6). We note that 4 CPUs suffice to originate a situation of no CPU contention among the involved components, i.e. the two active replicas of the PCS federate, their underlying B-RTI instance and the external workload generator (recall that, while the two replicas of the PCS federate and their underlying B-RTI instance run as separate processes due to the presence of ARML, the workload generator and its underlying B-RTI instance run within a same process). Additionally, we have verified that the size of the involved processes do not cause RAM contention and swapping phenomena, which, as well as CPU contention, would prevent a significative evaluation of the potential of the proposed replication approach.

5.3 Results

In Figure 5 we report the execution speed of the federation, evaluated in terms of simulated time units per wall-clock time unit, for the two investigated PCS test cases (fine grain, i.e. with no power regulation, and coarse grain, i.e. with power regulation) while varying the checkpoint interval \( K \). Each value reported in the plots is the average over a number of samples that ensures a confidence interval of 10% around the mean at the 95% confidence level.

By the results the overhead due to ARML is kept negligible even for the fine grain configuration, as confirmed by the values related to the case of ARML used with a single
instance of the overlying PCS simulator (labelled ARML-no-replication). In fact, these performance data coincide in practice with those obtained when ARML is not employed. Also, the configuration with replication is able to achieve a better balance of the checkpointing/recovery costs vs the parameter $K$ thanks to the out-of-phase placement of checkpoints, which also allows for reducing the impact of rollback costs (i.e. coasting forward latencies) on the responsiveness of the PCS application, externally seen by the B-RTI. This allows the configuration with replication to exhibit execution speed up to 24% (resp. 11%) better than the top speed achieved with no replication vs $K$ for the fine grain (resp. coarse grain) configuration. Actually, the reduced gain in case of coarse grain events is not due to reduced effectiveness of the replication framework for applications with large CPU requirements for simulation events, but simply to the fact that, for this specific test case, the varied parameter (i.e. the checkpoint interval $K$) has less impact on the run-time behavior in case of coarse grain events (since the relative checkpointing overhead is limited) when compared to the impact it has in the fine grain configuration.

To further support the validity of the results, we have also performed an execution in which the PCS simulator is run with no replication and interfacing the underlying B-RTI via the conservative Time Management services. The observed execution speed was on the order of 34.0 (resp. 1.05) simulation time units per wall-clock time unit for the fine grain (resp. coarse grain) configuration, which indicates that the reported data refer to a situation in which optimistic synchronization is effective.

6 Summary
In this paper we have presented a middleware level active replication manager for HLA-based simulations on SMP computing systems. This layer is able to transparently handle (diversity-based) active replicas of a federate simulator, in order to improve the timeliness for the production of simulation output. We have also presented the results of an experimental study on a mobile simulation application, demonstrating the effectiveness of our proposal in improving the execution speed of the federated simulation system.

References


