Enhancing the Performance of HLA-Based Simulation Systems via Software Diversity and Active Replication

Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
quaglia@dis.uniroma1.it

Abstract

In this paper we explore active replication based on software diversity for improving the responsiveness of simulation systems. Our proposal is framed by the High-Level-Architecture (HLA), namely the emerging standard for interoperability of simulation packages, and results in the design and implementation of an Active Replication Management Layer (ARML), which supports the execution of multiple software diversity-based replicas of a same simulator in a totally transparent manner. Beyond presenting the replication framework and the design/implementation of ARML, we also report the results of an experimental evaluation on a case study, quantifying the benefits from our proposal in terms of execution speed.

1 Introduction

In this paper we explore software diversity and active replication in the context of advanced simulation systems with the aim at improving the timeliness in the production of simulation results. This is done by exploiting the best instant responsiveness among the different replicas during different phases of the simulation run.

The replication approach we propose is framed by the High Level Architecture (HLA) [10, 11], namely the emerging standard for interoperability of simulation systems and applications. This standard defines a set of middleware services to be offered by a so called Run-Time-Infrastructure (RTI), aimed at supporting complex simulation applications resulting from the integration of (pre-)existing simulation packages. Hence, HLA appears as a natural candidate context for the development of middleware facilities able to transparently handle the replication of application level simulation components in an advanced simulation scenario.

An important aspect related to our proposal is that software diversity does not necessarily mean having different implementations of a same simulation entity based on, e.g., different types of data structures and algorithms. It might simply mean employing different (or differently parameterized) third party libraries in support of simulation related, general purpose, application level tasks in order to originate diversity-based replicas exhibiting different instant responsiveness. Hence, our replication approach can provide real run-time advantages with no (or minimal) effort from teams of application programmers, who are not necessarily required to provide diversity-based implementations of a same application level simulation component. This means in practice following a kind of "Opportunistic N-Version Programming" such as the one followed by [17] in the context of replication in support of fault tolerance.

Typically, an active replication scheme requires a higher amount of computing resources to achieve its objective. This also occurs for our approach, where the execution of the same simulation path needs to be carried out in a really parallel manner by the different software diversity-based replicas. However, such a higher resource consumption can be justified by scenarios for which the timeliness in the production of simulation outputs (e.g. real-time response from the simulation system) is the dominating factor.

The implementation of an Active Replication Management Layer (ARML), which transparently supports software diversity-based replicas by showing them as a single logical entity, is also presented. It has been based on `C` technology and standard UNIX APIs, therefore resulting portable across any king of UNIX system. Also, such an implementation has been tailored for integration with the Georgia Tech B-RTI package [7], even though the underlying design principles remain valid independently of the specific RTI to which replication handling facilities should be added.

Beyond providing the framework for software diversity-based replication, and describing the design and implementation of ARML, we also report experimental results in the context of HLA based simulation of a GSM system outlining the performance benefits from our proposal.

The remainder of this paper is structured as follows. In Section 2 the replication framework is described. The design and implementation of the replication layer to be integrated with the RTI are proposed in Section 3. Related work is discussed in Section 4. Experimental results are reported in Section 5.

1-4244-0054-6/06/$20.00

2 The Replication Framework

As pointed out, we are interested in an advanced, HLA-based simulation scenario, where instances of different simulators, namely federates in the HLA terminology, cooperate with each other through the services provided by the underlying RTI. Actually, a federate is typically composed of (i) application speci c simulation software, which includes all the modules and data structures implementing the simulation model, and (ii) general purpose simulation software, which instead includes all the modules and data structures used for typical, general purpose tasks in support of simulation applications. Examples of general purpose tasks include the maintenance of calendar queues and also checkpointing/recovery of the federate simulator state in support of optimistic synchronization. In such a context, our view of software diversity can be expressed according to the following two diversity criteria:

Application-Specific-Software-Diversity (ASSD). This type of diversity is achieved in case the simulation model is implemented multiple times according to (i) different data structure organizations (e.g. static pre-allocated memory vs dynamically allocated one) and/or (ii) different algorithms. It is also achieved in case a given implementation can be parameterized so to impact the run-time behavior of the involved modules under the same input conditions. As an example, the software might make use of a mixture of pre-allocated memory and dynamically allocated memory, used in case the pre-allocated one gets exhausted at a given point of the execution. In such a case, the size of the pre-allocated memory chunks might impact the run-time overhead for possible allocation/release of dynamic memory (e.g. by determining the amount of allocation/release operations per time unit).

General-Purpose-Software-Diversity (GPSD). This type of software diversity deals with the case of multiple libraries with the same interface ([1]), but with different internals, available for general purpose tasks in support of simulation systems, so that application speci c software can interface whichever of those libraries to achieve different instances of a same federate possibly exhibiting different run-time behaviors under the same input conditions (see, e.g., [4, 18] for comparisons between different algorithms/implementations of calendar queues and [5, 14, 16, 19] for comparisons between different algorithms/implementations of checkpointing). As for ASSD, GPSD can be also achieved in case a given library for general purpose tasks can be parameterized so to impact its run-time behavior under the same input conditions (e.g. the size of a hash-with-bucket table implementing an event list can determine the level of collision, and so the information access time).

[1] We recall that in most cases the same interface for a set of different libraries can be achieved by a simple wrapping approach.

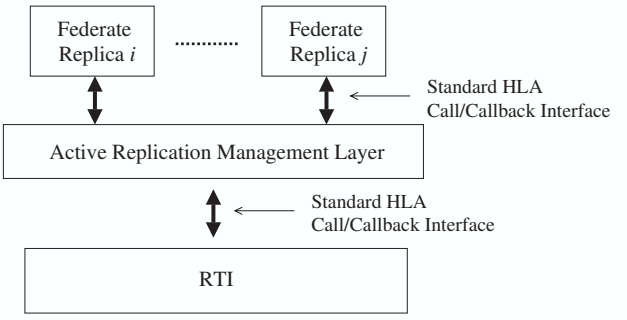

Figure 1. Software Architecture with the Active Replication Management Layer.

Once de ned ASSD and GPSD, the active replication approach we propose can be schematized as in Figure 1. Each federate is present within the whole federation as multiple diversity-based replicas. These replicas interact with a so called Active Replication Management Layer (ARML) which exposes the same call/callback interface exposed by the RTI. At the same time, ARML interacts with the underlying RTI via that same interface. ARML has the following tasks to perform:

A. It intercepts all the instances of calls to a given RTI service performed by the different federate replicas, and forwards a single one of these 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, ARML keeps 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 service call, the corresponding instance coming for rst among all the streams. A scheme similar to the one described in points A and B is adopted by ARML for handling the callbacks speci ed by the HLA standard [10, 11]. In particular, ARML intercepts each callback from the RTI and delivers it to all the overlying replicas. In case some of these replicas cannot yet accept the callback (since, 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.

Given overlying replicas exhibiting different instant responsiveness due to software diversity (ASSD and/or GPSD), the calls/callbacks to/from RTI associated with the stream provided by ARML are expected to follow a timing faster than what would be obtained in case of a single federate instance. This can provide benefits for the whole federation of simulators, whose run-time behavior depends on the timely invocation of RTI services (e.g. requests to advance in simulation time) of each involved federate.

Actually, there are some requirements which must be satisfied for both the effectiveness and the correctness of the whole approach. Concerning the effectiveness, we need to execute each replica and the RTI in real concurrency (e.g. as different threads or processes on an SMP computing system). 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, we require all the federate replicas to be Piece-Wise-Deterministic (PWD), with the meaning that they must exhibit the same trajectory for what concerns the state of the simulated object, and the same external interactions, under the same input conditions (e.g. the same callbacks from the RTI). In general, this is not a relevant limitation, 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., [13]), 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 object 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 Replication Layer Implementation

Although most of the design concepts we have used for engineering and developing an instance of ARML are independent of the specific underlying RTI package, the implementation is tailored for the Georgia Tech B-RTI package [7]. For this reason, we propose in this section a brief overview of B-RTI, which will also form the basis for understanding specific implementation choices.

3.1 Overview of B-RTI

B-RTI offers the following three classes of basic services in support of federated simulations:

Declaration Management Services. This class entails the services listed in Table 1, which can be used for creating classes of objects within the federation, and for publishing/subscribing classes of objects in order to allow interaction among different federates according to the publish/subscribe model adopted by HLA. For our purposes, it is important to note that 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 (object 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, `MCAST_WhereProc` 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 multi-casted to all the federates subscribing that class).

Object Management Services. This class entails the services listed in Table 2, which can be used for updating attributes of an object instance. The service named `RTI_UpdateAttributeValues()` can be invoked by the owner of the object instance, i.e. the federate that published the corresponding object class and created that object instance. `ReflectAttributeValues()` is a callback which can be issued by B-RTI to the federates which subscribed that class in order to make them reflect changes in the object state. `RTI_Retract()` is a service for undoing the delivery of an already issued object update, and `RequestRetraction()` is the corresponding callback for finalizing the undoing at the application level. For our purposes, it is important to note that the parameters used by these services are of type `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. This class entails the services listed in Table 3, which can be used by the federate for synchronization purposes. Among them, the unique callback is `TimeAdvanceGrant()`, which is used by B-RTI

Table 1. Declaration Management Services.

```
RTI_ObjClassDesignator RTI_GetObjClassHandle (char *)
RTI_ObjInstanceDesignator RTI_RegisterObjInstance (RTI_ObjClassDesignator)
void RTI_PublishObjClass (RTI_ObjClassDesignator)
void RTI_InitObjClassSubscription (RTI_ObjClassDesignator, MCAST_WhereProc, void *)
BOOLEAN RTI_IsClassSubscriptionInitialized (RTI_ObjClassDesignator)
void RTI_SubscribeObjClassAttributes (RTI_ObjClassDesignator)
RTI_ObjClassDesignator RTI_CreateClass(char *)
```

Table 2. Object Management Services.

```
EventRetractionHandle RTI_UpdateAttributeValues(RTI_ObjInstanceDesignator, struct MsgS *, long, long)
void ReflectAttributeValues(TM_Time, struct MsgS *,long, long)
void RTI_Retract(EventRetractionHandle)
void RequestRetraction(EventRetractionHandle)
```

Table 3. Time Management Services.

```
void TimeAdvanceGrant (TM_Time)
void RTI_NextEventRequest (TM_Time)
void RTI_TimeAdvanceRequest (TM_Time)
void RTI_FlushQueueRequest(TM_Time)
void RTI_SetLookAhead(TM_Time)
TM_Time RTI_GetLookAhead(void)
```

to notify a safe simulation time for the federate. All 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. For our purposes it is important to note that the parameters/return-values used by these services are all of type `double` (i.e. of type `TM_Time`).

There is a nal observation, the tick service triggering the delivery of all the pending callbacks is supported via the function `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

Our ARML design and implementation are oriented to portability on whichever UNIX system. Also, since one basic requirement for the effectiveness of our active replication strategy is the possibility to support real concurrency for the execution of the underlying RTI and the overlying federate replicas, we have decided to allow such a concurrency scenario independently of the nature of the underlying computing platform (e.g. SMP vs cluster based platforms). To this aim, we have organized ARML into two independent `C` software modules, namely $federate\_replication\_manager$ and $RTI\_replication\_manager$. The rst module must be linked to the federate code. It provides to the federate the same service interface as the one offered by B-RTI , and requires from the federate the corresponding callback interface (see Section 3.1). The second module must be linked to the B-RTI code. It uses that same B-RTI service interface and offers to the B-RTI the corresponding callback interface.

The interaction between these two independent modules takes place through standard socket API, with the meaning that a service call issued by the federate is intercepted by $federate\_replication\_manager$ and is then translated into a message sent to $RTI\_replication\_manager$ via sockets. Once received the message, $RTI\_replication\_manager$ issues that service call to the underlying B-RTI. When the service call returns, $RTI\_replication\_manager$ provides the output via socket to $federate\_replication\_manager$, which then provides it back to the federate.

Similarly, each callback issued by B-RTI is intercepted by $RTI\_replication\_manager$ and results in a message sent via socket to $federate\_replication\_manager$, which then really invokes the corresponding callback and returns the callback output to $RTI\_replication\_manager$, again via socket. This output is nally returned by $RTI\_replication\_manager$ to B-RTI.

Compared to a classical federate/RTI interaction based on calls/callbacks exploiting the application stack or even CPU registers for handling parameters and return values, the additional operations associated with the interaction between $federate\_replication\_manager$ and $RTI\_replication\_manager$ via sockets cause some overhead. However, in case the federate and the underlying RTI are hosted by the same SMP machine, the approach of using sockets boils down to a few additional copies of parameters/return-values into kernel level memory, possibly plus process dispatching operations (e.g. in case of an event of block on incoming data on a socket). Both previous tasks are typically executed in a very effective manner by the kernel, hence the expected overhead is likely to be affordable unless for (very) ne grain applications. Limited overhead for a coarse grain application is actually con rmed by the experimental results we report in Section 5.

3.3 Main Data Structures and Flow Control

The call/callback B-RTI interface in Section 3.1 treats two types of pointer parameters. One type (e.g. `RTI_ObjClassDesignator`) includes pointers that are maintained by the federate, but not directly de-referenced

```
typedef struct _instance_descriptor{      /* descriptor of a call or callback */
   int  callback;                          /* is this the description of a callback? */
   long sequence_number;         /* for the construction of ARML stream of calls/callbacks to/from the BRTI */
   int  code;                             /* call/callback numerical code */
   char arguments[MAX_ARGUMENTS_SIZE];   /* call/callback arguments marshalled in a contiguous buffer */
} instance_descriptor;
```

Figure 2. Call/Callback Descriptors.

by the federate. They represent memory addresses of data structures maintained and accessed by B-RTI code. The other type (e.g. `struct MsgS *` and `char *`) includes instead pointers to memory areas that contain data accessible by both the B-RTI and the federate. For instance, when the `ReflectAttributeValues()` callback is issued, the `struct MsgS *` parameter points to a memory buffer where the message to be delivered to the federate has been placed by B-RTI. Hence both the federate and the underlying B-RTI access in practice the same memory area via that parameter. Such a sharing of memory buffers cannot be supported within ARML since the federate and B-RTI run as separate processes interacting via sockets. To tackle this problem, we have adopted a classical RPC like marshalling/unmarshalling technique, based on linearizing all the call/callback parameters, including the pointed objects (i.e. pointed messages and strings), and packing them into the message to be sent over the socket connection. However, compared to classical RPC, ARML is faced with the additional issues of creating a single stream of calls to B-RTI services from multiple streams associated with different replicas, and also of creating multiple callback streams towards all the replicas starting from a single callback stream coming from the underlying B-RTI. To tackle these additional issues, as well as classical marshalling/unmarshaling, each message sent via socket between $federate\_replication\_manager$ and $RTI\_replication\_manager$ is packed into the `instance_descriptor` data structure shown in Figure 2. The `callback` field indicates whether the message is associated with a call to B-RTI services or a callback from B-RTI. The `sequence_number` field indicates the ordering position of a service call within the stream of calls from each federate replica. The `code` field identifies the numerical code of the call 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 `instance_descriptor` data structure is used also for packing the reply to a service call, sent from $RTI\_replication\_manager$ to $federate\_replication\_manager$ and the reply to a callback sent from $federate\_replication\_manager$ to $RTI\_replication\_manager$. Is such a case, the `arguments` field is used to pack the call/callback return value.

Looking back at the call/callback interface presented in Section 3.1, there is a single parameter type whose passage cannot be straightforwardly solved by using the RPC like marshalling/unmarshalling approach previously described. This parameter type is `MCAST_WhereProc`, namely the function pointer indicating to the B-RTI which is the federate level procedure for reserving 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 $RTI\_replication\_manager$, 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).

For each managed federate replica, two socket connections between $federate\_replication\_manager$ and $RTI\_replication\_manager$ are installed. One connection, which we refer to as Primary Connection (P-Con) is used by $federate\_replication\_manager$ to send the `instance_descriptor` associated with a call to B-RTI services, and for receiving both (i) the return value of the call and (ii) the `instance_descriptor` associated with a callback. Specifically, when the federate issues a call to a service that does not trigger any callback, $federate\_replication\_manager$ fills in the corresponding `instance_descriptor`, sends it over P-Con and then waits for the reply, i.e. for the `instance_descriptor` associated with the service return. Upon its arrival, $federate\_replication\_manager$ simply returns to the overlying federate with the established return value. Instead, when the federate issues a call to a service that triggers pending callbacks, i.e. `BRTI_Tick()`, $federate\_replication\_manager$ be-

haves differently. It lls in and sends the corresponding instance_descriptor, and then waits on P-Con for either (i) the reply to BRTI_Tick() or (ii) the instance_descriptor associated with a callback. If the latter descriptor arrives, then the callback is invoked and the callback return value is sent back to $RTI\_replication\_manager$ via the additional socket channel, which we refer to as Secondary Connection (S-Con). Then $federate\_replication\_manager$ waits again on P-Con for either the reply to BRTI_Tick() or an additional callback descriptor.

$RTI\_replication\_manager$ uses a different thread for each managed federate replica. Each thread controls the P-Con associated with the corresponding replica according to the following scheme:

1. When an instance_descriptor arrives along P-Con, the thread checks whether the sequence number carried by the descriptor is greater than the maximum sequence number of service calls already issued to the underlying B-RTI.

2. In the positive instance, the incoming descriptor is associated with a service call which must be inserted in the stream of calls to the underlying B-RTI. Hence, the thread updates the maximum sequence number of issued calls and really executes the call to the underlying B-RTI service.

3. In the negative instance, the thread simply skips performing the service call and goes waiting again for an incoming instance_descriptor on P-Con.

If the steps in point 2 are executed, i.e. the service call is really performed by that thread instance, the return value must be communicated to all the replicas. Hence, the thread sends the instance_descriptor carrying the return value along all the P-Con channels towards the different federate instances. Actually, this means that the return value is implicitly kept buffered within socket level memory buffers until the corresponding federate replica requests it. According to this communication scheme for the return value, it is possible that, upon the issue of a call to a service which has been already executed by the B-RTI thanks to a faster request instance from a different replica, $federate\_replication\_manager$ might nd the reply immediately available on its P-Con channel. Hence, it can immediately return control to the federate.

As a nal observation, in case the service requested by the thread to the underlying B-RTI in point 2 is BRTI_Tick(), all the callbacks from the B-RTI will result in callback descriptors sent by that thread along all the P-Con channels towards the different federate instances. For each callback from B-RTI, the thread waits for the fastest callback reply across all the S-Con channels (recall that $federate\_replication\_manager$ sends the reply to a callback request on S-Con), and then returns control back to the B-RTI. All the slower callback replies from the other replicas are discarded according to the sequence number based mechanism already discussed.

4 Related Work

Among the solutions oriented to enhancing the performance of simulation systems, a kind of replication approach known as cloning has been proposed in [3, 8, 9]. The aim of this approach is to allow fast exploration of multiple execution paths due to sharing of portions of the computation on different paths. Our proposal differs in nature from this approach since we aim at increasing the execution speed on a single execution path just 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.

Replication has been also exploited by running multiple copies of a same simulation program with different input parameters (see, e.g., [1, 12]), 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 our approach, this type of replication is not aimed at accelerating the execution speed on each single run, but is aimed at efciently providing a set of output samples (from differently parameterized runs) for statistical inference.

Finally, our proposal is also related to classical Parallel Discrete Event Simulation (PDES) techniques [6] in that, like in our approach, PDES attempts to exploit an increased amount of computing resources to speedup the simulation execution. The main difference with 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 approach is oriented to transparency in the context of integration of (existing) simulation packages via a middleware approach.

5 Experimental Results

In this section we report experimental data for an evaluation of the effects of ARML. The application level code we have used in the experimental study is a parameterizable simulation software for GSM mobile systems. It can simulate GSM systems at a high level of detail by explicitly modeling large-scale fading (i.e. path loss), small-scale fading (i.e. Rayleigh fading) and channel interference so to perform statistical inferences on the signal strength (or signal quality) based on the Signal-to-Interference Ratio (SIR). The supported mobility model is random way-point. This model has the ability to capture different classes of mobile behaviors, ranging from pedestrian to vehicular behavior, by simply parameterizing the pause duration and

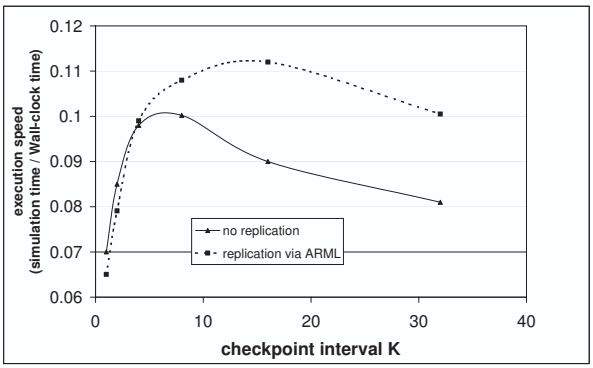

Figure 3. Simulation Execution Speed.

maximum movement speed according to a set of different values. Large scale-fading is simulated at non- xed time steps. Speci cally, path loss is (re-)evaluated on the basis of relevant events associated with the effects of the mobility model, namely the arrival of a mobile in a way-point. Small scale fading can be simulated at either xed time-steps or at non- xed ones. In the latter case, Rayleigh fading is evaluated at simulation time points associated with relevant events for both mobility (e.g. the arrival of a mobile in a way-point) and the call arrival pattern (e.g. the start/end of a call). The simulator includes modules for pseudo-random generation of call start/end patterns, and can also be federated with an external workload generator. For what concerns time management, the simulator uses the optimistic interface offered by the underlying RTI, and adopts a periodic approach for logging state information required to perform correct state recovery in case of violations on TimeStamp Ordered (TSO) interactions with external federates. In the experiments, we have federated this simulator with an external workload generator based on traces, which simply accesses a log of information to originate TSO interactions to be delivered at the GSM simulator side. Each interaction schedules the arrival of a call, and the interaction message includes any information required by the GSM simulator to simulate the call itself (e.g. the call duration). This workload generator interacts with the underlying RTI via the conservative time management interface.

To evaluate the effects of ARML we have adopted the following GPSD approach at the level of the GSM simulator. This simulator is run as two replicas and the underlying state log/recovery modules are parameterized so to perform the log of the application level simulation state in an out-of-phase manner. Speci cally, the two replicas take state log each K simulation events, with an out-of-phase of $K/2$ events. In other words, if $K = 1$, the two replicas exhibit the same behavior by taking state log at each event. Instead, for any value of K greater than one, they behave differently by logging the state at different points in simulation time. This allows different instant responsiveness of the two replicas during both forward computation and also in a rollback phase (due to different coasting forward lengths when a TSO violation occurs at a given simulation time point). Such con guration has been compared with a standard con guration employing no replication, formed by a single instance of the GSM simulator and by the workload generator, both directly interacting with the underlying B-RTI.

In the experiments we have simulated a large urban GSM coverage area with 1024 micro cells, each managing 200 channels. Also, 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. Small-scale fading is simulated at non- xed time steps (i.e. with fading recalculation triggered by relevant events associated with mobility and with the call arrival pattern). 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 suf ce to originate a situation of no CPU contention among the involved components, i.e. the two replicas of the GSM federate, their underlying B-RTI instance and the external workload generator (recall that, while the two replicas of the GSM 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 veri ed that the sizes of the involved processes do not cause RAM contention and swapping phenomena, which, as well as CPU contention, would prevent a signi cative evaluation of the potential of the proposed replication approach. In Figure 3 we report the execution speed of the federation, evaluated in terms of simulated time units per wall-clock time unit, for the two different investigated con gurations (i.e. with and without replication), while varying the checkpoint interval K up to the value 32. Each value reported in the plots is the average over a number of samples that ensure a con dence interval of 10% around the mean at the 95% con dence level.

When the checkpoint interval K is set to 1, we have in practice no diversity since the two replicas of the GSM application exhibit the same identical run-time behavior (therefore the same identical instant responsiveness). Hence replication is expected to provide no bene ts. This is confirmed by the experimental results which show that for such a value of K, the con guration with no replication runs about 6% faster than the con guration with the two replicas. Actually, the two con gurations do not exhibit the same identical performance just because of the overhead imposed by ARML when replication is employed.

When the checkpoint interval K tends to be increased, both the con gurations tend to show better performance. However, the con guration with no replication tends to

achieve the classical "optimal tradeoff" between checkpointing and recovery costs (see, e.g., [5, 15]) for relatively reduced values of the checkpoint interval, i.e. values of K up to 8. Instead, for large values of K, this configuration exhibits a clear degradation of the execution speed due to excessive penalties associated with time requirements while handling coasting forward during a rollback phase. On the other hand, the configuration with replication is able to achieve a better balance of the checkpointing/recovery costs thanks to the out-of-phase placement of checkpoints, which also allows reducing the impact of rollback costs (i.e. coasting forward latencies) on the responsiveness of the GSM application, externally seen by the B-RTI. This allows the configuration with replication to exhibit execution speed up to 12% better than the top speed achieved with no replication vs K. We also observe that, unless for very reduced values of K (i.e. up to 2), the configuration with replication exhibits execution speed that is much more flat than the other configuration vs variations of the checkpoint interval. This provides indications of higher guarantees of acceptable run-time performance from the configuration employing GPSD at the level of the checkpointing/recovery mechanism and replication, even in case of sub-optimal selection of the value of the checkpoint interval. To further support the validity of the results, we have also performed an execution in which the GSM 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 0.09 simulation time units per wall-clock time unit. This denotes that the reported data refer to a situation in which optimistic synchronization is effective. As a final observation, the overhead by ARML could be further reduced (possibly at the expense of reduced portability on different computing platforms), in case communication between the $federate\_replication\_manager$ and the $RTI\_replication\_manager$ were implemented via ad-hoc solution tailored for the SMP architecture, instead of using sockets. This could even increase the performance benefits from replication, which additionally stands the potential of the proposed approach in a wide range of settings, also including finer grain simulation applications compared to the one considered in this case study.

References

[1] D. Anagnostopoulos and M. Nikolaidou. Executing a minimum number of replications to support the reliability of FRTS predictions. In *Proceedings of the 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 138–146. IEEE Computer Society, 2003.

[2] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Concurrent replication of parallel and distributed simulations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 234–243. IEEE Computer Society, 2005.

[3] D. Chen, S. J. Turner, B.-P. Gan, and W. Cai. HLA-based distributed simulation cloning. In *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 244–247. IEEE Computer Society, 2004.

[4] J. Dahl, M. Chetlur, and P. A. Wilsey. Event list management in distributed simulation. In *Proceedings of the 7th International Euro-Par Conference*, pages 466–475. Springer, 2001.

[5] J. Fleischmann and P. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Computer Society, June 1995.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[7] R. M. Fujimoto, T. McLean, K. S. Perumalla, and I. Tacic. Design of high performance RTI software. In *Proceedings of the 4th International Workshop on Distributed Simulation and Real-Time Applications*, pages 89–96. IEEE Computer Society, 2000.

[8] M. Hybinette and R. Fujimoto. Cloning: A novel method for interactive parallel simulation. In *Winter Simulation Conference*, pages 444–451, 1997.

[9] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):307–407, Oct. 2001.

[10] IEEE Std 1516-2000 (2000). IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

[11] IEEE Std 1516.1-2000 (2000). IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface (FI) Specification. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

[12] Y. B. Lin. Parallel independent replicated simulation on a network of workstations. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 73–80. IEEE Computer Society, 1994.

[13] M. L. Loper and R. M. Fujimoto. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 157–164. IEEE Computer Society, May 2000.

[14] A. C. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 127–134. IEEE Computer Society, 1993.

[15] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

[16] F. Quaglia and A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[17] R. Rodrigues, M. Castro, and B. Liskov. Base: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, 2001.

[18] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[19] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.