International Journal of Foundations of Computer Science
© World Scientific Publishing Company

SOFTWARE DIVERSITY-BASED ACTIVE REPLICATION AS AN APPROACH FOR ENHANCING THE PERFORMANCE OF ADVANCED SIMULATION SYSTEMS[*]

FRANCESCO QUAGLIA
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
via Salaria 113, 00198 Roma, Italy

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Active replication has been widely explored to achieve fault tolerance and to improve system availability, especially in service oriented applications. In this paper we explore software diversity-based active replication in the context of advanced simulation systems, with the aim at improving the timeliness for the production of simulation output. Our proposal is framed by the High-Level-Architecture (HLA), namely a middleware based standard for simulation package interoperability, and results in the design and implementation of an Active Replication Management Layer (ARML) targeted to off-the-shelf SMP computing systems. This layer can be interposed in between each simulator instance and the underlying HLA middleware component, in order to support the execution of diversity-based active replicas of a same simulation package 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 both simulation execution speed and performance guarantees vs tunable software parameters. (Free software releases of ARML can be found at the URL `http://www.dis.uniroma1.it/~quaglia/software/ARML`)

Keywords: Simulation systems; High-Level-Architecture; Middleware; Performance optimization; Active Replication.

1. Introduction

The High-Level-Architecture (HLA) is an IEEE standard for the interoperability of simulation packages and applications [14, 15]. It relies on a middleware level component, referred to as Run-Time-Infrastructure (RTI), offering a set of interoperability services to the overlying application software. In the recent years, this standard has received great attention, and several works have addressed the issue of enhancing the run-time effectiveness of simulation systems relying on it. Among

[*]This article appeared in the International Journal of Foundations of Computer Science, vol.18, no.3, pp.495-515, June 2007.

them, numerous solutions (see, e.g., [3, 19, 20, 28]) have been oriented to improving the run-time behavior of the underlying RTI middleware component.

Orthogonally to these solutions, in this paper we provide a framework relying on software diversity and active replication of application level simulation components, which is aimed at tackling the impact of the application level software organization on the timeliness of the output production. The 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. Within this scenario, our view of software diversity entails the adoption of different third party libraries supporting typical simulation related tasks (e.g. housekeeping tasks), or even the same library, but with different choices for the parameters determining its run-time behavior. Hence application programmers are not necessarily required to provide diversity-based implementations of a same application level simulation component (e.g. by implementing the specific simulation model via different algorithms and data structures). This means in practice following a kind of "Opportunistic N-Version Programming" such as the one followed by [24] in the context of replication in support of fault tolerance.

Typically, any 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 in scenarios where the timeliness in the production of simulation outputs (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 or cluster architectures, have become relatively cheap Commercial-Off-The-Shelf (COTS) platforms.

Beyond providing the replication framework, we propose the design and implementation of an Active Replication Management Layer (ARML), which transparently supports software diversity-based replicas of a same HLA simulator, by showing them as a single logical entity. The implementation of ARML 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 SMP systems and for being integrated with the well known Georgia Tech B-RTI package [9], 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, namely a Personal Communication System (PCS), outlining the performance benefits from our proposal. The results also show that our replication approach can provide better performance guarantees vs tunable software parameters having an impact on the run-time behavior of the simulation system.

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 are proposed in Section 3. Related work on replication approaches is discussed in

Section 4. Experimental results for an evaluation of the benefits from the proposed solution are reported in Section 5.

2. Software Diversity-Based Active Replication

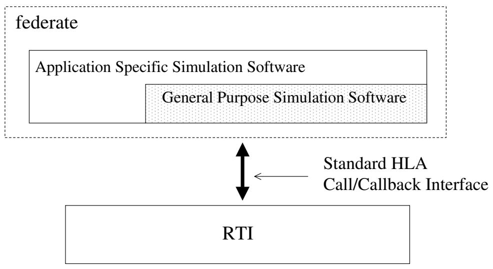
2.1. The Framework

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. (Details on main RTI services, such as object publication/subscription, message distribution and simulation time management, will be provided in Section 3.1 while presenting an overview of the B-RTI package.) As specified by HLA, the interface in between the federates and RTI includes both functions offered by the RTI and callback functions to be offered by the federates. From the point of view of software organization (see Figure 1), we see a federate 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, 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. Classical examples of general purpose simulation software include libraries implementing calendar queues and libraries implementing checkpointing/recovery mechanisms 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 software 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 potentially impact the run-time behavior of the involved modules under the same input conditions. As an example, the implementation might rely on a mixture of both pre-allocated memory and dynamically allocated memory, the latter 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/deallocation 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 ([a]), but with different internals, available for general purpose tasks in support of simulation systems,

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


federate
Application Specific Simulation Software
General Purpose Simulation Software
Standard HLA
Call/Callback Interface
RTI
Figure 1: Software Architecture in the Target Scenario.

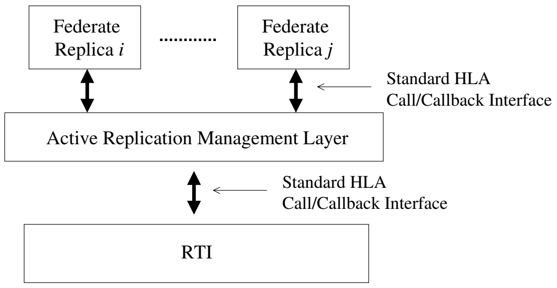
so that the application specific software can be linked to whichever of those libraries in order to achieve different implementations of a same federate possibly exhibiting different run-time behaviors under the same input conditions (see, e.g., [6, 25] for comparisons between different algorithms/implementations of calendar queues and [7, 21, 23, 27] 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 potentially impact its run-time behavior under the same input conditions.

Once defined ASSD and GPSD, the active replication approach we propose can be schematized as in Figure 2. A 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 interface exposed by 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 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 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 specific service call, the corresponding instance coming for first among all the streams. In this way,


Federate Replica *i*
....
Federate Replica *j*
Standard HLA Call/Callback Interface
Active Replication Management Layer
Standard HLA Call/Callback Interface
RTI
Figure 2: Software Architecture with the Active Replication Management Layer.

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. message distribution and related simulation 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. In particular:

C. ARML intercepts each callback 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.

2.2. Effectiveness and Correctness Requirements

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 entity, and the same external interactions, under the same input conditions (e.g. the same callbacks from the RTI). This is a classical requirement common to a wide spectrum of replication strategies proposed, e.g., in the context of fault tolerance and system availability. 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 relies 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., [18]), 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 entity 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 Computing 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 are independent of the specific underlying RTI package, the implementation is tailored for integration with the Georgia Tech B-RTI package [9]. For this reason, we initially propose a brief overview of B-RTI, which will also form the basis for the understanding of specific implementation choices. 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 itself.

3.1. Overview of B-RTI

B-RTI offers the following three classes of basic services (see Table 1):

Declaration Management Services. This class entails services 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. 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, `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 sent to all the federates subscribing that class).

Object Management Services. This class entails services for updating attributes of an object instance. The `RTI_UpdateAttributeValues()` service can be invoked by the owner of the object instance, i.e. the federate that published the corresponding object class and created that 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 retracting (i.e. undoing) the delivery of an object update already issued to the federates via `ReflectAttributeValues()`, 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`, `double` or memory addresses. Specifically, `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`. Also, `RTI_ObjInstanceDesignator` is the memory address of object related data structures maintained by B-RTI, and `struct MsgS*` is the memory address of B-RTI managed messages.

Time Management Services. This class entails synchronization services. Among them, the unique callback is `TimeAdvanceGrant()`, which is used by B-RTI 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 possibility to advance the local simulation clock to a given simulation time and the delivery of incoming messages (with timestamps up to that simulation 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 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

As pointed out in Section 2, the requirement for the effectiveness of the active replication framework is that software diversity-based replicas of the federate simulator and the underlying RTI must be executed in real parallelism. In our design, we satisfy this requirement by maintaining complete transparency for what concerns the presence of the ARML layer. This means that both the federate and B-RTI must undergo no modifications (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) in order to be integrated with the ARML layer. To achieve this objective, we have decided to organize ARML into two independent software modules, namely *federate_replication_manager* and *RTI_replication_manager*, to be linked to the federate and to B-RTI, respectively, and to let the federate replicas and B-RTI run as separate UNIX processes on the

Table 1: B-RTI Services.

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 *)`
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)`
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)`

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 (see Section 3.1), 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 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. To solve this problem, we have decided to support the interaction between the two independent modules, namely *federate_replication_manager* and *RTI_replication_manager*, via an ad-hoc mechanism based on a shared memory buffer (see Figure 3). Specifically, a service call issued by the federate is intercepted by *federate_replication_manager* and is then translated into a write operation on the shared memory buffer so that the corresponding information can be read by *RTI_replication_manager* and the 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, when B-RTI has control, each callback issued towards the federate (if any) 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 callback in the federate address space and returns the callback output to

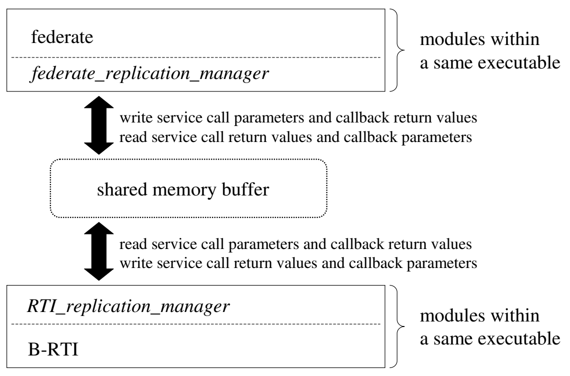

federate
federate_replication_manager
modules within a same executable
write service call parameters and callback return values
read service call return values and callback parameters
shared memory buffer
read service call parameters and callback return values
write service call return values and callback parameters
RTI_replication_manager
B-RTI
modules within a same executable

Figure 3: Interaction via the Shared Memory Buffer.

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 a 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. This objective means allowing 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 the call/callback to/from B-RTI involve a non-minimal amount of bytes. This might be the case of large size pointed objects (e.g. large size pointed messages related to the publish/subscribe communication mechanism) associated with pointer parameters, which need to be transferred via the shared memory buffer to make them available at different processes. This objective means in practice that each call to B-RTI services should be issued by the different *federate_replication_manager* instances to *RTI_replication_manager* by having the corresponding information written only once into the shared memory buffer, and the return value should be made available to all these replicas by writing it only once into the shared memory buffer. A similar reasoning can be done for the treatment of callbacks issued by B-RTI. Specifically, data associated with a callback operation should be made available to all the federate replicas by writing it only once into the shared memory buffer. The same should happen for the result value of each callback, i.e. only one among the different *federate_replication_manager* instances should perform the write operation of the callback return value into the shared memory 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 avoid the overhead due to system calls (e.g. the

```
typedef struct _call_callback_descriptor{  /* descriptor of a call or callback */
   int  callback;                         /* is this the description of a callback? */
   int  code;                             /* call/callback numerical code */
   char arguments[MAX_ARGUMENTS_SIZE]; /* call/callback arguments packed in a contiguous buffer */
} call_callback_descriptor;
```

Figure 4: Call/Callback Descriptors.

cost of saving user level CPU context into kernel level stack plus the cost of system call dispatching), and would also avoid any secondary overhead effect associated with process dispatching operations (e.g. in case of a previous event of block while in kernel mode) ([b]).

3.3. Implementation Details

We have developed an implementation of the data structures maintained into the shared memory buffer, and of the corresponding management modules, 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 requirement 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. 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.

As outlined while discussing the minimal number of data copies requirement, in general there might be call/callback parameters which identify pointed objects to be exchanged between federates and B-RTI. This is the case of `struct MsgS*` and `char*` parameters, which represent pointers to memory areas that contain data to be accessed by both B-RTI and the federate. For instance, when the `ReflectAttributeValues()` callback is issued, the `struct MsgS*` parameter

[b]An early implementation [22] 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.

points to a memory buffer where, according to the publish/subscribe communication mode, the message to be delivered to the federate has been placed by B-RTI. 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 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 packing operations is represented in Figure 4. It has the following fields:

- 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 code of the service or callback (this is used to perform correct binding of calls and callbacks in the different address spaces).
- 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 5.A.

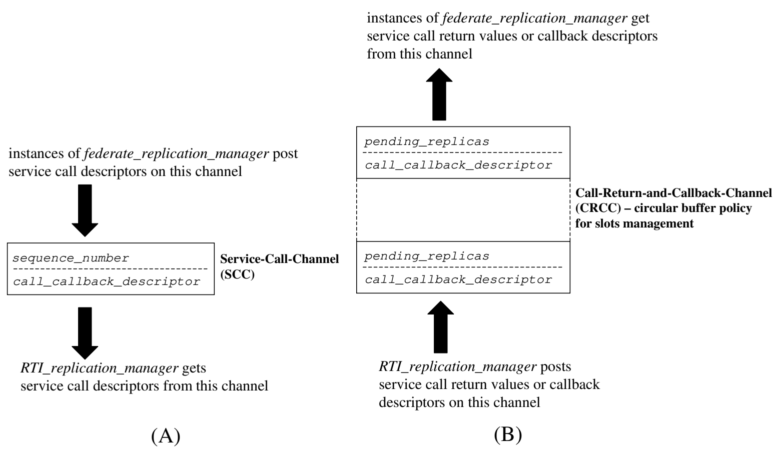
This channel is used by the different replicas of *federate_replication_manager* to notify to *RTI_replication_manager* calls to B-RTI services. The `sequence_number` field is used to implement the minimal number of data copies requirement as follows. When whichever instance of *federate_replication_manager* 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 current 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 assigned by the replica to its service call 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 B-RTI. Filtering out this call simply means that we avoid copying the corresponding description within SCC. In this way, each service call is issued by having the corresponding information written only once, i.e. by one federate replica only, 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 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 callback (e.g. callback parameters), needs to be buffered until all the replicas have read it. To achieve such a buffering objective, while also matching the minimal number of data copies requirement, 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 for CRCC is provided in Figure 5.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 the following two fields:

- A field of type `call_callback_descriptor`.
- A 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



instances of *federate_replication_manager* get service call return values or callback descriptors from this channel
pending_replicas
call_callback_descriptor
Call-Return-and-Callback-Channel (CRCC) – circular buffer policy for slots management
pending_replicas
call_callback_descriptor
RTI_replication_manager posts service call return values or callback descriptors on this channel
(B)
instances of *federate_replication_manager* post service call descriptors on this channel
sequence_number
call_callback_descriptor
Service-Call-Channel (SCC)
RTI_replication_manager gets service call descriptors from this channel
(A)

Figure 5: Channels Implemented within the Shared Memory Buffer.

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).

Finally, the third channel we have implemented within the shared memory buffer, namely Callback-Return-Channel (CRC), is used by the federate replicas to provide the return value of callbacks to B-RTI. This channel consists of a data structure identical to the one implementing SCC. Specifically, it is a record formed by a `call_callback_descriptor` field, used for packing the callback return value, and by a `sequence_number` field, which is used according to the same policy adopted for the SCC channel. More precisely, as for service calls, 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 $federate\_replication\_manger$ within CRC only in case the associated sequence number is greater than the current 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, which again matches the minimal number of data copies requirement.

Actually, wait statements on any channel have been implemented by exploiting the same application level spinlocks used for solving critical sections on data structures within the shared memory 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 based on the `call_callback_descriptor` data structure. This is `MCAST_WhereProc`, namely the function pointer indicating to 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 B-RTI are linked together to generate a same executable). In this way the federate can define a `NULL` value for the `MCAST_WhereProc` parameter (since B-RTI will reserve memory for object update messages into its own address space) and, consequently, it can also define a `NULL` value for the function pointer it should use for releasing those "B-RTI controlled" memory buffers once the corresponding messages are delivered to the federate through the appropriate callbacks. This even simplifies the application programmer job while developing applications to be integrated with the B-RTI package.

4. Related Work

Active replication has been widely explored in the literature for achieving fault tolerance and improving system availability, especially in the context of client/server and multi-tier systems (see, e.g., [24, 26]). The related solutions are mostly oriented to provide efficient mechanisms for allowing consistency of the state of the server side replicas in the presence of interleaving of requests from different clients, or even message ordering inversions due to, e.g., network delays. This is typically achieved via quorum systems [10], or atomic broadcast communication primitives allowing agreement on the processing order at different replicas [11]. Compared to these proposals, our approach differs in that it is specifically tailored to system responsiveness during normal operation mode, instead of system availability and effectiveness in faulty scenarios. This difference is also confirmed by the observation that, compared to our proposal, the speed of system output production in those fault tolerance techniques is bounded by the execution speed of the slower replicas each time a vote collection mechanism to mask (Byzantine) failures is employed (e.g. a quorum), which needs to collect output from different replicas before providing any result. Given that our target is different in nature, our middleware replication component takes the different approach of producing the output (e.g. messages for remote simulator instances) on the basis of the output provided by the overlying replica with fastest instant responsiveness, while still transparently maintaining eventual consistency of all the replicas via appropriate call/callback delivery ordering and filtering.

For what concerns solutions specifically oriented to enhancing the simulation system performance, a kind of replication approach known as cloning has been proposed in [4, 12, 13]. The aim of these solutions 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 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, 17]), 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 increasing the execution speed of 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 [8] 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 classical 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 SMP systems) 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 a mobile Personal Communication System (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) [16]. 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 explicit 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 that call (e.g. the call duration).

5.2. Achieving Software Diversity

In this experimental study we show the effectiveness of our implementation of ARML while exploiting the GPSD approach to software diversity. We recall that GPSD is the source of software diversity most directly related to a kind of pragmatic, opportunistic N-version programming [24] dealing with tasks common to a variety of simulation packages, which can be supported by different (or differently parameterized) third party libraries. Hence, GPSD fits the requirements of a pragmatic scenario in which we expect no (or minimal) application level programmer involvement for achieving diversity among software replicas of a given simulation component.

The PCS simulator uses the optimistic Time Management interface offered by the underlying RTI to synchronize with the external workload generator, and the GPSD 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 purposed of the application level state. Specifically, the HLA optimistic Time Management interface 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. This approach requires the replay (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 gives rise to different instant responsiveness from 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 point in simulation time). 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 software diversity-based active replication for performance purposes ([c]).

5.3. 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 processes, i.e. the two 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.

5.4. Results

In Figure 6 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 we can draw the following main conclusions. 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

[c]The overhead due to ARML might vary while increasing the number of managed replicas, due, e.g., to variations of the effects of the synchronization facilities used for handling the interactions among different processes via the shared memory buffer. Hence, the overhead evaluation with a single instance of the overlying federate simulator is a baseline reference indication, and is not intended as an exhaustive study on how the overhead scales up with the degree of replication.

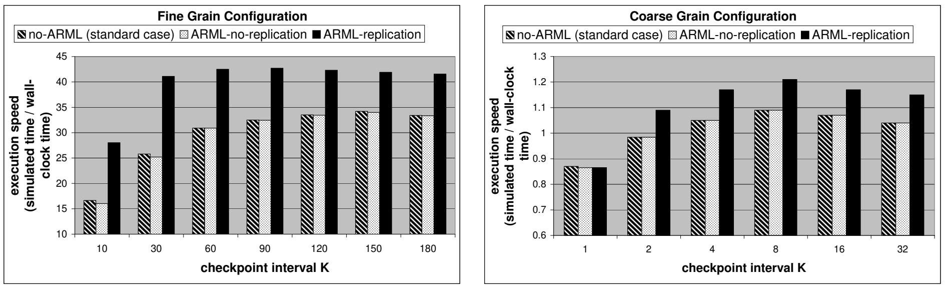


Figure 6: Simulation Execution Speed for the Two Configurations.

simulator (labelled ARML-no-replication). In fact, the corresponding 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 a reduction of the impact of rollback costs (i.e. coasting forward latencies) on the responsiveness of the PCS application, externally seen by 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 where simulation events have large CPU requirements, 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.

We also observe that, unless for very reduced values of K (i.e. up to 10 for the fine grain configuration and up to 2 for the coarse grain configuration), the active replication approach exhibits execution speed that is relatively flat vs variations of the checkpoint interval. This provides indications of guarantees of acceptable run-time performance with the configuration employing active replication and GPSD at the level of the checkpointing/recovery modules, 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 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. Conclusion

In this paper we have presented a replication framework for federated simulation systems based on the HLA standard for interoperability. Starting from the framework, we have also developed an implementation of a middleware layer, namely ARML, able to transparently handle replication of simulator instances, possibly based on a kind of opportunistic N-version programming to achieve software diversity, in order to improve the timeliness for the production of simulation results. We have discussed the relation between our approach and well known replication strategies. Additionally, we have presented the results of an experimental study on a mobile simulation application, demonstrating the effectiveness of our proposal both in improving the execution speed of the federated simulation system and in providing better guarantees of adequate performance vs tunable software parameters.

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. A. Boukerche and K. Lu. Design and performance evaluation of a real-time RTI infrastructure for large-scale distributed simulations. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real-Time Applications*, pages 203–212. IEEE Computer Society, 2005.
4. 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.
5. G. Cong and D. A. Bader. Lock-free parallel algorithms: An experimental study. In *Proceedings of the 11th International Conference High Performance Computing*, volume 3296 of *Lecture Notes in Computer Science*, pages 516–528. Springer, 2004.
6. 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.
7. 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.
8. R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.
9. 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.
10. D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.
11. V. Hadzilacos and S. Toueg. *Distributed Systems - Chapter 5: Fault-Tolerant Broadcasts and Related Problems.* Addison-Wesley, 1993.

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

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

14. 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.

15. 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.

16. S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

17. 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.

18. 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.

19. T. McLean and R. M. Fujimoto. Predictable time management for real-time distributed simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pages 89–96. IEEE Computer Society, 2003.

20. T. McLean, R. M. Fujimoto, and J. B. Fitzgibbons. Middleware for real-time distributed simulations. *Concurrency - Practice and Experience*, 16(15):1483–1501, 2004.

21. 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.

22. F. Quaglia. Enhancing the performance of HLA-based simulation systems via software diversity and active replication. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium - APDCM Workshop*, page 266. IEEE Computer Society, 2006.

23. 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.

24. 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.

25. 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.

26. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Survey*, 22(4):299–319, 1990.

27. 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.

28. H. Zhao and N. D. Georganas. HLA real-time extension. In *Proceedings of the 5th International Workshop on Distributed Simulation and Real-Time Applications*,

pages 12–21. IEEE Computer Society, 2001.