Fault-Tolerant Adaptive Parallel and Distributed Simulation

Gabriele D'Angelo Stefano Ferretti Moreno Marzolla
Dept. of Computer Science and Engineering, University of Bologna, Italy
Email: {g.dangelo,s.ferretti,moreno.marzolla}@unibo.it

Lorenzo Armaroli
Email: lorenzo.armaroli@gmail.com

***Abstract*—Discrete Event Simulation is a widely used technique that is used to model and analyze complex systems in many fields of science and engineering. The increasingly large size of simulation models poses a serious computational challenge, since the time needed to run a simulation can be prohibitively large. For this reason, Parallel and Distributes Simulation techniques have been proposed to take advantage of multiple execution units which are found in multicore processors, cluster of workstations or HPC systems. The current generation of HPC systems includes hundreds of thousands of computing nodes and a vast amount of ancillary components. Despite improvements in manufacturing processes, failures of some components are frequent, and the situation will get worse as larger systems are built. In this paper we describe FT-GAIA, a software-based fault-tolerant extension of the GAIA/ARTÌS parallel simulation middleware. FT-GAIA transparently replicates simulation entities and distributes them on multiple execution nodes. This allows the simulation to tolerate crash-failures of computing nodes; furthermore, FT-GAIA offers some protection against byzantine failures since synchronization messages are replicated as well, so that the receiving entity can identify and discard corrupted messages. We provide an experimental evaluation of FT-GAIA on a running prototype. Results show that a high degree of fault tolerance can be achieved, at the cost of a moderate increase in the computational load of the execution units.**

I. Introduction

Computer-assisted modeling and simulation plays an important role in many scientific disciplines: computer simulations help to understand physical, biological and social phenomena. Discrete Event Simulation (DES) is of particular interest, since it is frequently employed to model and analyze many types of systems, including computer architectures, communication networks, street traffic, and others.

In a DES, the system is described as a set of interacting entities; the state of the simulator is updated by simulation *events*, which happen at discrete points in time. The overall structure of a sequential event-based simulator is relatively simple: the simulator engine maintains a list, called Future Event List (FEL), of all pending events, sorted in non decreasing time of occurrence. The simulator executes a loop, where at each iteration, the event with lower timestamp t is removed from the FEL, and the simulation time is advanced to t. Then, the event is executed, possibly triggering the generation of new events to be scheduled for execution at some future time.

[0]The publisher version of this paper is available at https://doi.org/10.1109/DS-RT.2016.11. **Please cite this paper as: "Gabriele D'Angelo, Stefano Ferretti, Moreno Marzolla, Lorenzo Armaroli. Fault-Tolerant Adaptive Parallel and Distributed Simulation. Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2016)".**

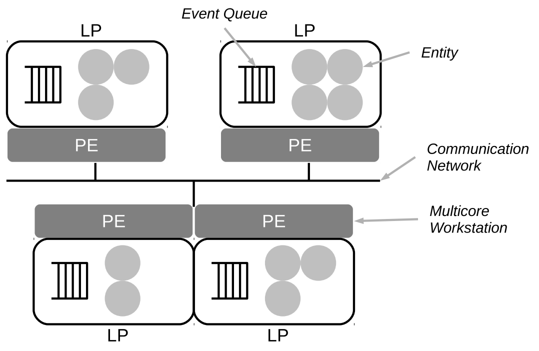

Fig. 1. Structure of a Parallel and Distributed Simulation.

Continuous advances in our understanding of complex systems, combined with the need for higher model accuracy, demand an increasing amount of computational power and represent a major challenge for the capabilities of the current generation of high performance computing systems. Therefore, sequential DES techniques may be inappropriate for analyzing large or detailed models, due to the huge number of events that must be processed. Parallel and Distributed Simulation (PADS) aims at taking advantage of modern high performance computing architectures – from massively parallel computers to multicore processors – to handle large models efficiently [1]. The general idea of PADS is to partition the simulation model into submodels, called Logical Processs (LPs) which can be evaluated concurrently by different Processing Elements (PEs). More precisely, the simulation model is described in terms of multiple interacting Simulated Entitys (SEs) which are assigned to different LPs. Each LP is executed on a different PE, and is in practice the container of a set of entities. The execution of the simulation is obtained through the exchange of timestamped messages (representing simulation events) between entities. Each LP has an queue where messages are inserted before being dispatched to the appropriate entities. Figure 1 shows the general structure of a parallel and distributed simulator.

Execution of long-running applications on increasingly larger parallel machines is likely to hit the *reliability wall* [2].

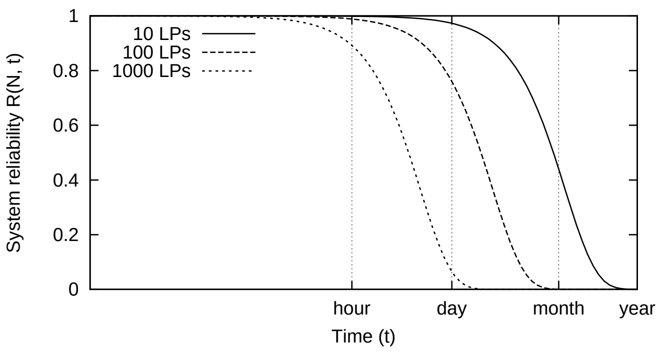

Fig. 2. System reliability $R(N,t)$ assuming a MTTF for each LP of one year; higher is better, log scale on the x axis.

This means that, as the system size (number of components) increases, so does the probability that at least one of those components fails, therefore reducing the system Mean Time To Failure (MTTF). At some point the execution time of the parallel application may become larger than the MTTF of its execution environment, so that the application has little chance to terminate normally.

As a purely illustrative example, let us consider a parallel machine with N PEs. Let X_i be the stochastic variable representing the duration of uninterrupted operation of the i-th PE, taking into account both hardware and software failures. Assuming that all X_i are independent and exponentially distributed (this assumption is somewhat unrealistic but widely used [3]), we have that the probability $P(X_i > t)$ that LP i operates without failures for at least t time units is

$$P(X_i > t) = e^{-\lambda t}$$

where λ is the failure rate. The joint probability that all N LPs operate without failures for at least t time units is therefore $R(N,t) = \prod_i P(X_i > t) = e^{-N\lambda t}$; this is the formula for the reliability of N components connected in series, where each component fails independently, and a single failure brings down the whole system.

Figure 2 shows the value of $R(N,t)$ (the probability of no failures for at least t consecutive time units) for systems with $N = 10, 100, 1000$ LPs, assuming a MTTF of one year ($\lambda \approx 2.7573 \times 10^{-8} s^{-1}$). We can see that the system reliability quickly drops as the number of LPs increases: a simulation involving $N = 1000$ LPs and requiring one day to complete is very unlikely to terminate successfully.

Although the model above is overly simplified, and is not intended to provide an accurate estimate of the reliability of actual parallel simulations, it does show that building a reliable system out of a large number of unreliable parts is challenging.

Two widely used approaches for handling hardware-related reliability issues are those based on *checkpointing*, and on *functional replication*. The checkpoint-restore paradigm requires the running application to periodically save its state on non-volatile storage (e.g., disk) so that it can resume execution from the last saved snapshot in case of failure. It should be observed that saving a snapshot may require considerable time; therefore, the interval between checkpoints must be carefully tuned to minimize the overhead.

Functional replication consists on replicating parts of the application on different execution nodes, so that failures can be tolerated if there is some minimum number of running instances of each component. Note that each component must be modified so that it is made aware that multiple copies of its peers exist, and can interact with all instances appropriately.

It is important to remark that functional replication is not effective against logical errors, i.e., bugs in the running applications, since the bug can be triggered at the same time on all instances. A prominent – and frequently mentioned – example is the failure of the Ariane 5 rocket that was caused by a software error on its Inertial Reference Platforms (IRPs). There were two IRP, providing hardware fault-tolerance, but both used the same software. When the two software instances were fed with the same (correct) input from the hardware, the bug (an uncatched data conversion exception) caused both programs to crash, leaving the rocket without guidance [4]. The N-version programming technique [5] can be used to protect against software errors, and requires running several functionally equivalent programs that have been independently developed from the same specifications.

In this paper, we present FT-GAIA, a fault-tolerant extension of the GAIA/ARTÌS parallel and distributed simulation middleware [6], [7]. FT-GAIA is based on functional replication, and can handle crash errors and byzantine faults, using the concept of *server groups* [8]: simulation entities are replicated so that the model can be executed even if some of them fail. We show how functional replication can be implemented as an additional software layer in the GAIA/ARTÌS stack; all modifications are transparent to user-level simulation models, therefore FT-GAIA can be used as a drop-in replacement to GAIA/ARTÌS when fault tolerance is the major concern.

This paper is organized as follows. In Section II we review the art related to fault tolerance in PADS. The GAIA/ARTÌS parallel and distributed simulation middleware is described in Section III. Section IV is devoted to the description of FT-GAIA, a fault-tolerant extension to GAIA/ARTÌS. An empirical performance evaluation of FT-GAIA, based on a prototype implementation we have developed, is discussed in Section V. Finally, Section VI provides some concluding remarks.

II. Related Work

Although fault tolerance is an important and widely discussed topic in the context of distributed systems research, it received comparatively little attention by the PADS community. The proposed approaches for bringing fault tolerance to PADS are either based on checkpointing or on functional replication, with a few works considering also partially centralized architectures.

A. Checkpointing

In [9] the authors propose a rollback based optimistic recovery scheme in which checkpoints are periodically saved on stable storage. The distributed simulation uses an optimistic synchronization scheme, where out-of-order ("straggler") events cause rollbacks that are handled according to the Time Warp protocol [10]. The novel idea is to model failures as straggler events with a timestamp equal to the last saved checkpoint. In this way, the authors can leverage the Time Warp protocol to handle failures.

In [11], [12] the authors propose a new framework called Distributed Resource Management System (DRMS) to implement reliable IEEE 1516 federation [13]. The DRMS handles crash failures using checkpoints saved to stable storage, that is then used to migrate federates from a faulty host to a new host when necessary. The simulation engine is again based on an optimistic synchronization scheme, and the migration of federates is implemented through Web services.

In [14] the authors propose a decoupled federate architecture in which each IEEE 1516 federate is separated into a virtual federate process and a physical federate process. The former executes the simulation model and the latter provides middleware services at the backend. This solution enables the implementation of fault-tolerant distributed simulation schemes through migration of virtual federates.

The CUMULVS middleware [15] introduces the support for fault tolerance and migration of simulations based on checkpointing. The middleware is not designed to support PADS but it allows the migration of running tasks for load balancing and to improve a task's locality with a required resource.

A slightly different approach is proposed in [16]. In which, the authors introduce the Fault Tolerant Resource Sharing System (FT-RSS) framework. The goal of FT-RSS is to build fault tolerant IEEE 1516 federations using an architecture in which a separate FTP server is used as a persistent storage system. The persistent storage is used to implement the migration of federates from one node to another. The FT-RSS middleware supports replication of federates, partial failures and fail-stop failures.

B. Functional Replication

In [17] the authors propose the use of functional replication in Time Warp simulations with the aim to increase the simulator performance and to add fault tolerance. Specifically, the idea is to have copies of the most frequently used simulation entities at multiple sites with the aim of reducing message traffic and communication delay. This approach is used to build an optimistic fault tolerance scheme in which it is assumed that the objects are fault free most of the time. The rollback capabilities of Time Warp are then used to correct intermittent and permanent faults.

In [18] the authors describe DARX, an adaptive replication mechanism for building reliable multi-agent systems. Being targeted to multi-agent systems, rather than PADS, DARX is mostly concerned with adaptability: agents may change their behavior at any time, and new agents may join or leave the system. Therefore, DARX tries to dynamically identify which agents are more "important", and what degree of replication should be used for those agents in order to achieve the desired level of fault-tolerance. It should be observed that DARX only handles crash failures, while FT-GAIA also deals with Byzantine faults.

III. The GAIA-ARTÌS Middleware

To make this paper self-contained, we provide in this section a brief introduction of the GAIA/ARTÌS parallel and distributed simulation middleware; the interested reader is referred to [6], [7], [19] and the software homepage [20].

The *Advanced RTI System* (ARTÌS) is a parallel and distributed simulation middleware loosely inspired by the Runtime Infrastructure described in the IEEE 1516 standard "High Level Architecture" (HLA) [21]. ARTÌS implements a parallel/distributed architectures where the simulation model is partitioned in a set of LPs [1]. As described in Section I, the execution architecture in charge of running the simulation is composed of interconnected PEs and each PE runs one or more LPs (usually, a PE hosts one LP).

In a PADSs, the interactions between the model components are driven by message exchanges. The low computation/communication ratio makes PADS communication-bound, so that the wall-clock execution time of distributed simulations is highly dependent on the performance of the communication network (i.e., latency, bandwidth and jitter). Reducing the communication overhead can be crucial to speed up the event processing rate of PADS. This can be achieved by clustering interacting entities on the same physical host, so that communications can happen through shared memory.

Among the various services provided by ARTÌS, time management (i.e., synchronization) is fundamental for obtaining correct simulation runs that respect the causality dependencies of events. ARTÌS supports both conservative (Chandy-Misra-Bryant [22]) and optimistic (Time Warp [10]) synchronization algorithms. Moreover, a very simple time-stepped synchronization is supported.

The *Generic Adaptive Interaction Architecture* (GAIA) is a software layer built on top of ARTÌS [20]. In GAIA, each LP acts as the container of some SEs: the simulation model is partitioned in its basic components (the SEs) that are allocated among the LPs. The system behavior is modeled by the interactions among the SEs; such interactions take the form of timestamped messages that are exchanged among the entities. From the user's point of view, a simulation model based on GAIA/ARTÈS follows a Multi Agent System (MAS) approach. In fact, each SE is an autonomous agent that performs some actions (individual behavior) and interacts with other agents in the simulation.

In most cases, the interaction between the SEs of a PADS are not completely uniform, meaning that there are clusters of SEs where internal interactions are more frequent. The structure of these clusters of highly interacting entities may change over time, as the simulation model evolves. The identification of such clusters is important to improve the

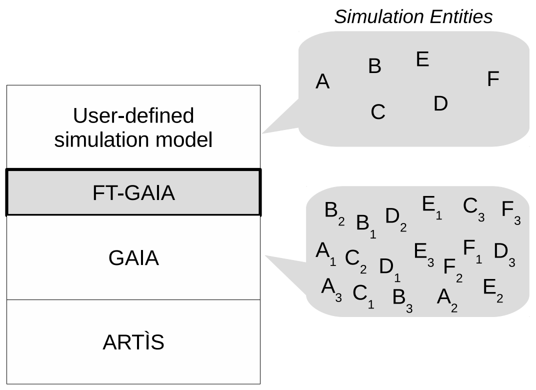
Simulation Entities
A
B
C
D
E
F
User-defined simulation model
FT-GAIA
GAIA
ARTÌS
B_2 B_1 D_2 E_1 C_3 F_3
A_1 C_2 D_1 E_3 F_2 F_1 D_3
A_3 C_1 B_3 A_2 E_2

Fig. 3. Layered structure of the FT-GAIA simulation engine. The user-defined simulation model defines a set of entities $\{A, B, C, D, E, F\}$; FT-GAIA creates multiple (in this example, 3) instances of each entity, that are handled by GAIA.

performance of a PADS: indeed, by putting heavily-interacting entities on as few LPs as possible, we may replace most of the expensive LAN/WAN communications by more efficient shared memory messages.

In GAIA, the analysis of the communication pattern is based on simple self-clustering heuristics [19]. For example, in the default heuristic, every few timesteps for each SE is found which LP is the destination of the large percentage of interactions. If it is not the LP in which the SE is contained then a migration is triggered. The migration of SEs among LPs is transparent to the simulation model developer; entities migration is useful not only to reduce the communication overhead, but also to achieve better load-balancing among the LPs, especially on heterogeneous execution platforms where execution units are not identical. In these cases, GAIA can migrate entities away from less powerful PEs, towards more capable processors if available.

IV. Fault-Tolerant Simulation

FT-GAIA is a fault-tolerant extension to the GAIA/ARTÌS distributed simulation middleware. As will be explained below, FT-GAIA uses functional replication of simulation entities to achieve tolerance against crashes and Byzantine failures of the PEs.

FT-GAIA is implemented as a software layer on top of GAIA and provides the same functionalities of GAIA with only minor additions. Therefore, FT-GAIA is mostly transparent to the user, meaning that any simulation model built for GAIA can be easily ported to FT-GAIA.

FT-GAIA works by replicating simulation entities (see Fig. 3) to tolerate crash-failures and byzantine faults of the PEs. A crash may be caused by a failure of the hardware – including the network connection – and operating system. A byzantine failure refers to an arbitrary behavior of a PE that causes the LP to crash, terminate abnormally, or to send arbitrary messages (including no messages at all) to other PEs.

Replication is based on the following principle. If a conventional, non-fault tolerant distributed simulation is composed of N distinct simulation entities, FT-GAIA generates $N \times M$ entities, by generating M independent instances of each simulation entity. All instances $A_1, \ldots A_M$ of the same entity A perform the same computation: if no fault occurs, they produce the same result.

Replication comes with a cost, both in term of additional processing power that is needed to execute all instances, and also in term of an increased communication load between the LPs. Indeed, if two entities A and B communicate by sending a message from A to B, then after replication each instance A_i must send the same message to all instances B_j, $1 \leq i, j \leq M$, resulting in M^2 (redundant) messages. Therefore, the level of replication M must be chosen wisely in order to achieve a good balance between overhead and fault tolerance, also depending on the types of failures (crash failures or Byzantine faults) that the user wants to address.

Handling crash failures: A crash failure happens when a PE halts, but operated correctly until it halted. In this case, all simulation entities running on that PE stop their execution and the local state of computation is lost. From the theory of distributed systems, it is known that in order to tolerate f crash failures we must execute at least $M = f + 1$ instances of each simulation entity. Each instance must be executed on a different PEs, so that the failure of a PE only affects one instance of all entities executed there. This is is equivalent to running M copies of a monolithic (sequential) simulation, with the difference that a sequential simulation does not incur in communication and synchronization overhead. However, unlike sequential simulations, FT-GAIA can take advantage of more than M PEs, by distributing all $N \times M$ entities on the available execution units. This reduces the workload on the PEs, reducing the wall-clock execution time of the simulation model.

Handling Byzantine Failures: Byzantine failures include all types of abnormal behaviors of a PE. Examples are: the crash of a component of the distributed simulator (e.g., LP or entity); the transmission of erroneous/corrupted data from an entity to other entities; computation errors that lead to erroneous results. In this case $M = 2f + 1$ replicas of a system are needed to tolerate up to f byzantine faults in a distributed system using the "majority" rule: an SE instance B_i can process an incoming message m from A_j when it receives at least $f + 1$ copies of m from different instances of the sender entity A. Again, all M instances of the same SE must be located on different PEs.

Allocation of Simulation Entities: Once the level of replication M has been set, it is necessary to decide where to create the M instances of all SEs, so that the constraint that each instance is located on a different PE is met. In FT-GAIA the deployment of instances is performed during the setup of the simulation model. In the current implementation, there is a centralized service that keeps track of the initial location of all SE instances. When a new SE is created, the service creates the appropriate number of instances according to the redundancy model to be employed, and assigns them to the LPs so that all instances are located on different LPs.

Note that all instances of the same SE receive the same initial seed for their internal pseudo-random number generators; this guarantees that their execution traces are the same, regardless of the LP where execution occurs and the degree of replication.

Message Handling: We have already stated that fault-tolerance through functional replication has a cost in term of increased message load among SEs. Indeed, for a replication level M (i.e., there are M instances of each SE) the number of messages exchanged between entities grows by a factor of M^2.

A consequence of message redundancy is that message filtering must be performed to avoid that multiple copies of the same message are processed more than once by the same SE instance. FT-GAIA takes care of automatically filtering the excess messages according to the fault model adopted; filtering is done outside of the SE, which are therefore totally unaware of this step. In the case of crash failures, only the first copy of each message that is received by a SE is processed; all further copies are dropped by the receiver. In the case of Byzantine failures with replication level $M = 2f + 1$, each entity must wait for at least $f + 1$ copies of the same message before it can handle it. Once a strict majority has been reached, the message can be processed and all further copies of the same messages that might arrive later on can be dropped.

Entities Migration: PADS can benefit from migration of entities to balance computation/communication load and reduce the communication cost, by placing entities that interact frequently "next" to each other (e.g., on the same LP) [19]. In FT-GAIA, entity migration is subject to the constraint that instances of the same SE can never reside on the same LP. Entity migration is handled by the underlying GAIA/ARTÌS middleware [6]: each LP runs a fully distributed "clustering heuristic" that tries to put together (i.e., on the same LP) the SEs that interact frequently through message exchanges. Special care is taken to avoid putting too many entities on the same LPs that would become a bottleneck. Once a new feasible allocation is found, the entities are migrated by moving their state to the new LP.

V. Experimental Evaluation

In this section we evaluate a prototype implementation of FT-GAIA by implementing a simple simulation model of a Peer-to-Peer communication system. We execute the simulation model with FT-GAIA under different workload parameters (described below) and record the Wall Clock Time (WCT) (excluding the time to setup the simulation) and other metrics of interest. The tests were performed on a cluster of workstations, each being equipped with an Intel Core i5-4590 3.30 GHz processors with 8 GB of RAM. The Operating System was Debian Jessie. The workstations are connected through a Fast Ethernet LAN.

A. Simulation Model

We simulate a simple P2P communication protocol over randomly generated directed overlay graphs. Nodes of the graphs are peers, while links represent communication connections [23], [24]. In these overlays, nodes have all the same out-degree, that has been set to 5 in our experiments. During the simulation, each node periodically updates its neighbor set. Latencies for message transmission over overlay links are generated using a lognormal distribution [25].

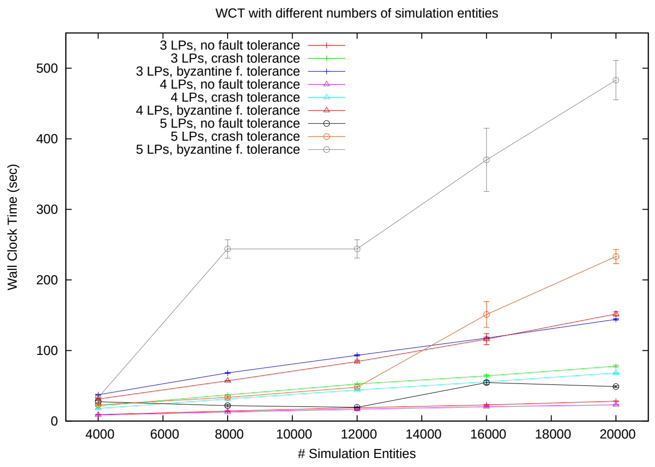

Fig. 4. Wall Clock Time as a function of the number of LPs, for varying number of SEs. The number of LPs is equal to the number of PEs. Migration is disabled. Lower is better.

The simulated communication protocol works as follows. Periodically, nodes send PING messages to other nodes, that in turn reply with a PONG message that is used by the sender to estimate the average latencies of the links (note that communication links are, in fact, bidirectional). The destination of a PING is randomly selected to be a neighbor (with probability p), or a non-neighbor (with probability $1-p$). A neighbor is a node that can be reached through an outgoing link in the directed overlay graph.

Each node of the P2P overlay is represented by a SE within some LP. Unless stated otherwise, each LP was executed on a different PE, so that no two LPs shared their execution node. We consider three scenarios: a *no fault* scenario, where no faults occur, a *crash* scenario, where crash failures occurs, and a *Byzantine* scenario where Byzantine faults occurs.

We executed 15 independent replications of each simulation run. In all the following charts, mean values are reported with a 99.5% confidence interval.

B. Impact of the number of LPs and SEs

Figure 4 shows the WCT of the simulation that was executed for 10000 timesteps with a varying number of SEs; recall that the number of SEs is equal to the number of nodes in the P2P overlay graph. The number of LPs was set to 3, 4, and 5. We show the WCT for the three failure scenarios we are considering: no failure, a single crash, and a single Byzantine failure. In all these cases the self-clustering (i.e. migration) is disabled.

Results with 3 and 4 LPs are similar, with a slight improvement with 4 LPs. Conversely, higher WCT is observed when 5 LPs are used. As expected, the higher the number

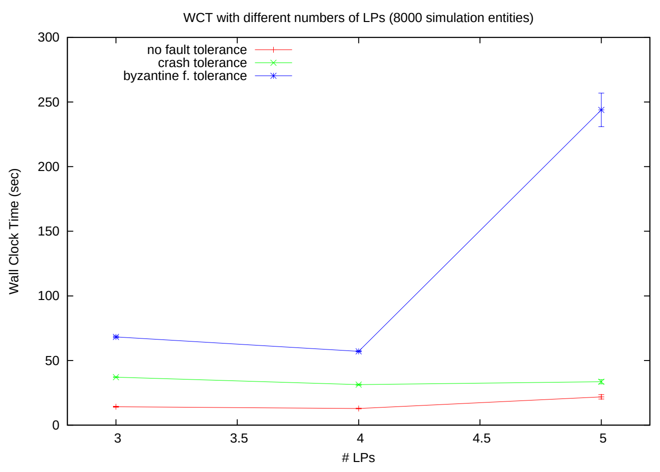

Fig. 5. Wall Clock Time as a function of the number of LPs, with 8000 SEs. Migration is disabled. Lower is better.

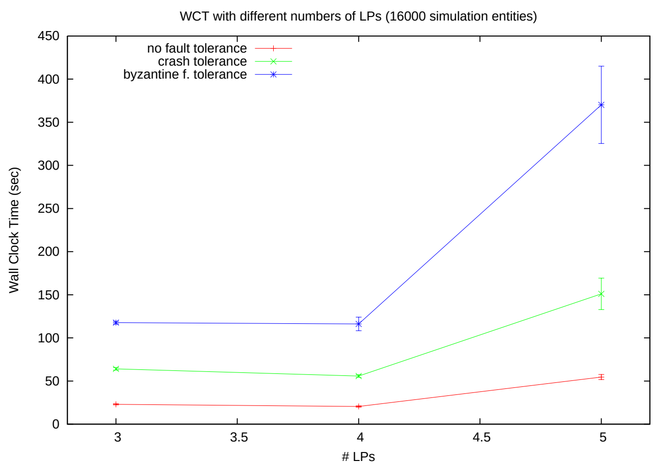

Fig. 6. WCT as a function of the number of LPs, with 10000 SEs. Migration is disabled. Lower is better.

of SEs the higher the WCT, since the simulation incurs in a higher communication overhead. Moreover, all curves have a similar trend. In particular, the increment due to the faults management schemes is mainly due to the higher amount of messages exchanged among nodes.

Figures 5 and 6 show the WCT when varying the number of LPs, with 8000 and 16000 SEs, respectively. The two charts emphasize the increment of the time required to terminate the simulations with 5 LPs and in presence of Byzantine faults. This is due to the increased number of messages exchanged among the LPs: each message needs to be sent to three $(2M+1)$ different destinations in order to guarantee fault tolerance.

C. *Impact of the number of LPs per host*

In the previous experiments, we placed each LP in a different PE. Figure 7 shows the WCT when more than one LP is placed in a PE. In particular, we consider the following scenarios: (*i*) 4 LPs placed over 4 PEs (1 LP per host), (*ii*) 8 LPs placed over 8 PEs (1 LP per host), (*iii*) 8 LPs placed over 4 PEs (2 LPs per host), and (*iv*) 16 LPs over 4 PEs (4 LPs per host). For each scenario, we consider the three failure scenarios already mentioned (no failures, crash, Byzantine failures). Also in these cases, the migration is disabled. Each curve in the figure is related to one of those scenarios, when varying the amount of SEs. It is worth noting that, when two or more LPs are run on the same PE, they can communicate using shared memory rather than by LAN.

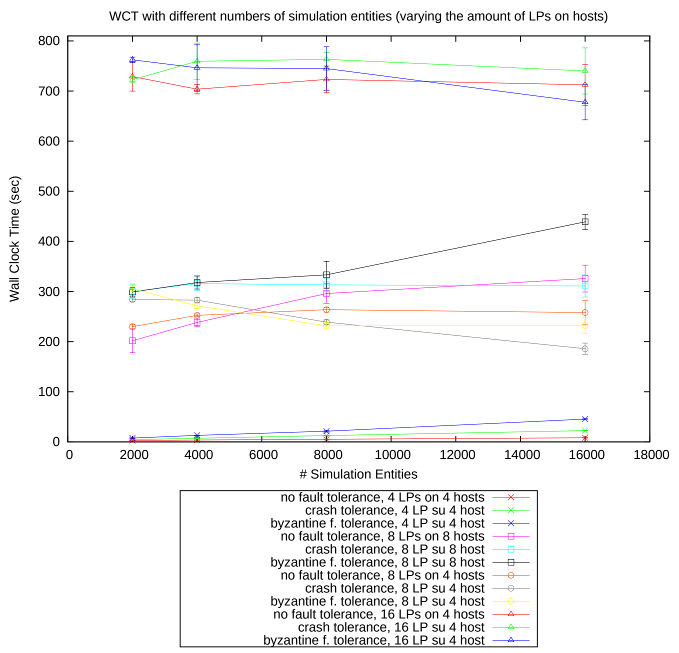

Fig. 7. WCT as a function of the number of LPs, with different numbers of LPs for each PE. Migration is disabled. Lower is better.

We observe that the scenario with 4 LPs over 4 PEs is influenced by the number of SEs and the failure scenario, while in the other cases it is the number of LPs that mainly determines the simulator performance. When 8 LPs are present, slightly better results are obtained with 4 LPs (rather than 8). This is due to the better communication efficiency (e.g. reduced latency) provided by the shared memory with the respect to the LAN protocols.

The worst performance is measured when 16 LPs are executed on 4 PEs. This is due to the fact that the amount of computation in the simulation model is quite limited. Therefore, partitioning the SEs in 16 LPs has the effect to increase the communication cost without any benefit under the computational point of view (i.e. in the model there is not enough computation to be parallelized).

D. *Impact of the number of failures*

We now study the impact of the number of faults on the simulation WCT. We consider two scenarios, one with 5 LPs over 5 PEs (Figure 8), and one with 8 LPs over 4 PEs (Figure 9). The choice of 5 LPs is motivated by the fact that this is the minimum number of LPs that allows us to tolerate up to 2 Byzantine faults. The scenario with 8 LPs on 4 PEs allows

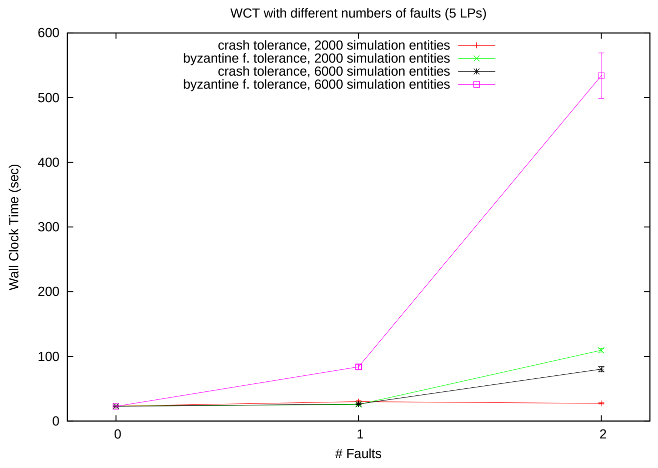

Fig. 8. WCT as a function of the number of faults; 10000 timesteps with 5 LPs. Migration is disabled. Lower is better.

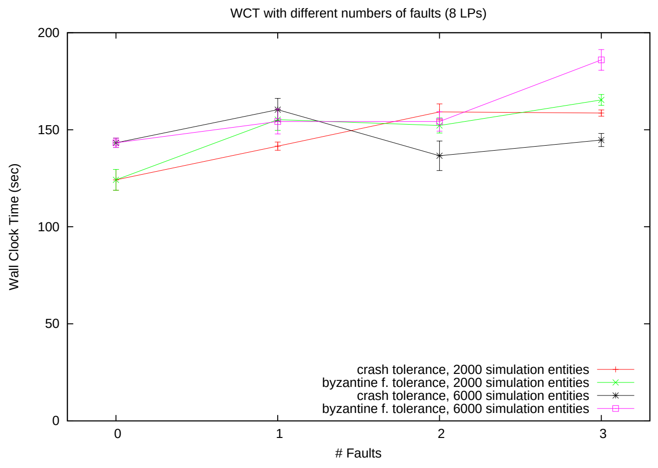

Fig. 9. WCT as a function of the number of faults; 2000 timesteps over 8 LPs. Migration is disabled. Lower is better.

testing 3 Byzantine faults with 2 LPs per hosts, reducing the communication overhead.

Figure 8 shows the WCTs measured with 0, 1 and 2 faults. Each curve refers to a scenario composed of 2000 or 6000 SEs with crash or Byzantine failures. As expected, the higher the number of faults, the higher the WCTs, especially when Byzantine faults are considered. Indeed, in this case a higher amount of communication messages is required among nodes in order to properly handle the faults.

A higher WCT is measured with 8 LPs, as shown in Figure 9. In this case, the amount of faults does not influence the simulation performance too much. As before, the computational load of this simulation model is too low for gaining from the partitioning in 8 LPs. In other words, the latency introduced by the network communications is so high that both the number of SEs and and the number of faults have a negligible impact.

E. *Impact of SEs migration*

Figure 10 shows the WCT with different failure schemes, when SEs migration is enabled/disabled. In this case, the trend obtained with the SEs migration is similar to that obtained when no migration is performed but the overall performance are better when the migration is turned off. This is due to the overhead introduced by the self-clustering heuristics and the SEs state that is transfered between the LPs. In other words, the adaptive clustering of SEs, in this case, is unable to give a speedup.

It is worth noting that, in this prototype, we have decided to use the very general clustering heuristics that were already implemented in GAIA/ARTÌS. We think that, more more specific heuristics will be able to improve the clustering performance and therefore balance the overhead introduced by the support of fault tolerance.

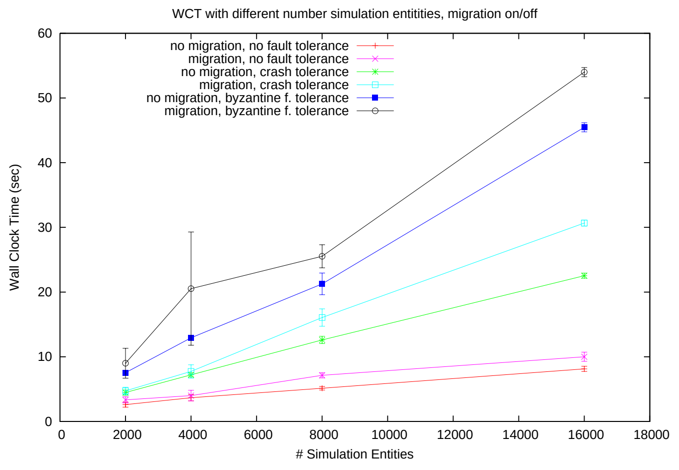

Fig. 10. WCT with SEs migration ON/OFF, as a function of the number of SEs. Lower is better.

VI. Conclusions and Future Work

In this paper we described FT-GAIA, a software-based fault-tolerant extension of the GAIA/ARTÌS parallel and distributed simulation middleware. FT-GAIA transparently replicates simulation entities and distributes them on multiple execution nodes. In this way, the simulation can tolerate crash-failures and Byzantine faults of computing nodes. FT-GAIA can benefit from the automatic load balancing facilities provided by GAIA/ARTÌS that allow simulated entities to be migrated among execution nodes. A preliminary performance evaluation of FT-GAIA has been presented, based on a prototype implementation. Results show that a high degree of fault tolerance can be achieved, at the cost of a moderate increase in the computational load of the execution units.

As a future work, we aim at improving the efficiency of FT-GAIA by leveraging on ad-hoc clustering heuristics. Indeed, we believe that specifically tuned clustering and load balancing mechanisms can significantly reduce the overhead introduced by the replication of the simulated entities.

ACRONYMS

DES	Discrete Event Simulation
FEL	Future Event List
GVT	Global Virtual Time
IRP	Inertial Reference Platform
LVT	Local Virtual Time
LP	Logical Process
MTTF	Mean Time To Failure
PADS	Parallel and Distributed Simulation
PE	Processing Element
SE	Simulated Entity
WCT	Wall Clock Time

REFERENCES

[1] R. M. Fujimoto, *Parallel and distributed simulation systems*, ser. Wiley series on parallel and distributed computing. Wiley, 2000.

[2] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *Computers, IEEE Transactions on*, vol. 61, no. 6, pp. 767–779, 2012.

[3] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley, 1998.

[4] M. Dowson, "The ariane 5 software failure," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, pp. 84–, Mar. 1997.

[5] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

[6] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello, "A new adaptive middleware for parallel and distributed simulation of dynamically interacting systems," in *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 178–187.

[7] ——, "ARTÌS: A parallel and distributed simulation middleware for performance evaluation," in *ISCIS*, ser. Lecture Notes in Computer Science, C. Aykanat, T. Dayar, and I. Korpeoglu, Eds., vol. 3280. Springer, 2004, pp. 627–637.

[8] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[9] O. P. Damani and V. K. Garg, "Fault-tolerant distributed simulation," in *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation*, ser. PADS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 38–45.

[10] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[11] M. Eklöf, F. Moradi, and R. Ayani, "A framework for fault-tolerance in hla-based distributed simulations," in *Proceedings of the 37th Conference on Winter Simulation*, ser. WSC '05. Winter Simulation Conference, 2005, pp. 1182–1189.

[12] M. Eklof, R. Ayani, and F. Moradi, "Evaluation of a fault-tolerance mechanism for hla-based distributed simulations," in *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 175–182.

[13] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules," IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), pp. 1–38, 2010.

[14] D. Chen, S. J. Turner, W. Cai, and M. Xiong, "A decoupled federate architecture for high level architecture-based distributed simulation," *Journal of Parallel and Distributed Computing*, vol. 68, no. 11, pp. 1487 – 1503, 2008.

[15] J. A. Kohl and P. M. Papadopoulas, "Efficient and flexible fault tolerance and migration of scientific simulations using cumulvs," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 60–71.

[16] J. Lüthi and S. Großmann, *Computational Science - ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part III*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. FT-RSS: A Flexible Framework for Fault Tolerant HLA Federations, pp. 865–872.

[17] D. Agrawal and J. R. Agre, "Replicated objects in time warp simulations," in *Proceedings of the 24th Conference on Winter Simulation*, ser. WSC '92. New York, NY, USA: ACM, 1992, pp. 657–664.

[18] Z. Guessoum, J.-P. Briot, N. Faci, and O. Marin, "Towards Reliable Multi-Agent Systems. An Adaptive Replication Mechanism ," *International Journal of MultiAgent and Grid Systems*, vol. 6, no. 1, 2010. [Online]. Available: http://liris.cnrs.fr/publis/?id=4840

[19] G. D'Angelo and M. Marzolla, "New trends in parallel and distributed simulation: From many-cores to cloud computing," *Simulation Modelling Practice and Theory (SIMPAT)*, 2014.

[20] "Parallel And Distributed Simulation (PADS) research group," http://pads.cs.unibo.it, 2016.

[21] IEEE 1516 Standard, Modeling and Simulation (M&S) High Level Architecture (HLA), 2000.

[22] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, pp. 198–206, Apr. 1981.

[23] G. D'Angelo and S. Ferretti, "Simulation of scale-free networks," in *Proc. of International Conference on Simulation Tools and Techniques*, ser. Simutools '09, 2009, pp. 20:1–20:10.

[24] ——, "LUNES: Agent-based Simulation of P2P Systems," in *Proceedings of the International Workshop on Modeling and Simulation of Peer-to-Peer Architectures and Systems (MOSPAS 2011)*. IEEE, 2011.

[25] J. Färber, "Network game traffic modelling," in *Proceedings of the 1st Workshop on Network and System Support for Games*, ser. NetGames '02. New York, NY, USA: ACM, 2002, pp. 53–57.