A Framework with Proactive Nodes for Scheduling and Optimizing Distributed Embedded Systems

Adrián Noguero[1], Isidro Calvo[2]

[1] European Software Institute,
Parque Tecnológico de Zamudio, #204, 48170, Zamudio, Spain
adrian.noguero@esi.es

[2] DISA (University of the Basque Country),
E.U.I. de Vitoria-Gasteiz, C/Nieves Cano, 12, 01006
isidro.calvo@ehu.es

Abstract. A new generation of distributed embedded systems (DES) is coming up in which several heterogeneous networked devices execute distributed applications. Such heterogeneity may apply to size, physical boundaries as well as functional and non-functional requirements. Typically, these systems are immersed in changing environments that produce dynamic requirements to which they must adapt. In this scenario, many complex issues that must be solved arise, such as remote task preemptions, keeping task precedence dependencies, etc. This paper presents a framework aimed at DES in which a central node, the Global Scheduler (GS), orchestrates the execution of all tasks in a DES. The distributed nodes take a proactive role by notifying the GS when they are capable of executing new tasks. The proposed approach requires from the underlying technology support for task migrations and local preemption at the distributed nodes level.

Keywords: Distributed embedded systems, Framework, Reconfigurable Architectures, Middleware

1 Introduction

Nowadays, there is a clear trend in the software industry to create distributed systems from already designed components. This trend, which is especially relevant in the embedded systems industry, has promoted the use of middleware technologies such as Java-RMI, CORBA/e, CORBA-RT, ICE, DDS or even SOA architectures. Indeed, distributed computing has proven its added value especially in some application domains, such as multimedia telecommunications, manufacturing, avionics or automotive [1], [2].

Typically, middleware technologies abstract the details of underlying devices facilitating the creation of distributed applications by providing uniform, standard, high-level interfaces to developers and integrators. Another objective of middleware technologies is supplying services that perform general purpose functions in order to

avoid duplicating efforts and facilitating collaboration among applications. This paper sticks to the second objective. Namely, it presents a middleware framework for DES aimed to manage the deployment and execution of a set of tasks in a set of distributed nodes, so the time requirements of the overall system are met and the use of the resources of the distributed nodes is optimized (e.g. CPU, volatile / non-volatile memory or battery). The proposed framework allows deploying tasks to the nodes in run-time for achieving a better overall optimization.

The framework uses a set of entities, namely the Global Scheduler (GS) and the Remote Servers (RS), which provide the infrastructure to execute the tasks at the distributed nodes. In particular, the GS orchestrates the execution of all tasks at the DES according to scheduling and optimization policies implemented as pluggable components. The RS act as local managers responsible for executing application tasks at the distributed nodes.

Even though typically DES must be configured at start-time, the framework allows dynamic reconfigurations of the system at run-time, providing a certain degree of flexibility and adaptability to changing requirements. These reconfigurations include modifications of both functional requirements at run-time, such as updating the executable code of the applications, as well as non-functional requirements, such as changing QoS parameters or introducing new nodes or removing existing ones without impacting the functionality of the system. These characteristics, which allow optimizing the computational load of a distributed system by adding or removing tasks from the system without changing the underlying hardware, may be applicable in certain application domains such as distributed multimedia applications or home automation, in which changing the hardware may become a complex issue.

A centralized scheduling approach has been selected for the sake of flexibility since concentrating all the information of the system in one single node facilitates the coordination of the distributed nodes. However, this approach has some drawbacks since a single GS may become a critical point of failure. In the future the authors intend to introduce replicated GS in the framework for improving fault tolerance.

The layout of the paper is as follows: section 2 presents a description of some relevant works on this topic; section 3 describes the proposed software architecture; and lastly, in section 4 some preliminary conclusions are drawn and the future work on the topic is described.

2. Related Work

The implementation of DES has been a very important research topic in the last decades. Examples of early investigations on the field can be found in [3] and [4], where some of the first solutions applicable to DES were described.

More recent works on the field have explored the implementation of complex scheduling frameworks for distributed systems on top of popular middleware architectures, such as CORBA or Java RMI. Their main advantage lies in the flexibility and the implementation simplicity provided by the middleware layers, which enables developers to abstract from low level details of the distributed system, such as communication protocols, operating systems, etc. These works vary in vision

and scope. For example, references [5] and [6] focus on the timing requirements of a DES, providing a framework able to orchestrate task activations in time and, therefore, allowing DES designers to easily decouple the execution of periodic tasks that make use of the same resources. Other works, such as [7] and [8] implement control loops to change the timing characteristics of the distributed tasks in order to achieve schedulability and improve the overall performance of a DES. Lastly, some works focus on specific characteristics of DES, such as the management of aperiodic tasks, admission control strategies or task migrations strategies [9] [10].

The framework proposed in this paper addresses some of the same objectives as previous works, but it introduces some innovative aspects. Firstly, the proposed framework aims at merging scheduling and resource optimization in the same DES structure. Secondly, it follows a different approach based on proactive distributed nodes, instead of reactive nodes fully dependent on the decisions of a global scheduler. Also, the proposed framework allows dynamic reconfigurations in run-time (e.g. changes on the number of nodes in the DES, software updates in the code of the tasks or changes in the tasks parameters). Finally, the proposed framework includes mechanisms to manage some of the complexities of DES, namely, precedence dependencies between tasks and remote preemptions.

3. Framework description

The proposed framework is composed of two component types: one Global Scheduler (GS), responsible for deploying and activating the tasks according to a predefined application graph (see Fig. 1) as well as an optimization criterion and several Remote Servers (RS), which encapsulate the distributed processors and execute the tasks. In this work applications are defined as the execution of a set of tasks following an application graph. As shown in the figure the tasks that compose an application may be executed in different nodes, being migrated from a node to another following to the decisions of the GS. Tasks are encapsulated with a special structure, known as Task Wrapper (TW), which contains not only the executable code, but also a set of parameters that characterize them.

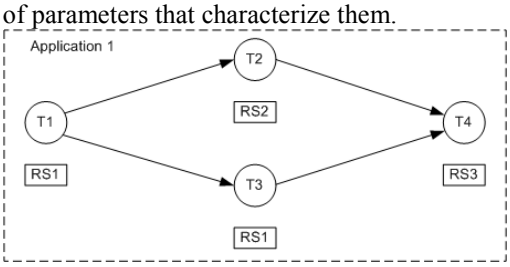

Fig. 1. Example graph of an application

Due to its characteristics, the nodes hosting the components of this framework must meet the following requirements:

- **Task migration**. The target software platform must support task migration between different nodes of the system. This can be achieved by explicitly sending the executable code of the application to the nodes of the system (e.g.

by using Java serialization). Sometimes, this restriction can be relaxed, reducing the flexibility of the framework, by previously deploying the tasks at the distributed nodes so they are activated by the central node.

- **Local schedulers**. In order to implement remote preemptions the framework requires the use of preemptive local schedulers. This role may be assumed either by a local preemptive OSs, or by dedicated local schedulers which must be able to preempt local tasks when required by the framework.

Any DES whose nodes meet the latter requirements is candidate for the implementation of the proposed framework. Task migrations may represent a considerable overhead when the required execution times of the tasks are similar to their migration times. Hence, the framework is applicable to applications where the execution times are much longer than migration times. Possible target applications may be found in the multimedia applications domain. As a matter of example, a Java based intelligent surveillance system formed by different types of nodes such as IP cameras, network video recorders, control centers and video analysis and streaming nodes, could be candidate for applying this approach since Java technology allows task migration via object serialization and the distributed nodes are capable of using local schedulers. Also, this kind of system requires an extensive use of CPU that may justify task migrations. Moreover, the optimal use of physical resources, such as CPU, memory or battery, justifies the implementation of optimization policies to improve the overall performance of the applications.

3.1 Functional overview

Briefly, the proposed architecture works as follows. The GS activates the tasks that compose the applications and orders them according to the application graphs and the scheduling and optimization policy. Since the GS is in charge of the activation of the periodic tasks, it needs a global timing reference for the whole DES, which defines a minimum granularity of the invocations at the DES. This timing reference unit will be called the *Elementary Cycle* (EC) and it will described below.

RS abstract the distributed processors in charge of executing tasks. Whenever an RS completes the execution of all assigned tasks, it notifies the GS its availability with a *WorkCompleted* message as depicted in Fig 2. The GS reacts by selecting the next task in the queue and deploying it to the free RS.

Deploying a task to an RS typically involves migrating a task from the GS to this RS; however, since task migrations have a great impact on the overall performance of the framework each RS is equipped with a sort of cache memory. So, depending on the status of the cache of the target RS, the GS may carry out three different deployment types. If (1) the deployed task is not in the cache of the target RS and it has enough memory for holding the new TW a regular deployment is performed and the TW object is physically transmitted to the RS. Else, (2) if the deployed TW is already in the cache of the target RS, the TW object is not sent; instead, the GS only sends the input parameters to run the task. Lastly, (3) if the deployed task is not in the cache of the target RS and it has not enough memory to keep it, an overwrite deployment takes place. In such case, the TW is sent to the RS, and it overwrites an older TW from the cache of the RS to store the new task.

To implement this functionality the GS keeps a set of ordered task queues, associated to every RS of a DES, that are updated when changes are produced. After each update, the task located in the first position of the queue is considered by the GS as the highest priority task to be deployed to the RS associated with that queue.

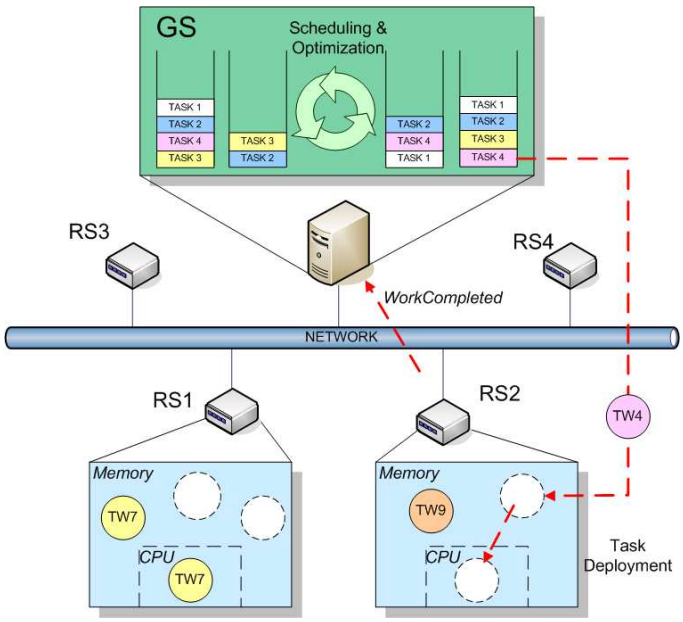

Fig. 2. Overview of the proposed architecture

Often, distributed applications are formed by a combination of tasks that are executed in a concrete sequence. This implies that every task may have one or several predecessors and successors. Tasks may also require inputs from other tasks to perform their work. This information is modeled in the GS by an application graph which is built from the TW. The GS uses it in run-time to compose the applications and to select the more appropriate nodes for optimizing the use of the resources according to the selected criteria. The communications between a task and its successors are centralized by the GS which receives the outputs of the completed tasks from the RS and hand them over as inputs to the next tasks in the graph. Predecessor and successor tasks along with input and output elements are included in the TW structure, as it will be further explained along with the system characterization.

The proposed framework assumes that tasks can be executed in any RS of the DES; however there will be cases in which some tasks will only be executable in certain nodes, e.g., when a specific library or hardware is required. The framework proposes the use of node bindings for modeling these requirements that will specify the list of nodes where a task may be executed.

In order to achieve a soft real-time behavior, it is necessary to implement a remote task preemption mechanism. The proposed framework relies on the local schedulers of the distributed nodes to implement a simple preemption mechanism. Preemptions

are triggered by the GS when a critical situation is detected, that is, when the laxity of a task, defined as its time to deadline minus its remaining execution time, becomes too short. If so happens the GS triggers a special deployment routine called *PreemptionDeployment*, which deploys the critical task to a non-free RS. This kind of deployment stops the running task and executes the newly deployed one.

One of the key benefits of the proposed framework is its high level of dynamism. The GS provides a reconfiguration interface to dynamically add, remove or change the tasks in the system. The GS is equipped with an admission control system to prevent changes that could lead the system to unschedulable situations. This approach allows changes in the applications ensuring certain QoS parameters.

Certain characteristics of the presented framework have direct implications in its behavior. For example, the authors have chosen a centralized approach because it provides more flexibility and higher scalability even though it may reduce fault tolerance. In future works the authors will introduce replicated GS in order to improve fault tolerance. Task migrations are also a key challenge since they may affect negatively the performance of the system. In future works the authors will quantify the impact of task migrations if different scenarios. Lastly, the presented framework does not consider shared resources dependencies among tasks as this issue can be worked around splitting the tasks and using precedence dependencies.

3.2 System characterization

The middleware framework described in the previous section requires keeping in memory a model of the tasks in the DES. Indeed, both the GS and the RS use a special structure to abstract the concept of a task, the Task Wrapper (TW). A TW not only models the task timing characteristics and precedence dependencies but also contains the logic of the application. Fig. 3 depicts this structure using UML notation and Java-like types for simplicity.

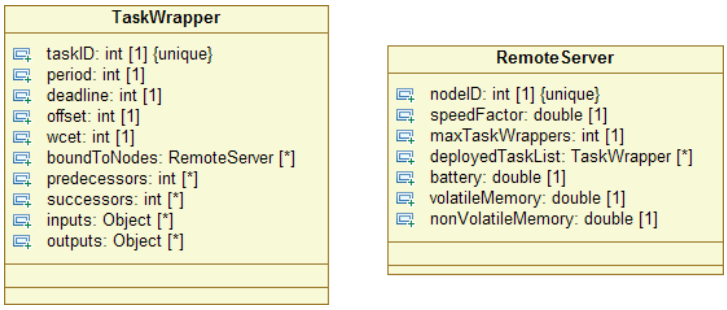

Fig. 3. Characterization of the Remote Servers and Tasks

The proposed task model requires a unified timing reference to be used in the entire DES. For the sake of consistency, all time measurements have been referred to an abstract time unit, the Elementary Cycle (EC), which is defined in a centralized way in the Global Scheduler as the minimum time between two subsequent task activations and specifies the time granularity of the system. The EC must be

configured during the start-up phase of the GS and it cannot be modified by any means in run-time. As a consequence, all the timing parameters of the TW model are referred to this parameter.

The task model is composed by the following parameters per TW:

- **Task identifier**. Used to identify univocally a task in the distributed system.
- **Timing parameters**. This set of parameters characterizes the timing properties of a TW. They are all defined as integer multiples of the EC. These parameters include: *period*, *deadline*, *offset* and *worst-case execution time (wcet)*. Note that the *wcet* parameter is referred to a node with a unitary speed factor as detailed in the RS description section below.
- **Precedence graph management parameters**. This group of parameters includes information related to the application graph such as *predecessors*, *successors*, *input data* (*inputs*) and *outcomes* (*outputs*) of one task. The GS uses these parameters to build the precedence graph in order to activate the tasks in the graph in order and manage the inputs and outputs involved.
- **Bound to nodes**. This parameter is used to attach one TW to a specific node or list of nodes. It may be used when a node owns a specific hardware resource or software component (e.g. a library) required for the execution of a task.

The framework also requires the GS to keep information about the status of the RS in the DES, since this information is essential for applying optimization policies. Therefore, the GS maintains a model in memory that represents the DES using a *RemoteServer* data structure per RS (also included in Fig. 3). Regarding the RS model, the following parameters are considered:

- **Node identifier**. Integer value that identifies univocally an RS in the distributed system.
- **Physical parameters.** Namely *speedFactor*, *maximumTaskWrappers*, *deployedTaskList*, *battery* and *volatile* / *non-volatile memory*. This group of parameters models the physical status of an RS. They are used to implement optimization policies in the GS. Special attention should be given to the *deployedTaskList* parameter, since it keeps a record of the current status of the cache of each RS. Also, the *speedFactor* parameter models the actual processing power of a node, compared with a reference node with a unitary speed factor.

3.3 Structure of the Global Scheduler

As shown in Fig. 4 the GS is a modular entity composed by four active components; namely (1) *Activator*, (2) *RSInterface,* (3) *ReconfigurationInterface* and (4) *PreemptionManager*. Along with these components the GS also uses several data structures that represent the current status of the DES.

For each connected RS, the *TaskQueuesManager* (see element #5 of Fig. 4) keeps an ordered queue with all the tasks that must activated at that node. These tasks are ordered according to a scheduling policy and then, the ordering is refined according to an optimization policy. A reordering is committed every time a new task is placed in a

queue. As shown the figure, the framework is open to be used with different scheduling policies such as RMS, EDF or MUF, which are connected to the queues as pluggable components. Similarly different optimization policies can be connected to the queues.

Additionally, the GS maintains a *SystemModel* (element #6), which is a data structure updated every EC by the *Activator* (element #1). This model is used to keep track of the remaining times for the next activation of each periodic task, the current laxities of the active tasks and the current status of the connected RS.

Apart from updating the *SystemModel*, the *Activator* is in charge of adding the tasks activated every EC to the queues. Therefore, it also manages the precedence dependencies between tasks.

All communications between the GS and the RS are handled via the *RSInterface* (element #2). Additionally, whenever the *RSInterface* receives a *WorkCompleted* message from any RS, it is the *RSInterface* itself who deploys the next task in the queue to the requesting RS and updates the *SystemModel* accordingly.

Users and applications may interact with the GS to require dynamic reconfigurations at runtime through the *ReconfigurationInterface* (element #3). This interface allows changes in the task parameters (period, deadline, etc.) as well as adding or removing tasks to the system. All changes are recorded in the *SystemModel*; however, before any changes are committed this component executes an admission control test that checks whether the new configuration is feasible. This functionality is provided by a pluggable component that may be exchanged to use different admission policies.

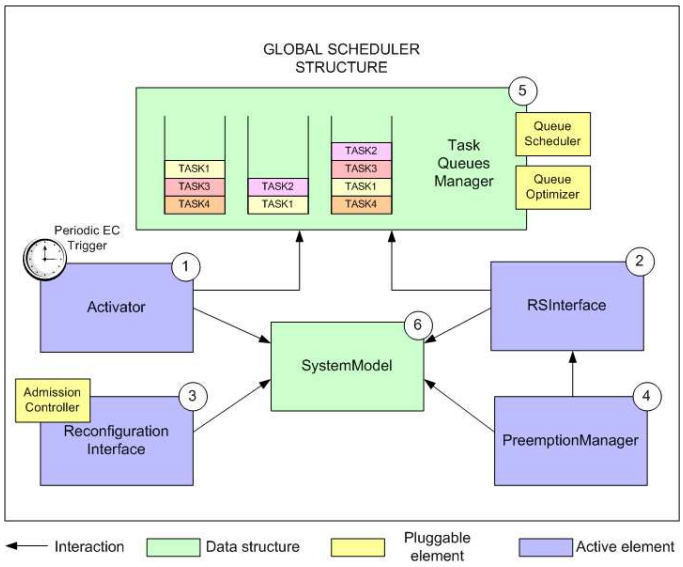

Fig. 4. Structure of the Global Scheduler

Finally, the GS implements a *PreemptionManager* (element #4) module whose objective is to prevent tasks from missing their deadline. It uses the tables in the *SystemModel* to detect potential deadline misses. Should any problems be detected,

the *PreemptionManager* would instruct the *RSInterface* to activate a preemption deployment routine to one or more RS.

3.4 Structure of the Remote Servers

An RS is an entity that manages only one processor of the DES (see Fig. 5). The main role of an RS is to execute the tasks deployed by the GS and, when all assigned tasks are completed, declare its availability to the GS via a *WorkCompleted* message.

Communications with the GS are handled via the *GSInterface* component (element #1). When a new TW is received it is stored in the *DeployedTaskList* (element #3). This element plays a similar role to the caches in processors, allowing the GS to reduce deployment times when a TW is already loaded in an RS, and improving the overall performance of the framework. TW are executed by the *TaskExecutor* component (element #2), which is capable of starting, stopping and resuming the execution of a TW and sends the *WorkCompleted* messages to the GS with the results of each task when they are completed. It also implements a mechanism that allows preempting a TW in execution with another. When an RS receives a preemption deployment message the TW in execution is put in the *PreemptedTaskList* (element #4) and the *TaskExecutor* starts the execution of the new task. When the latter task terminates its execution, the task executor notifies the GS with a special *TaskCompleted* message which includes the results of the completed task to the GS to be handed over to successor tasks, and continues to work until all tasks in the *PreemptedTaskList* have been executed.

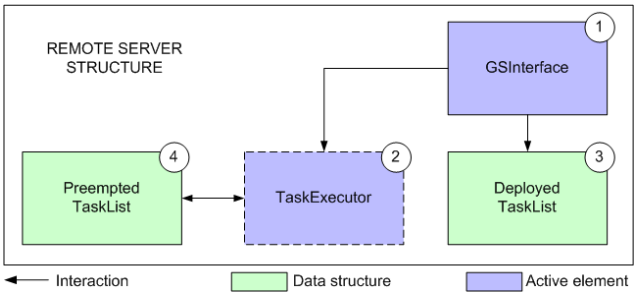

Fig. 5. Structure of the Remote Server (RS)

4 Conclusions and Future work

This paper presents a middleware framework for DES that supplies a set of services aimed to manage the deployment and execution of a set of tasks in a set of distributed nodes, so the time requirements of the overall system are met and the use of the resources is optimized according to different criteria (e.g. use of CPU, memory or battery). In particular, it provides a timing reference for the whole DES, support for executing in order the tasks of a DES, a certain degree of dynamism in run-time to adapt to changing requirements (e.g. task parameters, code updates) and a

reconfiguration interface to require these changes. This framework is aimed at DES that allow task migration and that may use local schedulers at the nodes. It uses a central modular component (known as GS) that orchestrates the system, another component type (RS) to abstract the individual processors involved in the DES and a special structure that abstracts the concept of task (TW). The GS is built in a modular way so the designers of the applications may easily choose from different scheduling and optimization policies the one that suits best to their applications.

In the future the authors will introduce replicated GS in the framework to improve fault tolerance and will evaluate the performance of the framework as well as its behavior using the real-time Java implementation. Special care will be taken regarding task migration costs and memory consumption issues.

Acknowledgments. This work has been supported by the ARTEMIS JU through the iLAND project 2008/10026, the Basque Government through the TEReTRANS project IE08-221 and by the University of the Basque Country (UPV/EHU) through grant EHU09/29.

References

1. OROCOS, "The OROCOS project – Smarter control in robotics and automation", Available at: http://www.orocos.org/
2. Real-Time CORBA with TAO, "ACE and TAO success stories", http://www.cs.wustl.edu/~schmidt/ACE-users.html
3. S. Levi, S. K. Tripathi, S. D. Carson, A. K. Agrawala. "The MARUTI Hard Real-Time Operating System", SIGOPS Operating Systems Review, Vol. 23, Is. 3, pp. 90-105, 1989.
4. J.A. Stankovic, K. Ramamritham, M. Humphrey, G. Wallace, "The Spring System: Integrated Support for Complex Real-Time Systems", *International Journal of Time-Critical Computing Systems*, Vol 16, pp. 223-251, 1999.
5. I. Calvo, L. Almeida, A. Noguero. "A Novel Synchronous Scheduling Service for CORBA-RT Applications", *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC07*, pp 181-188, 2007
6. P. Basanta-Val, I. Estévez-Ayres, M. García-Valls, L. Almeida. "A synchronous scheduling service for distributed real-time Java", *IEEE Transactions on Parallel and Distributed Systems*, Accepted for future publication, 2009.
7. X. Wang, C. Lu and C. Gill. "FCS/nORB: A feedback control real-time scheduling service for embedded ORB middleware", *Microprocessors and Microsystems*, June 2008.
8. V. Kalogeraki, P. M. Melliar-Smith, L. E. Moser, Y. Drougas. "Resource management using multiple feedback loops in soft real-time distributed object systems", *The Journal of Systems and Software*, Vol. 81, pp. 1144-1162, 2008.
9. Y. Zhang, C. Lu, C. Gill, P. Lardieri, G. Thaker. "Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems". *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium RTAS*, pp. 113-122, 2007.
10. Y. Zhang, C. Gill, C. Lu, "Reconfigurable Real-Time Middleware for Distributed Cyber-Physical Systems with Aperiodic Events", Proceedings of the 28th International Conference on Distributed Computing Systems ICDCS08, pp.581-588, 2008.