Communication Paradigms for High-Integrity Distributed Systems with Hard Real-Time Requirements

Santiago Urueña, Juan Zamorano, José A. Pulido, and Juan A. de la Puente

Abstract The development and maintenance of high-integrity software is very expensive, and a specialized development process is required due to its distinctive characteristics. Namely, safety-critical systems usually execute over a distributed embedded platform with few hardware resources which must provide real-time communication and fault-tolerance. This work discusses the adequate communication paradigms for high-integrity distributed applications with hard real-time requirements, and proposes a restricted middleware based on the current schedulability theory which can be certified and capable to obtain the required predictability and timeliness of this kind of systems.

1 Introduction

On-board embedded computers play a crucial role in spacecrafts, where they perform both platform control functions, such as guidance and navigation control or telemetry and tele-command management, and payload specific functions, such as instrument control and data acquisition. One distinctive characteristic of on-board computer systems is that hardware resources are scarce, due to the need to use radiation-hardware chips and limitations in weight and power consumption, and these resources are distributed due to the physical distance between the instruments and to replicate mission-critical components. Another key aspect of these systems is the presence of high-integrity and hard real-time requirements, which raises the need for a strict verification and validation (V&V) process both at the system and software levels [1]. This new step in the development process is called *certification*. It is a very expensive process which will shape the complete development tools and methods of the system.

Santiago Urueña, Juan Zamorano, José A. Pulido, and Juan A. de la Puente
Technical University of Madrid (UPM), Dept. of Telematic Systems Engineering (DIT), Spain.
e-mail: `{suruena,jzamorano,pulido,jpuente}@dit.upm.es`

It is worth noting that inside a high-integrity system not all the software has the same criticality: while some applications have a direct implication in the safety of the system, a fault in other parts of the code will result only in minor effects [2]. Therefore, not all the software is certified to the highest *criticality level* to save costs. *Ravenscar* is a computational model designed for high-integrity, hard real-time, embedded systems [3]. It is a profile that specifies the set of operations that the real-time operating system (RTOS) has to provide, and also the set of forbidden operations that would made the system unpredictable. On the one hand, Ravenscar compliant real-time kernels have to provide less functionality than other RTOSs, and therefore they are be smaller and easier to certify. On the other hand, applications developed under the Ravenscar restrictions are suitable to temporal analysis.[1]

Due to the specific characteristics of this kind of systems a general purpose middleware cannot be used to develop high-integrity code. The objective of this paper is to describe the design principles used in a safety-critical middleware for the European Space Agency (ESA), discussing the most adequate communication paradigms and the requirements of a high-integrity middleware. In the end, the main goal is to be able to analyze statically the schedulability of its hard real-time deadlines. This paper is organized as follows. Section 2 describes the contributions and related work, while Section 3 sets the computational model. Section 4 defines a set of restrictions for building safety-critical distributed systems, including the implementation requirements and an analysis of the adequate communication paradigms. Finally, Section 5 summarizes the main conclusions of this work.

2 Contributions and related work

This paper builds upon current advances in scheduling theory for distributed hard real-time systems. Tindell and Clark [4] extended the response time analysis techniques used for event-triggered single processors to distributed systems, introducing the concept of holistic schedulability. Later, Palencia and González Harbour [5] improved the technique to reduce the pessimism of transactions.

The main objective of this work was the development of a Ravenscar-compliant middleware for next-generation space-crafts. Specifically, the main contributions of this paper are:

1. Specification of the distribution requirements of the aero-space industry;
2. Modelization and response time analysis of the specific distributed system;
3. Restrictions needed for a safety-critical middleware, and adequate communication paradigms.

Some of the restrictions specific to the Ada programming language were published previously by the authors [6], but this paper extends that work and make the requirements language independent.

[1] Actually *Ravenscar* is a village in England, where experts from industry and academy in high-integrity and hard real-time systems met to define the profile.

Kopetz elaborated the Time-Triggered Architecture (TTA) [7] to provide hard real-time communication for safety-critical distributed systems. However, a time-triggered middleware presents similar scalability problems for development a maintenance than cyclic executives. In contrast, a Ravenscar-compliant middleware supports time-triggered and event-triggered programming. Some past publications about the specific topic of Ravenscar-compliant distributed systems exists [8], but only discussing the research challenges.

3 Computational Model

3.1 Industrial requirements

The following list of requirements has been extracted from the needs expressed by different companies from the aero-space industry. Namely, they represent the middleware requirements found during the development of different projects for the European Space Agency (ESA), including self-maintained long-term satellites, mission-critical unmanned space vehicles, and satellite fleets:

- **Predictability**: End-to-end transfers in bounded time for messages with hard real-time deadlines.
- **Fault tolerance**: Replication of network links and/or routers for resilience to hardware failures.
- **Diagnostic information**: The application should be able to know the status of a node and communication links.
- **Multicast communication**: One-to-many communication, even if the network does not support broadcasting operations.
- **Message segmentation**: The partitioning of messages greater than the maximum transfer unit should be done by the middleware.
- **Message forwarding**: Transparent communication between nodes not directly connected.

Some of these requirements complicate the implementation of the middleware and the static analysis of the whole system. However, it should be noticed that not all these requirements are needed in every application nor in every criticality level. In fact, the system integrator should be able to disable the unwanted functionality at design time to ease the certification of the system and to reduce the performance penalty. Therefore, the middleware must be tailorable at compilation time to be adapted to specific application needs.

3.2 Restrictions for the RTOS

As said above, the certification entailed by every safety-integrity system shapes its development process, thus a strict set of restrictions is needed when developing high-integrity software. These are the main restrictions dictated by the Ada 2005 Ravenscar Profile [9, § D.13.1] for the RTOS:

- A static number of threads and shared resources
- No thread termination (and no abortion)
- No dynamic memory at the kernel level
- Only a single thread can wait on a given condition variable

In addition, the threads are scheduling according to Fixed Priority Preemptive Scheduling (FPPS), using the Immediate Ceiling Priority Protocol (ICPP) for shared resources [10]. Thanks to these restrictions the implementation of the kernel is small enough to be certified, while offering a sufficient set of services which allow the *schedulability analysis* of the application. Another derived advantage for embedded systems is that Ravenscar implementations require very low resources and have a high performance. In addition, the ICPP assures that deadlocks cannot ever occur, a highly desirable property specially for safety-critical systems.

4 A restricted middleware for High-Integrity Systems

4.1 Holistic schedulability analysis

Current mono-processor response-time analysis can evaluate a static number of *periodic* or *sporadic tasks* (i.e. threads), each having a worst-case execution time (WCET), and synchronize by using a static number of shared resources. The response-time analysis method has been also extended for distributed systems [4]. The holistic schedulability analysis assumes that each sender thread can send a fixed set of messages, and no thread can receive more than one message. In addition, each message must have a bounded size, and a fixed destination thread.

A *transaction* Γ_i, composed of a set of tasks $\tau_{i,j}$ with precedence relations, is another important concept for the response time analysis of distributed systems. The objective is to analyze the end-to-end response time of each transaction to asses the schedulability of the system. And although each task of the system has a unique priority, due to their precedence relations every task of a transaction (except the first one) is activated by the preceding task of the transaction, even if the second one has a higher priority. As a side note, the deadline of a task inside a transaction is usually longer than its period because the transaction can start another activation even if the last one is not still running.

For example (see figure 1), the transaction Γ_1 is composed of task $\tau_{1,1}$ (which runs over the node N_1), message $m_{1,1}$ (transmitted via the network A), and task $\tau_{1,2}$

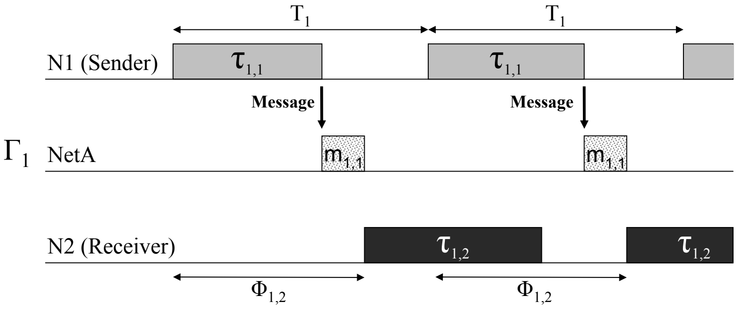

Fig. 1 Example of a distributed transaction.

(executing inside N_2). The transaction has a period T_1 (i.e. the number of times the transaction is activated per second), and $\tau_{1,2}$ has the offset $\Phi_{1,2}$ since the start of transaction. Thus the network can be modeled as a CPU (but messages cannot be preempted), and each message is like a task, with a fixed priority, a period, and a worst-case transmission time. The original holistic schedulability method had been improved with more exact response time analysis [5]. Later, the system model was extended with the analysis of multiple events [11], a message can activate more than one task, or also a task can be activated by multiple messages.

Although the computational model can be seen as too restrictive, it is rich enough to provide the common services needed in a safety-critical system. However, current response time analysis techniques require a *single activation point* for each task (either an event for sporadic tasks or a timer for periodic ones). But in some communication paradigms including Remote Procedure Call (RPC) and Remote Method Invocation (RMI), the client thread sends a message to the server, and blocks until the other thread sends another message with the response (two activation points). A general method should be developed to analyze more than one activation point.

4.2 Modelization of synchronous calls

In this paper, each thread is modeled by $n+1$ tasks inside a transaction, where n is the number of activation points of the thread. As can be seen in figure 2, although the transaction Γ_2 is composed by two *threads*, it is modelled as three *tasks*:

1. the sender thread sends a query to the server, and then performs a blocking receive operation (task $\tau_{2,1}$).
2. the server thread processes the petition and then sends-back the response (task $\tau_{2,2}$).
3. finally, the message wakes-up the client thread and reads the answer (task $\tau_{2,3}$).

Although a RPC or RMI can be modeled using this technique, the analysis is not completely accurate because multiple activations of a transaction can be executing at

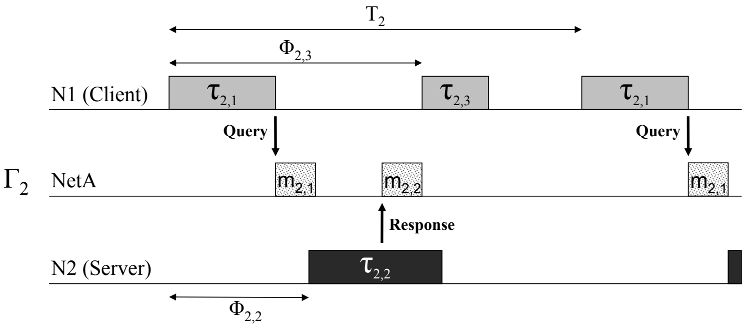

Fig. 2 Model of a Remote Procedure Call.

the same time unlike an RPC (the client thread cannot start another activation until the last one is complete). Blocking operations are required not only for RPC-like operations, but also for message segmentation, and one-to-many communication in a point-to-point topology or for networking technologies with no broadcast support. It is worth noting that not all transactions are distributed. For example, using this technique it is possible to model blocking system calls, like a read operation from a file where the thread is blocked until the information is read from the hard disk. However, all the tasks of the same thread share the same priority, thus the response time analysis methods must be extended to cope with non-unique priorities.

4.3 Implementation requirements

Multiple design choices were studied when developing the middleware for this specific ESA project. It is desirable that a task invoking a remote operation does not delegate the message generation (including data marshalling, message partitioning, composition of message headers, and even message queueing) to another task to avoid priority inversion. *Priority inversion* is a undesired effect typically found when a task cannot execute until a lower-priority task exits from a shared resource. Total priority inversion is in general not possible but it can (and must) be bounded. From the point of view of the middleware, if the message generation is done by a specific task of the communications stack then a high-priority task will be preempted by this task even if the message is sent by another task with the lowest priority of the node. Therefore, it is encouraged that the middleware code for message generation is executed directly by the sender task, i.e. with its priority.

For the transfer of the message, there are typically two possible implementations:

1. **middleware thread**: the sender task puts its message into a buffer, which will be sent by a sporadic thread of the middleware.
2. **self service**: the sender task calls the device driver directly.

The advantage of the first implementation is that the sender task can be completely asynchronous. In contrast, the self-service model should have to wait until the message is completely transferred to know the status of the sent operation. In the other hand, the self-service implementation has a lower priority inversion.

Therefore, if the remote operation is asynchronous, the call to the middleware can be fully non-blocking. However, if the operation is synchronous (e.g. a RPC) the call will be blocking, and in addition the middleware must set a timer to detect a communication problem, e.g. the message was lost or the receiver node is not responding. Otherwise, the sender thread will be blocked forever. It is worth noting that message acknowledgement and retransmissions are not usually done by software in a safety-critical distributed system because guaranteed delivery is provided by the hardware communication bus.

Communication networks also introduce some priority inversion: The network is normally non-preemptable, so if a low-priority message is being transferred then another message with a higher priority cannot be sent until that frame is completely transmitted. For that reason, the maximum size of a message must be bounded. Of course, in the first implementation, the middleware thread will sent the output-messages by priority.

At the destination node, the receiver thread should then process each call with the priority specified in the message. The above guideline about message generation is also applicable at the receiver side of the middleware: it is desirable that the composition and unmarshalling of the message are performed directly by the receiver task. It should be noticed that each partition can still have an independent run-time system. No clock synchronization is needed because the communication is message oriented [12, p. 1.27], but of course a mechanism to obtain a certain degree of common time is desirable in a real-time system.

In summary the implementation must document the architecture of the middleware, specifying if any step is delegated by another task in the caller or called node. Also, the metrics of the maximum blocking time of the biggest critical section should also be documented, otherwise a complete response time analysis of the whole system would be not possible.

4.4 Restrictions for the middleware

In addition to the restrictions for the RTOS explained in section 3.2, another set of constraints is needed for safety-critical middlewares. As said above, the schedulability theory assumes a static computational model, where the number of connections and messages does not change at all during the mission. That is, there is a *static number of nodes*, where *no dynamic connections* are allowed, and where all the nodes perform a *coordinated initialization* to start the application at the same time (in a real-time system it is not acceptable to enqueue a request until the server node is active).

Nodes are not allowed to stop its execution, as enforced by the RTOS restriction about no thread termination. And if the connections are not dynamic, there is a static number of messages, and each one has a fixed origin and destination, as well as a fixed priority. Finally, the computational model also assumes *bounded size messages* to be able to compute the maximum transfer time. This does not mean that each message has a fixed size but a maximum size limit.

Another implicit restriction is that *no concurrent remote calls* are allowed. Therefore, while in a general-purpose middleware usually a thread pool serves all requests—including calls to the same remote operation at the same time—in this restricted middleware there is a unique thread per remote operation that receives and processes each message. It is worth noting that an interesting property derived from this restriction is that distributed deadlocks are not possible in this restricted middleware, thus reducing the costs of the certification of the whole system [13].

In addition to the above restrictions which always must be enforced, there is also another set of **optional restrictions** which is not deemed essential for all safety-critical middlewares, but some kinds of distributed systems can benefit from it [6]. The key goal of these restrictions is to simplify the implementation of the middleware, thus facilitating its certification, and to ease the response time analysis of the system, reducing the main sources of pessimism and unpredictability. However, some of this restrictions has no impact in the implementation of the middleware, and even are difficult to detect violations statically.

The first optional restriction is to allow *asynchronous calls* only, i.e. to forbid all blocking remote operations (like a remote procedure call). A related restriction is "*no segmentation*", so only messages up to the MTU are allowed. This avoids a blocking send operation until all the parts of the message are sent. For the same reasons, "*no multicast*" is also needed if the hardware does not support the broadcasting of messages, however this restriction is always required to avoid the analysis of multi-event systems.

Finally, it can be useful to enforce the *no complex remote types* rule, i.e. a parameter of a remote operation cannot be an unconstrained or recursive type (e.g. linked list). With those types the exact size of the message cannot be computed until runtime, including its maximum size. So thanks to this restriction the maximum size of every message can be computed statically and thus the worst-case transfer time, and in addition the middleware does not need to handle the serialization of complex data [14].

4.5 Adequate Communication Paradigms

The communication paradigms supported in this Ravenscar-compliant middleware includes *message passing*, *remote procedure calls* (RPC), and *real-time publish/subscribe* (P/S). These paradigms can be implemented with little code, and they are supported by current response time analysis techniques to asses the schedulability of the system. But, due to its blocking nature, the RPC paradigm requires more code and timers than the the message passing or the P/S paradigm, and therefore it can be more difficult to certify.

However, although the Remote Method Invocation (RMI) can also be analyzed using similar techniques, in general it is difficult to ensure some restrictions in this communication paradigm. For example, although the number of distributed objects can be static, it is possible to send a remote reference to another node and therefore a new connection would be created at run-time, clearly violating the restriction about no dynamic connections. It is worth noting that OOP is not usually employed in safety-critical software due to its highly dynamic nature.

The Distributed Shared Memory (DSM) paradigm also presents some problems for safety-critical middlewares. The main advantage of DSM is that the programmer does not have to write explicitly the data transfer because at run-time the middleware transparently handles this, also easing the port of existing applications to distributed platforms. But this transparency is difficult to modelize and thus to perform the schedulability analysis of the application.

In summary, the message passing paradigm is well understood, and simple to learn, codify and analyze, and therefore it is very adequate for the development of high-integrity systems. The P/S paradigm, needed to fully meet the industrial requirements because it allows multicast communications, is also adequate for a safety-critical middleware because it can also be certified, although the response time analysis of multi-event systems can be more difficult to perform. The RPC paradigm can also successfully be used in a safety-critical middleware, although its blocking nature makes harder the certification at the highest-criticality levels.

However, as said above the RMI and DSM paradigms are the less adequate of the studied communication paradigms. Although *shared memory* can be used for inter-partition communication inside a node (e.g. among different criticality levels), DSM is not recommended for hard real-time communication in a safety-critical distributed system.

5 Conclusions and future work

All safety-critical systems must be certified prior deployment, and thus adequate development methods and tools must be used for this type of high-integrity software (like the Ravenscar profile). This heavily affects the middleware, which usually have to support hard real-time communication over a resource-constrained embedded platform.

This paper has described the design of a Ravenscar-compliant safety-critical middleware with hard real-time deadlines for future projects of the European Space Agency (ESA). After analysing the industrial requirements and the current schedulability theory for distributed systems, a set of restrictions and implementation and documentation requirements was proposed to allow certification of the middleware and to perform the response time analysis of distributed applications.

Finally, it was discussed the most adequate communication paradigms for this kind of systems. Simple paradigms like message passing or publish/subscribe are expressive-enough and can be implemented and analyzed more easily than remote procedure calls, distributed shared objects, or distributed shared memory.

Acknowledgements This work has been funded in part by the Spanish Ministry of Science and Technology (MCYT), project TIC2005-08665-C03-01 (THREAD), by the IST Programme of the European Commission under project IST-004033 (ASSERT), and by the Council for Education of the Community of Madrid and the European Social Fund.

References

1. ECSS. *ECSS-Q-80B Space Product Assurance — Software Product Assurance*, 2003. Available from ESA.
2. RTCA Inc. *Software Considerations in Airborne Systems and Equipment Certification — RTCA/DO-178B*, 2002.
3. ISO/IEC. *TR 24718:2005 — Guide for the use of the Ada Ravenscar Profile in high integrity systems*, 2005. Based on the University of York Technical Report YCS-2003-348 (2003).
4. Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2–3):117–134, April 1994. Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems).
5. Juan Carlos Palencia Gutiérrez and Michael González Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *RTSS 1999: Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 328–339, December 1999.
6. Santiago Urueña and Juan Zamorano. Building high-integrity distributed systems with Ravenscar restrictions. In Juan Antonio de la Puente, editor, *Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007)*, 2007.
7. Hermann Kopetz and Gnther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.
8. Neil Audsley and Andy Wellings. Issues with using Ravenscar and the Ada distributed systems annex for high-integrity systems. In *IRTAW '00: Proceedings of the 10th international workshop on Real-time Ada workshop*, pages 33–39, New York, NY, USA, 2001. ACM Press.
9. ISO SC22/WG9. *Ada 2005 Annotated Reference Manual. ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*, 2006. Available on `http://www.adaic.com/standards/ada05.html`.
10. Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Tr. on Computers*, 39(9), 1990.
11. J. Javier Gutiérrez, J. Carlos Palencia, and Michael González Harbour. Schedulability analysis of distributed hard real-time systems with multiple- event synchronization. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pages 15–24. IEEE CS Press, June 2000.
12. Juan Carlos Palencia Gutiérrez. *Análisis de planificabilidad de Sistemas Distribuidos de Tiempo Real basados en prioridades fijas*. PhD thesis, Universidad de Cantabria, 1999. Supervisor: Michael González Harbour.
13. César Sánchez, Henny B. Sipma, Zohar Manna, Venkita Subramonian, and Christopher Gill. On efficient distributed deadlock avoidance for real-time and embedded systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006*. IEEE Computer Society, April 2006.
14. Daniel Tejera, Alejandro Alonso, and Miguel Ángel de Miguel. Predictable serialization in Java. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'07)*, May 2007.