Novel Generic Middleware Building Blocks for Dependable Modular Avionics Systems

Christophe Honvault[1], Marc Le Roy[1], Pascal Gula[2],
Jean Charles Fabre[3], Gérard Le Lann[4], Eric Bornschlegl[5]

[1] EADS Astrium SAS, 31 avenue des Cosmonautes, 31402 Toulouse Cedex 4, France
{Christophe.Honvault, Marc.LeRoy}@astrium.eads.net
[2] AXLOG Ingéniérie, 19-21 rue du 8 mai 1945, 94110 Arcueil, France,
[3] LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse, France
[4] INRIA, Domaine de Voluceau - B.P. 105, 78153 Le Chesnay, France
[5] ESA/ESTEC, Keplerlaan 1 - P.O Box 299, 2200 AG Noordwijk, The Netherlands

Abstract. The A3M project aimed to define basic building blocks of a middleware meeting both dependability and real-time requirements for a wide range of space systems and applications. The developed middleware includes Uniform Consensus (UCS) and Uniform Coordination (UCN) protocols and two services implemented to solve two recurring problems of space applications: "distributed consistent processing under active redundancy" and "distributed replicated data consistency, program serialization and program atomicity". The protocols have been verified through extensive and accurate testing under the Real Time OS simulator RTSim supporting fault injections. The performances measured on a representative platform based on three LEON SPARC microprocessors interconnected with point-to-point SpaceWire links show that A3M solution may be applied to very different fields, from high performance distributed computing to satellite formation flying coordination.

1 Introduction and motivations

The complexity of the satellite and launcher on-board data management systems tends to increase rapidly, as well as the strictness of their performance requirements. The efficiency of their development depends on the availability and integration of standard software products (e.g. COTS) solving recurrent problems in space vehicle avionics systems. To address these problems, our objective in this project was to define a basic middleware support for a wide range of space systems and applications. Indeed, the evolution of space platforms is nowadays faced with the inherent distribution of resources and must meet strong real-time and dependability requirements. The handling of combined real-time constraints and fault tolerance strategies raises new problems and calls for new approaches. The A3M (*Advanced Avionics Architecture and Modules*) project was launched to address these issues. The objective was to develop a new generation of space platforms for a wide range of infrastructures, providing generic components as basic building blocks for the development of ad hoc middleware targeting various on-board space applications. The core components of this middleware enable real-time fault tolerant applications to be developed and reused without experiencing the traditional limitations of existing approaches.

The major components of the architecture rely on distributed fault-tolerant consensus and coordination protocols developed at INRIA [1]. These are based on *an asynchronous computational model*, thus making no assumption about underlying timing properties at design time. Therefore, the logical safety and liveness properties always hold without assuming the availability of a global time in the system. The interest of asynchronous solutions for safety-critical systems – choice made by ESA/ESTEC and EADS Astrium – is examined in [2] and explored in details in [3].

These facilities are key features from a dependability viewpoint, as most distributed fault tolerance strategies rely, in one way or another, on this kind of protocols. Their role is essential as far as replicated processing is concerned. For instance, they ensure that all replicas of a given software task receive the same inputs in the same order. Assuming that replicas are deterministic, the processing of the same inputs leads to the same results in the absence of faults. In addition, they are used to ensure that distributed scheduling decisions are consistent in a distributed system as a whole. For instance, in systems where tasks cannot be rolled back (our case), tasks must be granted access to shared persistent resources (updating data in particular) in some total ordering that must be unique system-wide.

In other words, concurrent accesses to shared resources can be serialized by relying on scheduling algorithms based on these protocols. This is a significant benefit, since avoiding uncontrolled conflicts enables real-time deadlines to be met despite failures. One essential assumption regarding failure modes is that nodes in the system fail by crashing, this being a conventional assumption in distributed dependable systems. Note however that the proposed solution may tackle more severe failure modes. Consequently, we assumed a very high coverage of this assumption [4], the means to achieve this being out of the scope of this paper (e.g. evaluation of failure modes as in [5,6]). As usual, it was assumed that no more than f crash failures could be experienced during the execution of a service – middleware-level service in our case.

Application problems are most conveniently tackled at the middleware layer. Such a layer provides appropriate generic services for the development of fault tolerant applications, independently from the underlying runtime support, namely COTS operating system kernel. The core components mentioned earlier constitute the basic layer of this middleware architecture, on top of which standard personalities (e.g. CORBA or better a microCORBA, Java) could be developed. The focus in A3M Phase 2 was the development and the validation (incl. real time characterisation) of this basic layer. The development standard personalities such as CORBA or POSIX are long term objectives not covered by the work reported here.

The paper is organized as follows. Section 2 sketches the A3M architectural framework. In Section 3, we describe the basic principles of the core components, namely UCS and UCN protocols and their variants developed in the project. In Section 4 we describe two key problems in space applications and their corresponding solutions based on these protocols. Section 5 focuses on the development strategy and justifies its interest from a validation viewpoint. Section 6 addresses the final platform used for the implementation and gives an overview of the A3M results in term of performance. Section 7 draws conclusions of this work.

2 Architectural framework

The architectural framework of the middleware comprises a basic layer implementing core components for the development of fault tolerant and real-time applications. This basic layer was designed and developed in A3M phase 2 on top of an *off-the-shelf* Real-Time Operating System kernel (RTOS), namely VxWorks [7], and a protocol stack specific to space platforms (COM). The design was such that no particular assumption was made regarding the selected candidates for RTOS or COM. The middleware itself has two facets. On top of the basic layer (facet 1) providing key algorithms, some additional services and personalities can be developed (facet 2) for targeting new problems and new application contexts. The work done until now was to develop the basic layer, i.e. the first facet of the A3M middleware. The overall architecture of the middleware is depicted in Fig. 1.

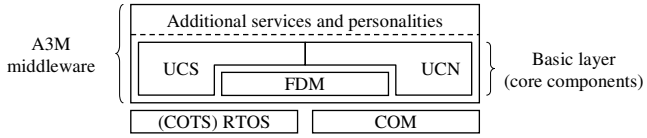

Fig. 1. Architectural framework

The basic A3M middleware layer comprises two major algorithms [1], Uniform Consensus (UCS) and Uniform Coordination (UCN), both relying on a Failure Detection Module (FDM). The A3M middleware layer is also populated with some specific services to tackle problems specific to space applications (see Sect. 4).

3 Basic components and protocols

3.1 Uniform ConsensuS (UCS): basic principles

UCS is the name of the protocol that solves the Uniform Consensus problem, in the presence of processor crashes and arbitrarily variable delays. UCS comprises two algorithms, one which consists of a single round of message broadcasting, accessible to application programs via a UCS primitive, another called MiniSeq, that runs at a low level (above the physical link protocol level). Both algorithms run in parallel.

UCS works as follows. Upon the occurrence of some event, every processor runs UCS by (1) invoking a best effort broadcasting (BEB) algorithm, which broadcasts a message containing its *Proposal* to every other processor, (2) invoking MiniSeq. A Proposal may be anything (the name of a processor to be shut down, a list of pending requests to be scheduled system-wide, the name of a processor chosen as the “leader”, etc.). Upon termination, UCS delivers a Decision. A Decision must be (1) unique and (2) one of the initial Proposals. Uniformity requires that even those processors that are about to crash cannot Decide differently (than correct processors) before crashing. Note that, in practice, Consensus without the Uniformity property is useless.

The MiniSeq election algorithm, which is sequential, is run by processors according to their relative orderings (based upon their names, from 1 to n, every processor hav-

ing known predecessors and known successors in set [1, n]). It is via MiniSeq that the Decision is made. MiniSeq uses a Failure Detector, denoted FD. Every processor is equipped with a Strong FD [10]. An FD periodically (every τ) broadcasts FD-messages meaning "I am alive". When processor p has not been heard by processor q "in due time", q puts p on its local list of "suspects". Analytical formulae (see [1]) which express an upper bound (denoted Γ) on FD-message delays are used for setting timers. An interesting feature (F) of Strong FDs is that it suffices to have just one correct processor never suspected by any other processor for solving the Uniform Consensus problem (even though lists of suspects are inconsistent).

Sequentially, every processor waits until some condition fires (namely, its Proposal has been sent out), and then, it broadcasts a Failure Manager message (FM-message), which is an FD-message that contains the name of the processor proposed as the "winner". This name is the name received from the nearest predecessor or its own name if all predecessors are "suspected". The winning value is the value heard last in set [1, n]. The condition for running MiniSeq locally is that every predecessor has been heard of (or is suspected). Thanks to (F), the "winner" is unique system-wide. The Proposal sent by the "winner" is the Decision. Note that we do not assume that broadcasts are reliable (a processor may crash while broadcasting).

The rationale for decoupling BEB and MiniSeq (parallel algorithms) simply is to minimize the execution time of UCS. Indeed, the upper bound B on middleware-level messages that carry Proposals (BEB) is very often much larger than bound Γ on FD-message delays. The execution time of MiniSeq is "masked" by the execution time of BEB whenever some analytical condition is met (see [1]). A simplified condition is as follows: $B > (f+1)\, d$, d given in Sect. 4. There is no a priori ordering of invocations of the UCS primitive (they may be truly concurrent). This above is referred to as *the regular UCS invocation mode*.

Another invocation mode of UCS, called *the Rooted UCS invocation mode* is available. R-UCS serves to make a particular kind of unique Decision, namely whether to abort or to commit a set of updates computed for data variables that may be distributed and/or replicated across a number of processors. R-UCS implements Atomic Commit. The major difference with the regular mode is that instead of having processor number 1 in charge of starting MiniSeq, it is the processor that actually runs the updating task – called the root processor – that is in charge of starting MiniSeq.

3.2 Uniform CoordinatioN (UCN): basic principles

The objective UCN is to reach distributed agreement on a collection of data items, each item (a Proposal) being sent by some or all processors. UCN makes use of UCS, starting with a round where every processor broadcasts a message containing a Contribution (to some calculus). At the end of that round, having collected all Contributions sent (up to f Contributions may be missing), every processor runs some computation (called "*filters*" – see below). The result of that computation is a Proposal. UCS is then run to select one unique result that becomes the Decision.

UCN may serve many purposes and can be customized on a case-by-case basis. The algorithm used to establish the final result is called a filter. Such *filter* determines the

final result from a collection of input values, namely the proposals. Examples of typical filtering algorithms are: majority voting on input values, aggregation of input values, logical and/or arithmetic expressions on input values, etc. This has many merits since it enables solving several problems such as distributed scheduling.

3.3 Real-time and asynchrony

Another innovative feature of the work presented is the use of algorithms designed in the pure asynchronous model of computation augmented with Strong (or Perfect) FDs [10] aimed at "hard" real-time systems. The apparent contradiction between asynchrony and timeliness is in fact unreal, and can be easily circumvented by resorting to the "design immersion" principle (see [2] for a detailed exposition). Very briefly, let us explain the general ideas. With "hard" real-time systems, worst-case schedulability analyses should be conducted. It is customary to do this considering algorithms *designed in some synchronous model*, running in a system S conformant to some synchronous model of computation (S-Mod). It is less customary to do this considering algorithms *designed in some asynchronous model,* running in S, i.e. "immersed" in S-Mod. Within S-Mod, every computation/communication step "inherits" some delay bound proper to S-Mod, this holding true for any kind of algorithm. Hence, the fact that asynchronous algorithms are timer-free does not make worst-case schedulability analyses more complex. One lesson learned has been that the analytical formulae/predictions regarding bounded response times for UCS and UCN were matched with the measurements performed on the A3M platform.

4 Case studies

A major objective of the A3M study was to assess the interest of UCS/UCN algorithms for space applications. For this purpose, the following two generic and recurrent problems have been selected:

- Problem 1: distributed consistent processing under active redundancy,
- Problem 2: distributed replicated data consistency, program serialization and program atomicity.

This section describes each problem and its solution based on the UCS and UCN algorithms.

Table 1. Hypothesis common to both problems

(F1)	Application level programs are denoted P_k. An instance of every such program is mapped and run, fully, on exactly 1 processor. Programs are assumed to be deterministic.
(F2)	Application level programs, or instances of such programs, can be invoked, activated and run at any time. This means that triggering events can be aperiodic, sporadic or periodic, that programs can be suspended and resumed later, and speed of execution of a program need not be know in advance (asynchronous computational model)
(F3)	There should be no restriction regarding the programming models. In other words, there is no restriction regarding which variable can be accessed by any given program, nor is there any restriction regarding which variables happen to be shared by which program,

(F4)	Neither application level programs nor the system's environment can tolerate the rolling back of a program execution. When a program has begun its execution, it should terminate – in the absence of failure – without replaying some portions of its code.
(F5)	A processor is unambiguously identified by a unique name; names are positive integers; the total ordering of integers induces a total ordering on the set of processors,
(F6)	The system includes n processors. Up to f processors can fail by stopping (crash, no side-effect) while executing some A3M middleware algorithm (UCS, UCN). This number obeys the following condition: $0<f<n$.

Every processor is equipped with a Failure Detection & Management (FDM) module, which consists of an FD module – which broadcasts FD-messages periodically, every τ – and a failure manager (FM), which runs MiniSeq. Locally, at every processor, an FDM module maintains a list of suspected/crashed processors.

"Good" clocks are such that their relative drift is null – or infinitesimal – over "short" time interval, equal to d, the worst-case latency for detecting the occurrence of a processor failure. We have $d = \tau + 2\Gamma - \gamma$, where Γ is a (tight) upper bound on FD-messages transit delays and γ a lower bound on such delays. It is important to note that the processors do not share a common clock (i.e. a common absolute time scale).

In addition, the following assumptions are made about the failures:

- Failures at software level: the software (application software and middleware) is supposed to be "perfect", or monitored by application level failure detectors that halt the processor in case of unrecoverable software error. The middleware is not in charge of application software internal failures detection and recovery. Nevertheless, in the particular case of problem 1, the proposed solution is able to mask a software failure that generates an incorrect result, provided that a majority of processors compute the correct result. Other software failures are not tolerated by the middleware (e.g. common mode failures and Byzantine failures).
- Failures at processor hardware level: the hardware is supposed to have only one failure mode, "crash" having the same effect as a processor halt. Like a halted processor, a crashed processor is no longer able to send any message to the other processors. This implies that the processors have built-in failure detection mechanisms, which is generally the case with space computers.

Failures at communication link or network level: the communication network is supposed to be reliable. This assumption can be "enforced" via classic retransmission-based protocols. For any message the maximum number of retransmissions (denoted R) needed for delivery is finite and bounded. A message can be lost (and recovered), but not corrupted. With the SpaceWire protocol, this assumption is valid concerning the demonstration platform. If the reliability of the SpaceWire error detection mechanism is not considered as acceptable for a real application, some higher level protocol can be added. If a processor halts or crashes during the transmission of a message, the receiver ignores the partially received message. If the error is only transient, the subsequent messages are transmitted correctly. Of course, analytical formulae for delay bounds such as B or Γ are established taking into account variable R (conventional worst-case schedulability analysis). Recall that we do not assume a reliable broadcast service for neither the BEB nor the MiniSeq algorithms.

4.1 Problem 1: Replicated processing

The computer system is made of *n* identical computers linked by a network or point-to-point links. The application software can be described as sets of programs, with causal dependencies, i.e. the output of programs are the input of other programs (i.e. a "chain" of application programs – see Fig. 2). External data acquisition comes first in the chain. Two cases are possible for implementing data acquisition (in both cases, the commands are sent by only one processor): only one processor performs the data acquisition or each processor performs the data acquisition in parallel with the others, via its own data acquisition interface.

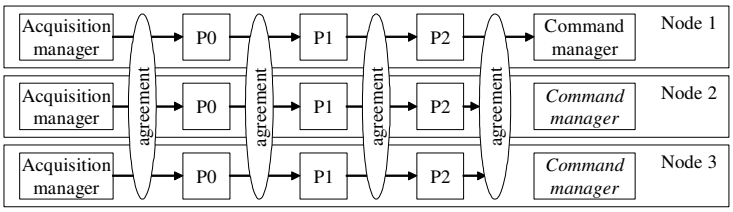

Fig. 2. Replicated processing framework

The same set of programs run in parallel on each computer, exactly in the same way: the output result of each program instance must be validated with respect to those produced by the other replicas, before delivering them as inputs to the next programs in the chain. The middleware allows each Pi replica to make an agreement on input or outputs values by calling dedicated functions. For example, in the case depicted in Fig. 2, P0 calls a function to validate the data produced by the acquisition manager. The measurement may be identical or different on the three processors, but in any case, all the P0 replicas will use as inputs exactly the same values.

The same mechanism is activated after execution of P0, in order to ensure that all the P1 replicas will run with the same inputs, and so on … The use of this agreement service is not limited to a single sequential chain of applications. For example, if P0 and P0' are running "in parallel" replicated on each processor, both can use the agreement service without having to be synchronized.

UCN with a "max filter" solves problem 1. This filter performs a majority selection on contributions. If no majority can be decided, the result is any of the contributions.

Application-level programmers can insert calls for agreement in their code wherever this seems appropriate. The consistency of these calls within application programs must be verified beforehand during the development process of the software.

4.2 Problem 2: Distributed access to shared objects

The second problem consists in replacing the existing "Data-Pool" mechanism used in the current Astrium mono-processor software by an equivalent compatible with the distribution of the processes on several processors – see Fig. 3.

The existing Data-Pool is a collection of data issued from software applications ("software data") or from equipments connected to avionics buses like MIL1553B ("hardware acquisitions"). In addition to inter process communication, the data pool is used by the periodic housekeeping data reporting applications, by the spy application,

and by the monitoring library. Its main purpose is to provide to data consumers consistent data generated from various producers at various frequencies. Concurrent processes running on different processors perform update operations in a pool of data items, called variables. These variables are shared by all processes and replicated for availability reasons. The objective here is to solve the mutual exclusion problem by means of consistent distributed scheduling among competing processes.

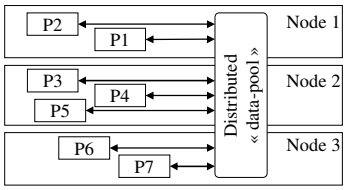

Fig. 3. Distributed access to shared objects

The application tasks use middleware services to ensure the consistency of operations performed on data pool variables sets. The programming model selected for these services is based on the notion of "critical section". The critical section of an application shall be considered as atomic by all the other applications with respect to the data-pool contents. There is no limitation on the number of read and write accesses performed on data-pool variables in a single critical section. Obviously, the duration of the critical sections should be kept as small as possible, in order to avoid a significant degradation of the overall system performance.

In order to reach this objective, we have introduced two additional services:

- The monitoring service receives from local and remote applications the requests for operations to the distributed data-pool. Using the UCN with a "total ordering filter", it ensures the system-wide serialization of the conflicting operations on shared variables.
- The guardian service executes the write operation on the replicated data pool, ensuring that any operation is executed on all the copies or on none of the copies. This property is ensured by a variant of UCS, called R-UCS, ensuring atomic program termination (Atomic Commit).

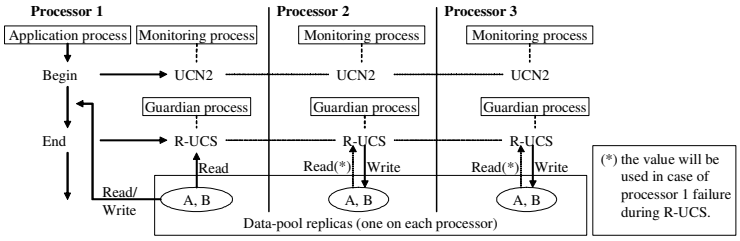

Fig. 4. Process entering a critical section

Fig. 4 illustrates the mechanism in the simple case where only one process wants to enter into a critical section. This process sends a request to the monitoring process that performs an UCN call to agree on which process must run first. Obviously, in this simple case, the requesting process wins, as there are no other competing processes. More generally, thanks to UCN, concurrent requests to enter a critical section are sorted out in the same order on all processors, i.e. all requesting processes will be

triggered by the monitoring process in the same order on every processor. The first in the list (according to any urgency criteria) is elected as the next process to be run.

It is worth noting however that, as processes are not replicated here (problem 2), only one processor P will host the active application process, say *p*. When the elected process *p* terminates its critical section on processor P, it signals the guardian process, which then calls R-UCS to perform a consistent update of the replicated copies of shared variables in the data-pool. However, as process *p* was running on processor P only, R-UCS must also be triggered on other processors different from P (those not running a copy of process *p*). This is done via the monitoring process, which directly signals the guardian process, which in turn activates a R-UCS call.

5 Development and validation

The main originality of the development was to fully develop (in C) the building blocks (BBs) and validate their behavior by simulation, including in the presence of faults. This approach was possible thanks to RTSim, a real-time executive simulator [8]. A distributed framework enabled us to execute complex scenarios, including the creation of an architecture, the setting of a network model, various ways of algorithms stimulation, and temporal and logic fault injection mechanism [9].

5.1 RTSim Overview

RTSim is a product developed by AXLOG. It aims at the development, in host-based environment, of real-time software based on various real-time executives of the market (pSOS+m, VXWORKS, Chorus, ARINC653, etc.), and their running under simulation without any target hardware. By creating nodes, which are the representation of a hardware module running a real-time executive, RTSim could manage simulated architecture from a tiny application running on a single-target system to complex multi-node architecture running a complex distributed application.

One of the main interests of this tool is that it is the real (final) code which is used in the simulation runs, which avoids the problem of abstraction mapping. Moreover, this tool offers advanced debugging, monitoring, controlling and instrumentation facilities over the application. In addition, RTSim has its own internal simulation clock, which enables us to replay a given simulation indefinitely with the same behavior, permitting a fast and reliable way to track defects.

5.2 Development and simulation environment

The first phase – the development of the BBs – followed a test-driven development approach (TDD), where each function described by the application programming interface (API) is unitary tested. The main advantages of this approach are:

- a use of the API in tests as in the future user code, giving possible refactoring enhancements before effective coding;
- a test database provides indicators on the current development velocity;
- non-regression tests could be passed easily in case of future implementation.

The second phase was the integrated testing of the BBs (to pass functional validation).

The architecture of the distributed simulation framework used for that purpose, shown in Fig. 5, consists of a multi-node architecture composed of a set of Processor under Test (PuT) representing the target hosting the module to be validated and a special node, the Processor of Test (PoT), responsible for the network simulation, scenarios execution and reporting. All these nodes are connected to a perfect link.

The PoT is in charge of stimulating and controlling the distributed applications, simulating the network, and archiving all actions taken by the system. The PuT hosts the algorithm module, a synthetic applicative workload, and a network service.

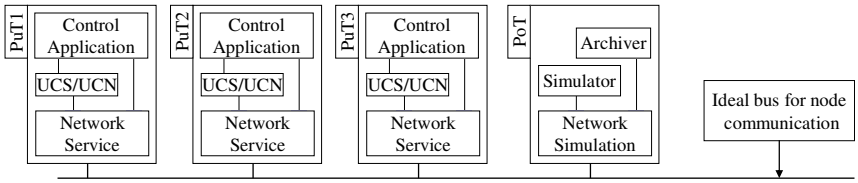

Fig. 5. Simulation framework

5.3 Test scenarios

After integration of the building blocks in the distributed simulation framework, test scenarios must be identified to cover the entire range of the application use and failure modes. Since UCS and UCN are asynchronous algorithms, time has no effect on their safety and liveliness properties, thus failure occurrence is the only parameter we have to handle (which reduces the complexity of integrated testing very significantly). The algorithms being described by state-machine diagrams, the only concern was to place the failure occurrences in these state diagrams, to derive the scenarios. An analysis of the state diagrams and the effects of a failure have led to simplifying the state diagrams, since the effect of a failure on some contiguous states was unique. The number of test scenarios is a solution to a combinatorial problem, i.e. the assignment of a number of failures varying from 0 to k (k = 2 in our case).

Crash faults have been injected according to this abstract behavioral model, in such a way that all possible states of the protocol are covered. The experiments showed that the implementation was correct according to the considered failure model.

6 Demonstration and evaluation

The demonstration and evaluation of the middleware has been performed on a platform representative of the next generation of on-board computer platform.

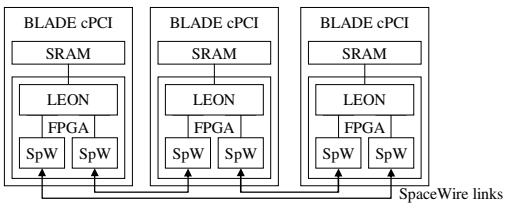

Fig. 6. The A3M Platform

This platform (fully operational at ESA/ESTEC) mainly consists in three double Europe Blade boards plugged in a CompactPCI cabinet. Each Blade board includes a FPGA programmed with a LEON processor and is connected to the two others by a SpaceWire link, and its two serial ports are connected to a RS232/Ethernet adapter. The SpaceWire communication software has been tested, and its performance measured. The raw performance of the hardware is pretty good (30 Mbits/s from memory to memory with a bit rate equal to 40 Mbit/s, which correspond to 99.5% of practical data rate), but the software mechanisms required at communication level by the middleware algorithms decrease the data transfer rate.

The test software consists in several tasks that can run either fully replicated on, or distributed across, the three processors. The objectives of the tests were to perform accurate performance measurements and check the behavior of the system in nominal conditions *and* in the presence of failures. To match these objectives, the duration of the different services of the middleware and operating system was measured with a great precision, i.e. less than 4μs. Computation errors and failures could be generated at predefined times.

To obtain a trustworthy characterization of the middleware, the tests have been executed on different platform configurations, with diverse parameters for FD algorithms (τ, γ and Γ) to be representative of existing spaceborne networks and with different size of exchanged data. The analysis of the measurements also had to take into account the resolution of the operating system timer, which has an negative impact on the worst-case latency for detecting the occurrence of a processor failure. These parameters are summarized in Table 1.

Table 2. Test configuration parameters (time expressed in milliseconds and size in bytes)

Test parameters	Set 1	Set 2	Set 3	Set 4
Processor number	1 to 3			
Variable size	40			272
Variable set size (2 variables)	80			544
Maximum number of tasks	16			
τ (FD messages period)	1000	100	50	1000
γ (min. transmission time of the FD message):	100	0	0	100
Γ (max. transmission time of the FD message)	1000	18	18	1000
ε (operating system timer resolution)	17			

The measures performed during the tests are compared with the theoretical maximum delay *d* for the detection of a failure on a processor given by the formula:

$$d = \tau + 2.\Gamma - \gamma + 2.\varepsilon \,. \quad (1)$$

Test of service 1: distributed consistent processing under active redundancy. This service is supported by a single function belonging to the middleware API, "*int dp1_check(dp_varset_id varset_id)*". Several processes running on the processors call the service to make a consensus on a set of values to use at the next step of a computation. The duration of the computation may vary from a processor to another to simulate a diversified programming.

Table 2. Measures performed during test executions of service 1

Nb of Proc.	Param. set	Service duration (ms)		
		Without failure		At failure
		Best	Worst	Worst
3	1	7.5	14.5	1300
2	1	5.5	7.8	1200
1	1	0.5	0.6	N/A
3	2	8	12	222
2	2	5.9	6.8	528
1	2	1.4	1.5	N/A
3	2	8	10	-
3	3	7.6	12.8	82.5
2	3	5.7	9.2	92.5
3	4	11	16	-

The results of the tests show the correct behaviour and performances of the middleware that are in accordance with the underlying theory. The WCET for the failure detection delay is never reached and errors are corrected without overhead. Moreover, the size of the exchanged data is not the main factor of service duration. By exchanging nearly 700% more data, the service duration only increases by 35%.

Test of service 2: distributed replicated data consistency. This service is supported by a couple of functions belonging to the middleware API, "*int dp2_begin(dp_task_id task_id, dp_varset_id varset_id)*" and "*int dp2_end(dp_task_id task_id)*" respectively used to enter and leave a critical section associated to a set of variables. Several processes running on the processors call the service to access shared variables. The duration of the reservation may vary. The next table shows some of the measures performed during test executions.

Table 3. Measures performed during test executions of service 2

Nb of Proc.	Param. Set	Operation	Service duration (ms)		
			Without failure		At failure
			Best	Worst	Worst
3	1	begin	10,5	12	1805
		end	6	7,2	1929
3	2	begin	10,5	11	130,5
		end	6,2	10	240
2	1	begin	9,5	10,5	1921
		end	5,8	6,8	1628
2	2	begin	9,25	11,1	786,5
		end	6,3	7,5	404
3	3	begin	9,5	11,5	93
		end	6,3	7,6	59,8
3	4	begin	10,6	11,8	786,5
		end	7,6	10	404

In all the tests, the service fulfils its specifications. The data consistency is ensured on all processors and the priority for accessing them is respected. The WCET for the failure detection delay is never reached. As for service 1, the duration of the procedures is weakly tight to the size of the exchanged variables.

In additional tests, not presented here, the two middleware services have run concurrently and the results obtained were totally consistent.

7 Conclusion and perspectives

The aim of the A3M project is to investigate and develop a new generation middleware for space applications. The distributed nature of the space platforms, the real-time requirements of the on-board applications together with stringent dependability constraints call for novel architectural frameworks and core services. The full picture also includes development processes and tools at various stages of the design and the

implementation of a system. As far as real-time constraints are concerned, a detailed specification of the application organization, the shared resources (variables, critical sections, etc.) and their timing behavior must be built first. Then, a detailed analysis aimed at establishing timeliness properties must be conducted a priori. This entails analyzing middleware-level algorithms, such as schedulers, and how they cope with concurrency, failures and timing constraints.

The work carried out in A3M Phase 2 led to the development of the basic sub-layer of the middleware. The core components rely on UCS and UCN-based modules devoted to solving conventional problems in space applications, ensuring fault-tolerance and real-time behavior in a distributed environment.

This work is the starting point for the development of middleware-based platforms for space applications. It demonstrates the benefits of advanced research results in this context and calls for further development of middleware-based systems for dependable modular avionics systems. It also shows the significant interest of simulation-based approaches for the development of basic middleware services and their validation, including in the presence of faults using fault injection techniques. This work should be continued, possibly with other industrial partners and targeting different application contexts. The results obtained are very promising and the development of additional middleware capabilities should be very attractive for the space industry at large.

References

1. J.-F. Hermant, G. Le Lann, Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems », *IEEE Transactions on Computer*s, vol. 51(8), August 2002, pp. 931-944
2. G. Le Lann, Asynchrony and Real-Time Dependable Computing, Proceedings of the 8th *IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS*), January 2003, Guadalajara, Mexico
3. G. Le Lann, U. Schmid, How to Maximize Computing Systems Coverage, *Technical Report 183/1-128,* Department of Automation, Technical University, Vienna (Austria), April 2003
4. D. Powell, Failure mode assumptions and assumption coverage, *22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston (USA), 8-10 Juillet 1992, pp.386-395. [Revised version in Predictably Dependable Computing Systems, Springer, ISBN 3-540-59334-9, 1995, pp.123-140]
5. J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, Assessment of COTS Microkernels by Fault Injection, *in Proc. 7th IFIP Conf. on Dependable Computing for Critical Applications (DCCA-7),* San Jose, January 1999, pp. 25-44 .
6. J. Arlat, J.C.Fabre, M. Rodriguez, F. Salles: Dependability of COTS Microkernel-based Systems, *IEEE Transactions on Computers*, Special Issue on Embedded Fault Tolerant Systems, Feb. 2002, pp.138-163.
7. VxWorks Realtime Kernel, WindRiver Systems, 1998.
   (see http://www.windriver.com/products/platforms/general_purpose/)
8. RTSIM real-time executives simulator, AxLog. (see http://www.axlog.fr/prod/rtsim.html)
9. M.-C. Hsueh, T. K. Tsai and R. K. Iyer: Fault Injection Techniques and Tools, *Computer*, vol. 30, no. 4, pp. 75-82, April 1997.
10. T. Chandra, S. Toueg, " Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the Association for Computing Machinery*, Vol. 43, N. 2, Mar. 1996, pp 225-267.