Supporting ARINC 653-based Dynamic Reconfiguration

Víctor López-Jaquero, Francisco Montero, Elena Navarro
Computing Systems Department
University of Castilla-La Mancha
Albacete, Spain
{ victor, fmontero, enavarro }@dsi.uclm.es

Antonio Esparcia, José Antonio Catalán
Eurocopter España
EADS Group
Albacete, Spain
{Antonio.Esparcia, Jose-Antonio.Catalan}@eurocopter.com

***Abstract*—** As the software for avionics becomes more complex, the challenge to provide the required reliability and safety mechanisms becomes also more complex. Embracing the ARINC 650 and 653 standards to provide a means to embark several systems into a single hardware cabinet opens the door to the development of even more elaborated software avionics systems, by overcoming the space constraints once found in this kind of system. Nevertheless, ARINC 653 exhibits also some limitations regarding fault redundancy management, especially when handling redundant applications with spares that back up other applications. In this paper, a framework to support fault tolerance and reconfiguration in avionics systems under the umbrella of ARINC 653 standard is described. This paper represents part of the results of the research projects carried out during the last two years by Eurocopter España in collaboration with the University of Castilla-La Mancha.

Keywords- Integrated Modular Avionic architecture, fault-tolerance, reconfiguration, redundancy, RTOS, APplication Executive, voting algorithm.

I. INTRODUCTION

The tendency towards more complex avionics systems in all kinds of aircraft is likely to continue for the predictable future, as digital electronics continue to become more powerful and cost-effective. As system complexity increases, the challenge of achieving the necessary reliability for safety critical systems becomes much more difficult, mainly because system failures will be more common as the number of components increases. The aerospace avionics domain is not exempt from this tendency, as the definition of guidelines, standards, etc for the specification of avionic systems demonstrates. ARINC 650 [2] is one of the most well-known of these standards and describes how avionic systems can be developed applying an *Integrated Modular Avionic* (IMA) architecture. IMA promotes a specific decomposition of avionic systems in order to deal properly with their inherent complexity.

The avionic systems have in the specification of fault tolerance mechanisms one of their most important and demanding tasks, because of the high constraints they have to satisfy. The exploitation of *system reconfiguration* results in a potential solution to improve the fault tolerance of these systems. Reconfiguration is the capability to adapt system functionally to the changing external or internal conditions of its environment [18]. By providing avionic systems with these facilities they can reconfigure at runtime when some components, either hardware or software, fail. They are well-suited for reconfiguration, because at their core they are naturally-partitioned and they include low-cost processing units that can be commanded to carry out the most appropriate function at runtime.

However, despite the inherent advantages of exploiting reconfiguration in avionic systems, the ARINC 653 standard [3], which belongs to the series ARINC 650 and provides guidelines for dealing with safety critical issues, does not offer any recommendations about how to incorporate reconfiguration mechanisms into IMA architectures. It is in this context where we present our proposal: an architecture for supporting reconfiguration in avionics systems and services based on the ARINC 653 standard. This proposal covers part of the most relevant outcomes of the industrial project carried out in collaboration with and supported by Eurocopter Spain, part of the EADS group.

The basic idea of the proposal presented is to use an intermediary layer, named *Redundancy and Reconfiguration Management Layer* (RRML), located between the *Real-Time Operating System* (RTOS) and the *APplication EXecutive* (APEX) layer. This intermediary layer is in charge of triggering and applying the reconfiguration mechanisms whenever a fault is detected. Therefore, by using this intermediary layer a reconfigurable method has been integrated into IMA.

This paper is organized as follows. Section II describes the application domain, that is, avionics architectures, reviewing their evolution from federated to IMA architectures. Section III identifies and introduces the related work. Section IV describes our proposal, how to add ARINC 653-based reconfiguration capabilities to avionics systems. Finally, Section V presents our conclusions.

II. APLICATION DOMAIN: AVIONICS ARCHITECTURES

One of the most important artifacts during the development of avionics systems is the specification of its architecture because it helps to manage properly the inherent complexity of this kind of system. One of the initial proposals for its specification was the exploitation of *Federated Architectures*. They are based on a set of distributed equipments, in which each one is mainly in charge of performing one function and its own physical interfaces. Therefore, *Federated Avionics Architectures* (FAA) make use of distributed avionics functions that are packaged as self-contained units [24]. One of the most interesting advantages this architecture provides is that each function is

978-0-7695-4827-2/12 $26.00 © 2012 IEEE
DOI 10.1109/WICSA-ECSA.212.9

isolated from the others, acting this separation as a natural fault propagation barrier. However, this architecture entails that a set of different computer systems have to be assembled in the aircraft which consume cost, weight and power. In order to face this drawback, during the last years, there has been an evolution in the aerospace avionics domain [8], from FAA to *Integrated Modular Avionics* (IMA). IMA [6] uses a high-integrity, partitioned environment that hosts multiple avionics functions of different criticalities on a shared computing platform. This provides evident advantages in terms of weight and power, since computing resources can be used more efficiently [14]. NH90 [11] or Airbus A380 [1] are examples of commercial aircrafts that use the IMA architecture.

Figure 1 depicts an example of the main elements that comprise the IMA architecture:

- *Partitions* are the smallest executable units in the system. Under the IMA architecture, each processor can host multiple partitions in which applications can be executed using the resources assigned. They are independent in execution and memory management according to ARINC 653 standard [3]. A partition is the sole owner of its resources, such as memory segments, I/O devices, and processor time slots. The applications running in different partitions cannot interfere with each other. To facilitate communications between applications, several channels can be assigned the partitions, but they are established at design time.
- *Module* is a component that contains at least processing resources and memory. A set of partitions are executed in a module. A module is a set (hardware and software) which provides one or more types of services, which can be used by one or more systems e.g. a processor module may provide processing capability for one or more systems. IMA module must be configured and loaded to reach the expected behaviour.
- *Cabinet* is a physical package containing one or more IMA components or modules that provides mechanical structure, partial protection from environmental effects (shielding) and cooling facilities.

The IMA architecture is described and documented in a series of standards called ARINC 650 [2]. ARINC standards, developed and adopted by the Engineering Standards for Avionics and Cabin Systems committee, deliver substantial benefits to airlines and aviation industry by promoting competition, providing inter-changeability, and reducing life-cycle costs for avionics and cabin systems. Among these standards, it is worth noting the *Avionics Application Standard Software Interface* (ARINC 653, [3]), usually applied by Eurocopter. This standard provides the guidelines for specifying the interface between the application software (partition in Figure 1) and the safety-critical avionics real-time operating system (RTOS in Figure 1) of an aircraft computer. Aside from aerospace and defense, where ARINC 653 separation standards are well-defined, most industries lack a unified approach for functional safety. In this sense, two major IMA goals to be highlighted are system reconfiguration, and incremental integration of new functionalities into a pre-existing system. This has been one of the main reasons that have led to the development of this work.

III. Related Work

As aforementioned, during the last years the space and aeronautic industries are paying special attention to IMA systems because of the advantages that modularity could bring to the development of avionics applications, especially in terms of interoperability, flexibility and software reusability. The ultimate objective of IMA is to produce a reconfigurable system.

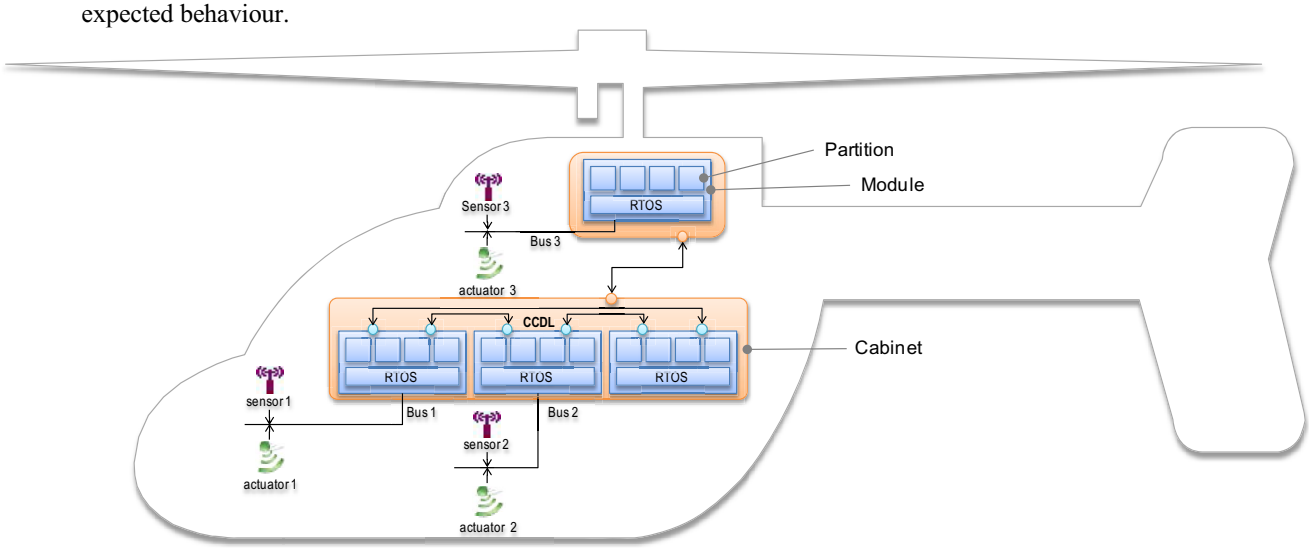

Figure 1. Example of Integrated Modular Avionics System

As was stated in section II, the ARINC 653 standard encourages the distribution of behavior (user functions) among partitions. Communications between partitions are set through the use of APplication EXecutive (APEX) channels which link statically a source partition with a target one. Though partitioning is expected to enhance reusability, portability and scalability, lots of interconnections tend to reduce all these capabilities. The main reason is that it would be less likely that a partition can work without the support of others (that is, the partitions would be strongly coupled), so that the system would act as it were monolithic. Moreover, *fault tolerance mechanisms,* such as reconfiguration, cannot be implemented since there is no way to avoid this strict structure at runtime.

Some related work has been done with the purpose of solving the weaknesses derived from the ARINC 653 standard, so that fault tolerance mechanisms, such as dynamic software reconfiguration, could be implemented. Among them, the *SCAlable & ReconfigurabLe Electronics plaTforms and Tools* project (SCARLETT, [20]) project tries to define, validate and demonstrate a new generation of on-board electronics platforms to answer to future aerospace challenges. The project is based on a shared analysis made by the European Aerospace Community, which has identified the need to undertake research leading to a new generation of IMA, defining a scalable, adaptable, reconfigurable fault-tolerant driven and secure avionics platform, namely *Distributed Modular Electronics* (DME). By implementing the innovations in the DME concept, SCARLETT will progress the state-of-the-art beyond the current IMA1G (IMA first generation) in the following areas:

- Scalability, portability and adaptability.
- Fault tolerance and reconfiguration capabilities.
- Minimize the number of types of standardized electronic modules.
- Support a full range of avionics function.

Another interesting alternative has been presented by Younis and He [24]. They describe an approach for the integration of redundancy management services into an IMA-based system. A proprietary scheme is used for time and space partitioning instead of the ARINC 653 standard with the APEX API. The authors use the MAFT architecture [15] and the X-33 vehicle management computers [7] as the starting point for the system architecture. The authors implement a software version of a *Reconfiguration Management System* (RMS) that runs on the same processor as the user applications. The software version of the RMS implements clock synchronization and voting of data mechanisms. However, these systems use a custom hardware board to implement the redundancy management system.

Black and Fletcher [5] discuss the usage of the IMA architecture with ARINC 653 for the avionics systems on spacecraft and booster vehicles. These authors propose using a master/shadow approach (or master/shadow/shadow) in each IMA cabinet. In other words, an N-Modular Redundancy (NMR, [21]) architecture with cross channel voting would not be required thanks to the use of a lock-step processors architecture. Details are not provided on how the shadow channels would take over as master. Also, it seems that the master and shadows would operate asynchronously, but details are not provided on how divergence between the channels could be prevented.

Lee et al. [16] discuss methods and tools for scheduling user application partitions and I/O processing within a single IMA cabinet that uses a shared backplane. The used IMA cabinet contains several modules and each module may contain one or more processors. All processors use an ARINC 653 compliant RTOS. The ARINC 653 *two-level scheduling method* is used on each processor, i.e. partitions are scheduled and then processes within a partition are scheduled. Authors present how user partitions can be scheduled along with the I/0 processing required between the modules within an IMA cabinet. However, no details about fault tolerance are provided in this work.

Another alternative is the Generic Avionics Scalable Computing Architecture (GASCA, [23]). It uses a system prototype with some functionality for fault tolerance. Each module consists of a VME (VersaModular Eurocard bus) chassis with several PowerPC boards. Every board within a VME chassis is also connected to all other processing boards using an SCI link. Although the API used by the applications is based on the ARINC 653, it has been modified and extended. Authors do not provide details of whether POSIX based RTOS supports true robust partitioning. Moreover, there is no mention of how synchronization and voting is performed.

Ferguson et al. [12] propose a common system software framework that uses the Real Time Publish/Subscribe protocol for framework-to-framework communication in order to extend ARINC 653. The authors show how such a framework would be well-suited for the Constellation program system-of-systems approach where vehicles can be reconfigured (e.g. by docking in orbit). However, fault tolerance mechanisms have been not explicitly addressed.

Conmy [10] presented a high level failure analysis of IMA. This analysis was based on six basic functions required by other systems using the IMA platform. This analysis revealed many derived requirements for the IMA platform including the need for further refinement of the health management system specification.

The main difference of the proposal presented in this paper, regarding the previous alternative works, is that we rely on an ARINC 653 compliant RTOS and the APEX API in order to implement a fault detection, isolation and recovery function. Thus, we support the integration of critical user application partitions on the same processor. Another difference illustrated in our work is that it also enables user applications, which require fault tolerant services, to use the APEX API defined by the ARINC 653 standard.

IV. AN INFRASTRUCTURE TO SUPPORT ARINC 653-BASED DYNAMIC RECONFIGURATION

One of the first decisions made for developing this infrastructure was to define the system architecture to be used. An *N-modular redundancy* architecture (NMR) was

selected because of its use of efficient real-time techniques to detect errors. The proposed IMA architecture consists of n redundant modules, one for each IMA platform, that operate simultaneously either in synch or not. For example, in Figure 1 there are three redundant modules interconnected by means of CCDL. The proposed architecture has the following features:

- The NMR architecture should have an odd number of redundant modules greater than 2, although it will work also using an even number of modules.
- The NMR architecture will work in synchronous mode, that is, all the redundant modules will share a common clock, since this is a requirement for most error detection techniques. Although real-time synchronizing algorithms can be more or less complex, it will save many efforts in other tasks, such as real-time error detection and reconfiguration strategies. The use of a synchronous mode makes also mandatory using a distributed real-time clock synchronizing algorithm.
- Since an NMR architecture is used, a dedicated *Cross Channel Data Link* (CCDL) to connect the replicated modules is required. Moreover, this CCDL is necessary to execute the distributed real-time clock synchronizing algorithm and to carry out error detection tasks (among other things). This CCDL can be developed by using either *dedicated point-to-point* connections or *bus* connections. The latter is the most widely used in avionics and, thus, the selected alternative for our architecture. In an avionics system, the data bus connects the different modules (or channels), sensors, actuators and other avionics subsystems to the N redundant modules. It is a completely independent bus from the CCDLs which links channels together. One important design decision is the topology of this data bus. The selection of the method used to connect flight control sensors and actuators to the redundant modules is a crucial decision. In this solution, a *channelized data bus* is used, where a set of sensors and actuators is only connected to one module using a single data bus (see Figure 1). This single data bus is usually referred to as a *string*. The advantage of this design is simplicity and isolation, since one string cannot interfere or damage another string. However, a disadvantage is that the loss of the computer module means that the entire string is lost, including all the sensors and actuators attached to that module. Therefore, critical sensors and actuators must be replicated in several strings.

Each single module of the used IMA architecture has its own internal architecture as Figure 2 illustrates. Both the hardware board and the hardware interface switching layers provide the underlying physical means to run the applications. These hardware layers are managed by a RTOS compliant with ARINC 653 standard. The communication between the applications running in the partitions and the RTOS are provided by the APplication EXecutive (APEX) layer, which decouples the applications of the specific RTOS. This API is also specified in the ARINC 653 standard. As observed, the different components proposed for redundancy and reconfiguration management have been grouped into the *Redundancy and Reconfiguration Management Layer*. These components are the following:

- *Error detection*. This component is in charge of feeding the redundancy and reconfiguration management processes. It is further described in section IV.A.
- *Communication mediator*. This component helps to manipulate the decoupling of the applications. It is presented in section IV.B.
- *Real Time Routing Table* (RTRT). This element has been defined to facilitate the implementation of the communication mediator. It is described in section IV.C.
- *Reconfigurator*. This component implements the reconfiguration actions. It is illustrated in section IV.D.
- *Clock synchronization.* Many algorithms applicable to this problem can be found in the literature, for instance the one proposed by Connell [9]. A variation of the synchronizing algorithm proposed in [22] can be used. The implementation of this kind of algorithms is a low-level task, closely coupled with the underlying hardware. Therefore, choosing the most appropriate synchronizing algorithm should be delayed until the hardware platform is chosen.

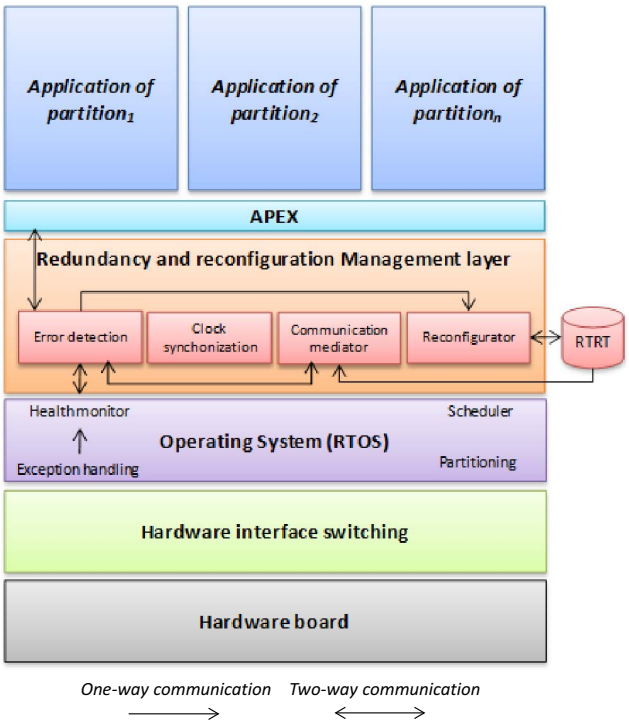

Figure 2. Single module architecture

In the following sections, these components are discussed in depth.

A. Error detection

This component triggers the redundancy and reconfiguration management processes when it detects errors in the data acquired from the sensors, in the data sent to the actuators or in the processes running in the partitions. The errors in these processes are detected by using the health monitoring facilities provided by the ARINC 653 RTOSes (see Figure 2), which is described in section IV.A.1). Both the errors in the incoming data from sensors and the outgoing data to actuators are detected by using a custom voting algorithm that is detailed in section IV.A.2).

The error detection component, as responsible for the management of the redundancy process, also has to handle the replicated modules. Therefore, it is mandatory that it implements a mechanism to detect when a module is no longer available and to mark both that module and its related partitions as unavailable. This mechanism can be carried out by means of two tactics:

- Ideally, the health monitor provided by the ARINC 653 compliant RTOS of the available modules would report to the *Redundancy and Reconfiguration Management Layer* (RRML) whether the module they are connected to failed or not. Then, the RRML would carry out the necessary actions to tackle this anomalous situation. For instance, depending on the number of available modules the voting algorithm detailed in section IV.A.2) will be adapted.
- *IsAlive protocol*. Although the previous one is the ideal way of reporting problems in a module, a critical error can arise, preventing the health monitor from reporting an error to the RRML. Therefore, a second mechanism to detect this kind of problems is supported: the *isAlive* protocol. The RRML of each module sends a message to the RRML of each other module it is connected to every *IsAliveTime* time units. If no reply is received from a module then that module is assumed to be unavailable.

For redundancy and reconfiguration management it is also necessary to detect when an error happens in a partition. Two mechanisms are used for this purpose:

- *Voting timeout*. Every time a partition requires/submits data, a voting process is run. The voting process for each module receives the value each other module proposes as the correct one. If any module does not submit its proposal for the voting in time, then the source/target partition (depending on whether it is an input or output operation, respectively) is considered to be down and therefore unavailable.
- Any critical error detected by the health monitor, either at the process level or partition level, will be reported to the RRML of its module, and the partition will be considered as unavailable, triggering reconfiguration actions.

1) Using health monitor facilities for error detection

The *health monitor* plays an important role for the redundancy and reconfiguration support presented in this work. It is in charge of capturing any exception in the processes running in a partition and reporting to the RRML with the error information that will drive the redundancy and reconfiguration process. Any exception, such as, insufficient memory, numeric error or unavailable ports, at the partition level will also supply the RRML with the information required to carry out reconfiguration actions.

As Figure 2 shows, and following the ARINC 653 guidelines, the implementation of the health monitor provided by the different RTOSes supports some extension mechanisms. The ARINC 653 standard attaches user defined handlers to the events in order to handle the different exceptions that can arise when running the applications in the partitions. The standard also allow for defining new events. The error information provided by the health monitor will be redirected to the RRML by attaching user defined handlers to health monitor notifications. By forwarding exceptions to RRML, the error control mechanism is centralized and fully integrated with redundancy and reconfiguration management.

2) Voting algorithm

The voting algorithm hereafter proposed is based on the classical voting algorithms found in the literature, such as [17]. Nevertheless, the voting process has been adapted to make it more flexible and powerful. The voting algorithm takes as input the values that each redundant partition from the different modules wants to acquire from a sensor or send to an actuator and carries out a voting process to guess what value is the correct one. This value will be then delivered to all the target ports for its processing. The voting algorithm is executed concurrently in all the modules.

In the classic voting algorithms, the number of redundant modules must be odd to avoid reaching a tie vote. Nevertheless, in the proposed algorithm this limitation has been overcome by introducing the concept of *partition trust*. A partition trust describes the level of trust the voting process has on a specific partition. At system startup, all the partitions have the same partition trust value and they are marked as *trusted*. When a value from a partition gets into the voting algorithm and it is wrong (it is different from the consensus value obtained after applying the voting algorithm), this partition trust value will be decreased. If a partition reaches a threshold (*trust threshold*) that partition will be left out of any further voting, and marked as *not trusted*. A partition marked as not trusted will be still monitored. Even if a partition is not trusted, its values will be still compared with the consensus value reached by using the values provided by the trusted partitions. If the values provided by the not trusted partitions match the consensus value then their partition trust value will be increased to reflect the fact that the partition seems to be working properly again. When the partition trust reaches the *back-to-life threshold*, the partition is marked again as trusted. A partition that is rebooted will be marked as trusted and its partition trust value will be set to the default value. A

partition that becomes *Unavailable* is automatically marked also as *not trusted*. If a module is rebooted, all its Unavailable partitions will be marked as trusted again and their trust value set to the default one. If the partition trust value of a not trusted partition drops below the *failure threshold* then that partition will be marked as *Failing* and will not be monitored anymore. If this Failing partition is not rebootable then it will reach the Unavailable state. This aforementioned behaviour is illustrated in Figure 3.

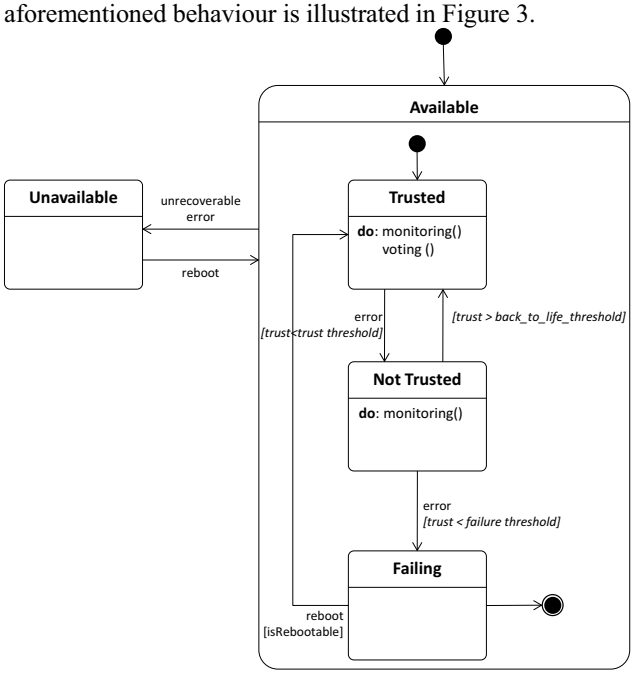

Figure 3. State diagram for partition object

Therefore, the actual voting process consists of a simple comparison to find out what value was voted by most of the involved partitions. Nevertheless, this voting is modified depending on whether the number of trusted partitions is odd or even:

- If the number of trusted partitions is odd (and greater than one) a regular voting process is made as aforementioned.
- Otherwise, if the number of partitions is even (and greater than two) the vote for each partition will be weighted according to its partition trust value, if a tie vote is reached. Thus, those partitions with a higher partition trust value will have a stronger position in the voting process. If even after the weighted vote the tie lasts, the value of the partition numbered with the lowest *id* will be chosen.
- Finally, if the number of trusted partitions is two, then a partition will play the master role and the other one the slave role. The master partition will be that with a higher partition trust value. If both partitions have the same trust value, the value for the partition numbered with the lowest id will be selected.

If the number of available partitions becomes one then no redundancy is possible, so that the remaining partition will be always treated as trusted and the values it provides to the voting process will always be considered correct.

B. Communication mediator

The ACR Specification [19] defines two important concepts widely used in IMA: *time and space partitioning* (TSP). TSP therefore ensures each partition uninterrupted access to common resources and non-interference during their assigned time periods. In this proposal, we coin a new concept: *communication partitioning*.

Space partitioning means that the memory of a partition is protected. No application can access memory out of the scope of its own partition. In this model, applications running in an IMA partition must not be able to deprive each other of shared application resources or those provided by the RTOS kernel. This is usually achieved by using different virtual memory contexts enforced by the processor's *memory management unit* (MMU). These contexts are referred to as *partitions* in ARINC 653. Each partition contains an application with its own heap for dynamic memory allocation and a stack for the application's processes (the ARINC 653 term for a context of execution).

Time partitioning means that only one application at a time has access to system resources, including the processor; therefore, only one application is executing at one point in time –there is no competition for system resources between partitioned applications. This ensures that one application cannot use the processor for longer than planned, avoiding the detriment of the other applications. ARINC 653 addresses this problem by defining an implementation that uses a *partition-based scheduling*. The ARINC scheduler allocates a time slot for each partition to be run depending on its specific needs. Within its time slot, a partition may use its own scheduling policy, but when it is over, the ARINC scheduler forces a context switch to the next partition in the schedule.

IMA design and implementation encourages the distribution of behaviour among partitions. Such distribution can result in an application structure with many connections between applications. In the worst scenario, every application ends up knowing about every each other. Although partitioning a system into many applications generally enhances reusability, proliferating interconnections tend to reduce it again. Lots of interconnections make it less likely that an application can work without the support of others –the system acts as it was monolithic. Moreover, often there are dependencies between the applications of the IMA partitions. For example, an application gets data from others applications in other partition. All these situations make difficult to change or reconfigure the system's behaviour in any significant way, since behaviour is distributed among many applications and partitions.

We can avoid these problems by encapsulating this collective behaviour in a separate component, namely *communication mediator* (see Figure 4). A mediator is responsible for controlling and coordinating the interactions of a group of applications, serving as an intermediary that

keeps applications in the group from referring to each other explicitly. Thus, the partitions only know the mediator, thereby reducing the number of interconnections. It is inspired on *Mediator design pattern* [13].

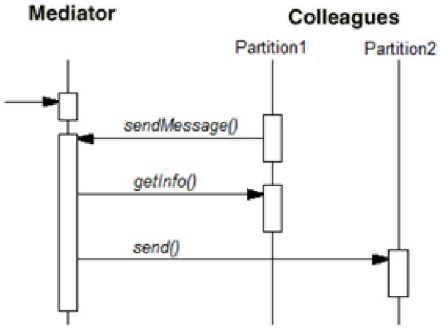

Figure 4. Communication partitioning by using a mediator element

In our proposal several participants are identified (see Figure 4):

- A *Mediator*: This element defines an interface for communicating with *Colleague* objects.
- *Colleagues* (Partitions): they send and receive requests from a mediator object. The mediator implements the cooperative behaviour by routing requests between the appropriate partitions. In our proposal: (i) each colleague (partition) knows its mediator object; (ii) each colleague communicates with its mediator whenever it has to communicate with another colleague.

The use of the mediator is recommended whenever any of these situations happen:

- A set of partitions communicate in well-defined, but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing a partition is difficult because it refers to and communicates with many other partitions
- A behaviour that is distributed between several partitions should be customizable.

The introduction of the communication mediator component provides our proposal with several advantages:

- The mediator encapsulates a behaviour that otherwise would be distributed among several partitions. Changing this behaviour requires only changing the mediator component, so that partitions (and its applications) can be directly reused.
- The mediator promotes a loose coupling between partitions. We can change and reuse both the partitions and the mediator independently.
- The mediator simplifies the communication protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain and extend.
- The mediator abstracts away how the partitions cooperate. A mediator promotes that partitions focus on their own behaviour, which can help to clarify how partitions work in a system.

Finally, a possible drawback of this proposal is that it promotes a centralized control. The mediator pattern also shifts from complexity of interaction to complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague making difficult its maintenance.

C. Real Time Routing Table

In order to implement the communication mediator described in section IV.B, a *Real-Time Routing Table* (RTRT) has been also included in the design of the Redundancy and Reconfiguration Layer to facilitate the decoupling of the communications (see Figure 2). A mirror RTRT is available in each module. Therefore, they must be kept updated by the system. To do so, a protocol has been implemented to report any change detected in the availability of the resources and their location.

Since the routing must be carried out in real time, the RTRT includes each source and target of every communication that an application does. The source and target columns contain a unique port identifier, composed of the module where the source and target partition are hosted, the name of the target and source partition and the name of the target and source port. Furthermore, the routing table also includes an up-to-date view of the partitions available in the system and their state (including the spare partitions). This information is required whenever a decision has to be made related to the rerouting of communications or the reallocation of partitions. Therefore, for each partition, the RTRT stores the module where it is and its state (see Figure 7). For the spare partitions, the RTRT also stores the identifier of the partitions they are spare of.

In the solution proposed, applications must only use pseudo-ports, otherwise applications would not be using redundancy and reconfiguration facilities. Pseudo-port are a special kind of port in the ARINC 653 standard that support the implementation of the communications by using a custom driver. Thus, we can intercept communications between partitions, sensors and actuators to forward them to the RRML components for the fault-tolerant mechanisms to be applied.

As only those data received from sensors or sent to actuators from the partitions are used for the voting process, the RTRT is used for the communication of the partitions with the sensors and actuators as described in the following. When a partition reads data from a sensor, the following steps are carried out to perform the communication (see Figure 5):

1. The sensor will be mapped to a RRML source port. Whenever data are received in this pseudo-port, they will be communicated to the error detection component. The error detection component will then use the incoming data from the RRML source port (and the data provided by the other redundant modules, received via

the CCDL) to carry out the voting algorithm described in section IV.A.2).

2. Once the voting has been carried out, the consensus value will be delivered to the communication mediator component (see step 2 in Figure 5).
3. Finally, the communication mediator will route the value to the target partition by using the paths specified in the RTRT.

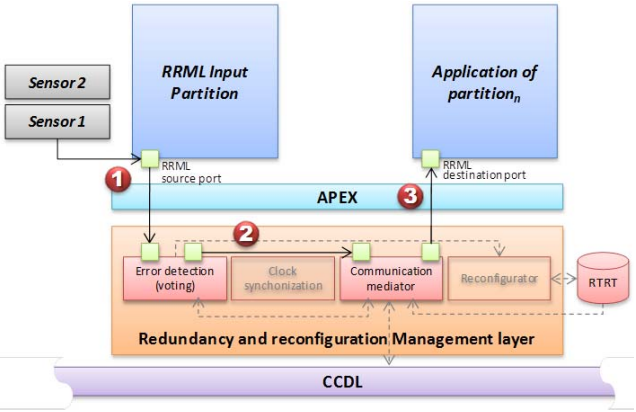

Figure 5. Input communication data flow

When a partition wants to send some information to an actuator the following process will be carried out:

1. A RRML source port will be used to write the data. These data will be used in conjunction with the data provided by the redundant modules to carry out the voting algorithm (see section IV.A.2)).
2. Once the voting has been carried out, the consensus value will be delivered to the communication mediator component (see step 2 in Figure 6).
3. The communication mediator will route the message to the appropriate target RRML output partition by using the paths specified in the RTRT, so it is delivered to the right actuator (see step 3 in Figure 6).

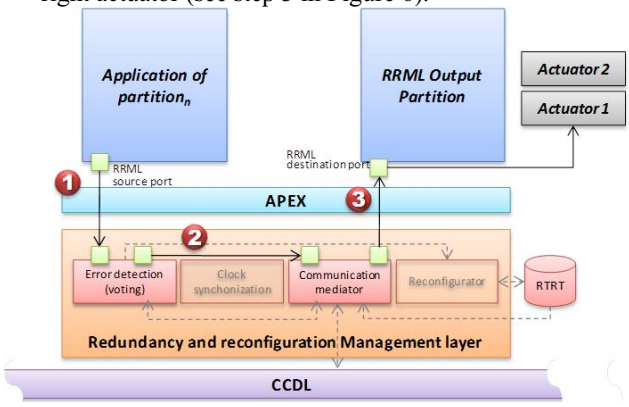

Figure 6. Output communication data flow

Whenever a partition wants to communicate with another partition no voting is required. Nevertheless, the data sent must arrive to the communication mediator to be rerouted to the right target partition, since a partition can be reallocated due to reconfiguration policies.

D. *Reconfigurator*

This component is in charge of taking the required actions to reconfigure the system whenever the error detection component, described in section I.A, reports an anomalous situation. Therefore, this component implements the *reconfiguration algorithms* that are the key element in any reconfiguration schema. Nevertheless, the effectiveness of these algorithms is purely dependent on the system information available at the moment of failure in real time. Making the right decisions in reconfiguration requires as much information from the current status of the partitions and modules as possible. Furthermore, this information should be always updated.

It is worth noting that not every partition can be reconfigured, since reallocating a partition requires resuming the state of the partition. Therefore, only stateless partitions have been considered in the proposed reallocation strategy.

Deciding what partition can be reallocated requires of run-time information that describes the features of the partitions, such as whether they are stateless or not. Moreover, the storage of these features must be extensible, since adding more reconfiguration strategies will surely require of extra features in the partition model. Bearing these ideas in mind, the partition model shown in Figure 7 has been described. As can be observed, a partition can be defined as *rebootable*, *active*, *spare*, and *available* (the trust value was already explained in section IV.A.2)). A rebootable partition can be reallocated, since it can be rebooted regardless of its state. An active partition will be a running partition. A spare partition will be created at startup, but it will not become active until a reconfiguration is required because of the detection of an unrecoverable error in the partition is it spare of. A partition is available when it is not in an unrecoverable error situation, such as when a module crashes. In this last scenario, none of its partitions will be available.

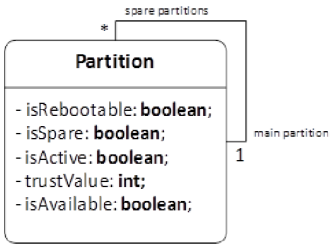

Figure 7. A model of partition

Moreover, in this work three kinds of partitions have been considered:

- *Non-critical partitions*. These partitions have no spare partitions and the system keeps running even if any of these partitions fail.
- *Critical partitions*. These partitions contain critical applications mirrored in different partitions in

different modules. A critical partition can have one or many spare partitions.

- *System partitions*. These partitions contain applications related to the RRML.

Taking into account these kinds of partitions, the following actions have been defined in the reconfiguration strategy supported by this work:

1. *Voting algorithm-based reconfiguration*. As described in section IV.A.2), the voting algorithm determines that the reconfiguration will be carried out according to the number of partitions involved in a voting and the trust values of those partitions. The voting algorithm-based reconfiguration is supported by the Error detection component (see Figure 2).
2. *Partition reallocation*. A partition will be reallocated whether any of the following situations happens:
   (i) A critical partition becomes unavailable (*partition.isAvailable=false*).
   (ii) A partition trust value drops under failure threshold.
   For a partition to be reallocated, an available spare partition must exist. Moreover, a rerouting of the communication (see section IV.C) must also be carried out. Non-critical and system partitions cannot be reallocated, since they do not have a spare partition. Therefore, their functionality will be lost in case of error.
3. *Reroute communications*. The rerouting of the communications is only applied when a partition reallocation happens. This reconfiguration action will update the RTRT to reflect the new location of the reallocated partition. This activity is supported by the reconfigurator component (see Figure 2).
4. *Reboot partition*. Rebooting a partition is considered under the following situations:
   (iii) If either an error is detected in a system partition or it becomes unavailable (*partition.isAvailable=false*), that partition will be rebooted, since system partitions have no spare;
   (iv) If either a non-critical partition becomes unavailable or it has an internal failure, and it is rebootable (*partition.isRebootable=true*), then this partition will be rebooted, since non-critical partitions have no spare. The only mean to detect a failure in a non-critical partition is by using the health monitor, since this kind of partition does not take part in the redundancy mechanism (voting process).
   (v) If either a critical partition becomes unavailable (*partition.isAvailable=false*) or if the partition trust drops under failure threshold, it will be reallocated in a spare partition in the same or a different module. But, if no spare is left, the only solution available will be to reboot the partition, if it is rebootable (*partition.isRebootable=true*). If the partition is reallocated in a different module then the sensors/actuators it was using must be available in the new host module (or made available via communication rerouting).

V. Conclusions and Further Work

This paper aims at covering two paramount topics in avionics: fault tolerance and reconfiguration. Fault tolerance can be achieved in many different ways, being the most usual those that include some kind of redundancy.

Since current avionics developments tend to embrace ARINC 653 standard, it is important to discuss how fault tolerance can be managed when considering this standard.

The framework presented in this paper contributes in two topics closely related, namely fault tolerance and redundancy. The devised communication mechanism supports the distribution of the redundant mirrors of the most important partitions, which include the applications, in the same module, or event in separate ones. It is achieved by supporting communications decoupling. This decoupling is achieved by providing an intermediary layer (RRML) that enables rerouting the communications to the currently active redundant mirror application.

An error detection mechanism based on the concept of trust value is also presented. This mechanism will detect faulty partitions and prevent them from delivering wrong data to other partitions. Moreover, this algorithm also helps in deciding what reconfiguration actions to take.

Furthermore, a custom voting algorithm is presented that takes into account the trust value for the partitions to make the decision of what is the correct value. This algorithm takes as input the values that every redundant mirror would like to send, and it guesses what value among them is the most trustworthy.

Lastly, the reconfiguration actions supported when an error is detected are described, thus providing the means to deal with faulty partitions.

Some tests have been conducted aimed at assessing the overhead in the communications caused by the fault-tolerance mechanisms proposed in this work. The results show that there is no significant overhead derived from the implementation of the solution designed. According to these results it seems like the scalability of the solution is not an issue. Nevertheless, additional testing is required to actually conclude that the solution is fully scalable.

To sum up, this paper presents a framework intended to improve the fault tolerance of avionics systems designed following the ARINC 653 standard, and to provide these systems with support for reconfiguration, that is usually constrained by the hardwired communications between partitions. It is worth noting that the solutions have been designed for reconfiguration in avionics systems, having this domain high hardware and software constraints. These constraints are imposed by the real-time operating system making it an especially challenging scenario, far from regular architecture. Many of the aspects shown here could be also applied to any distributed architecture with fault-tolerance requirements. For instance, the way partitions are handled and the voting algorithm could be extrapolated to web services.

ACKNOWLEDGMENT

This work has been mainly supported by Eurocopter España Corporation in Albacete in the context of the industrial projects UCTR100230 and UCTR090213 carried out during the last three years. This work has been also partially supported by the grant TIN2008-06596-C02-01 from the Spanish Government Department of Science and Innovation and also by the grant PEII09-0054-9581 from the Junta de Comunidades de Castilla-La Mancha.

REFERENCES

[1] Airbus, Airbus A380, http://www.airbus.com/aircraftfamilies/passengeraircraft/a380family/a380-800/, last access 08/03/2012.

[2] ARINC Specification 650, July 1995, Aeronautical Radio, Inc., https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=19, last access 08/03/2012.

[3] ARINC Specification 653 P1-2, December 2005, Aeronautical Radio, Inc., https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=1072, last access 08/03/2012.

[4] T. V. Batista, A. Joolia, G. Coulson, "Managing Dynamic Reconfiguration in Component-Based Systems", 2nd European Workshop on Software Architecture (EWSA 2005): 1-17.

[5] R. Black and M. Fletcher, "Next Generation Space Avionics: Layered System Implementation," IEEE Aerospace and Electronic Systems Magazine, vol. 20, no. 12, pp. 9 – 14, Dec. 2005.

[6] R. J. Bluff, "Integrated modular avionics: system modelling," Microprocessors and Microsystems vol. 23, pp. 435–448, 1999.

[7] L. P. Bolduc, "X-33 Redundancy Management System," IEEE Aerospace and Electronic Systems Magazine, vol. 16, no. 5, pp.23 – 28, May 2001.

[8] M., Di Natale, A.L., Sangiovanni-Vincentelli, "Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools", Proc. of the IEEE, vol.98, no.4, pp.603-620, April 2010.

[9] B. A. O'Connell, "Achieving fault tolerance via robust partitioning and N-modular redundancy", Master of Science in aeronautics and astronautics. Massachusetts Institute of Technology, 2007.

[10] P. Conmy, J. A. McDermid, "High Level Failure Analysis for Integrated Modular Avionics", SCS 2001: 13-22.

[11] Eurocopter, NH90, http://www.eurocopter.com/site/en/ref/Overview_177-884.html, last access 08/03/2012.

[12] R. C. Ferguson, B. L. Peterson, and H. C. Thompson, "System Software Framework for System of Systems Avionics," Proc. 24th Digital Avionics Systems Conference (DASC), IEEE, pp. 8A1/1 – 8A1/10, 2005.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[14] C. Hofmeister, J M. Purtilo, "Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement", 13th International Conference on Distributed Computing Systems (ICDCS) 1993: 101-110.

[15] R. M. Keichafer, et al., "The MAFT Architecture for Distributed Fault Tolerance," IEEE Transactions on Computers, vol. 37, no. 4, pp. 398 - 404, April 1988.

[16] Y.-H. Lee, D. Kim, M. Younis, J., "Scheduling Tool and Algorithm for Integrated Modular Avionics Systems," Proc. 19th Digital Avionics Systems Conference (DASC), IEEE, pp. 1C2/1 – 1C2/8, 2000.

[17] N. Littlestone and M. Warmuth, "Weighted Majority Algorithm". IEEE Symposium on Foundations of Computer Science, pp. 256-261, 1989.

[18] J. Moore, 2001, "Advanced Distributed Architectures," in The Avionics Handbook, C. Spitzer, ed., Boca Raton, FL, CRC Press, pp. 33-1 - 33-13.

[19] M. Nicholson, "Health monitoring for Reconfigurable Integrated Control Systems", Department of Computer Science, University of York. 2005

[20] RTCA, DO-255, "Requirements Specification for Avionics Computer Resource (ACR)." www.rtca.org, last access 06/02/2012.

[21] SCARLETT project, "SCAlable & ReconfigurabLe Electronics plaTforms and Tools", http://www.scarlettproject.eu/, last access 08/03/2012.

[22] M. L., Shooman, "Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design," Willey, 2002.

[23] T. K. Srikanth, S. Toueg, "Optimal clock synchronization", Journal of the ACM 34 (3), pp. 626-645. 1987, doi: 10.1145/28869.28876.

[24] J. C. N. Ventura and J. A. S. Neves, "Generic Avionics Scalable Computing Architecture," Proc. Data Systems in Aerospace (DASIA), 1999.

[25] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics", Proc. 26th Digital Avionics Syst. Conf., Oct. 2007.

[26] M. F. Younis and B. He, "Integrating Redundancy Management and Real-time Services for Ultra Reliable Control Systems," Honeywell International Inc., Columbia, Maryland. 2001.