Integration of Data Distribution Service and distributed partitioned systems

Marisol García-Valls†, Jorge Domínguez-Poblete†, Imad Eddine Touahria†, Chenyang Lu*
†Universidad Carlos III de Madrid
Av. de la universidad 30
28911 Leganés, Madrid, Spain
{mvalls,jdominguez}@it.uc3m.es
*Washington University in St. Louis
1 Brookings Dr.
Saint Louis, MO 63130, USA
lu@cse.wustl.edu

***Abstract*—Avionics systems are complex and time-critical systems that are progressively adopting more flexible (though equally robust) architectural designs. Although a number of current avionics systems follow federated architectures, the Integrated Modular Avionics (IMA) paradigm is becoming the dominant style in the more modern developments. The reason is that the IMA concept promotes modular designs where applications with different levels of criticality can execute in an isolated manner in the same hardware. This approach complies with the requirements of cost, safety, and weight of the avionics systems. FACE standard (Future Airborne Capability Environment) defines the architectural baseline for easing integration in avionics systems, including the communication functions across distributed components. As specified in FACE, middleware will be integrated into avionics systems to ease development of portable components that can interoperate effectively. This paper describes the usage of publish-subscribe middleware (precisely, DDS – Data Distribution Service for real-time systems) into a fully distributed partitioned system. We describe, from a practical point of view, the integration of the middleware communication overhead into the hierarchical scheduling (as compliant with ARINC 653) to allow the usage of middleware in the partitions. We explain the design of a realiable communication setting, exemplified on a distributed monitoring application in a partitioned environment. The obtained implementation results show that, given the stable communication overhead of the middleware, it can be integrated in the time windows of partitions.**

I. Introduction

Communication *middleware* and *virtualization* technologies are two main contributions to the development and maintainability of software systems as well as to machine consolidation [9]. These were initially used in mainstream applications, but are progressively entering into the critical environments and complex systems, where their role is increasingly important. In fact, in the avionics domain, the combination of IMA [24] and FACE [21] require the usage of both *virtualization technologies* to develop partitioned systems and *middleware* to ease interoperability and portability of components. This satisfies key requirement regarding cost, space, weight, power consumption, and temperature.

On the one hand, *middleware* brings in the capacity to abstract the low-level details of the networking protocols and the associated specifics of the physical platforms (e.g. endianness, frame structure, and packaging, among others). Consequently, the productivity of systems development is augmented by easing the programmability, maintanability, and debugging.

On the other hand, the penetration of *virtualization* technology has opened the door to the integration of heterogeneous functions over the same physical platform. This effect of virtualization technology has also arrived to the *real-time systems* area. The design of *mixed criticality systems* (MCS) [4] is an important trend that supports the execution of various applications and functions of different *criticality levels* [30] in the same physical machine. The term *criticality* refers to the levels of assurance over the system execution in what concerns failures. For example, in avionics systems, software design follows DO-178B [27] and DO-178C [26] that is a de facto standard for software safety; software is guided by DAL (Design Assurance Levels), and failure conditions are categorized against their consequences: from catastrophic (DAL A) to no effect (DAL E). Then, an MCS is one that has, at least, two functions of different criticalities on the same physical machines.

Over the past 30 years, middleware technology has been applied in critical domains but, mostly, in those subsystems of lower criticality levels. This is the case of, e.g., CORBA applied to combat systems [28] or, recently, DDS [19] applied to control of interoperability of unmanned aircraft and air traffic management[1], mainly for ground segment control. Still middleware is mostly used directly on bare machine deployments; yet it is not used in partitioned software systems.

This paper provides a design of a fully distributed partitioned deployment that integrates Data Distribution Service middleware. To ensure temporal isolation and communication timeliness, hierarchical scheduling is used as the execution framework. It is not the purpose of the paper to contribute new scheduling theory for this domain, but to apply it to the proposed novel distributed partioned design; this design

[1]http://www.atlantida-cenit.org

integrates a distribution middleware based on data distribution system to support timely communication among different functions located, both, within and across partitions. We exemplify this concept on a data monitoring application that has been developed to provide hands on the actual technology, to analyze the temporal behavior of the overall distributed partitioned setting.

This paper is an extended version of a previous contribution [8] that introduced the design of the fully partitioned deployment that made use of DDS middleware. The system was fully distributed across different physical machines that would perform data sampling and transmission that was later received, processed and displayed at a remote node. The nodes of the monitoring system emulate a mixed criticality system, with a setting that replicated that of a partitioned system in a FACE compliant architecture. The paper concentrated on the design of the software stack and the analysis of the middleware performance over an Ethernet network that emulates an AFDX (Avionics Full DupleX) [1] compliant communication. The present paper extends the previous one in a number of ways:

- A description of FACE standard is provided to better position the role of the middleware and identify the level at which it operates to support interoperability.
- We elaborate on the description of the hierarchical scheduling, providing a practical and simple model that leverages the elements defined in ARINC 653 scheduling for partition isolation.
- We provide a simple model to integrate the middleware communication overhead into the scheduling, considering application-level requirements with respect to the allowed safety margins.
- More results have been presented to provide the baseline performance of DDS. Moreover, the experimental section has been improved to illustrate the scheduling in the partitioned scenario. A more thorough presentation of the system parameters has been carried out with the goal of illustrating the conceptual part in the experiments.

The rest of the paper is structured as follows. Section II analyzes the most important characteristics and properties of communication middleware, and especially of DDS, for distributed partitioned systems within FACE. Section III presents the design of the distributed partitioned system, illustrated for a remote monitoring application. Section IV provides the implementation of the system and presents the results obtained for the communication. Section V presents the work background and selected contributions that are most related to our work, including concepts and technologies relative to partitioned systems and distribution middleware technologies for critical domains. Eventually, section VI draws some conclusions and describes the future work.

II. Middleware in FACE compliant distributed partitioned systems

A. Middleware within the FACE Standard

FACE [21] standard defines the software computing environment and interfaces designed to support the development of portable components across the general-purpose, safety, and security profiles required by the avionics domain. Its goal is to define the interaction points between the different technologies that ease systems integration. As a matter of fact, FACE makes use of industry standards for distributed communications, programming languages, graphics, and operating systems, among other areas. Version 2.0 further promoted application interoperability and portability, with enhanced requirements for exchanging data among FACE components; also, v2.0 emphasizes on defining common language requirements for the standard. Precisely, the levels defined by the *operating system segment* of FACE are the following, shown in figure 1:

- the *portable components segment*, that is composed of the applications, also named application components or FACE components;
- the *transport services segment*, where middleware technologies are employed for interoperability,
- the *platform specific services segment* for the common services of a given domain;
- *I/O services segment*, that is related to the low-level adapters and drivers to access peripherals including the network.

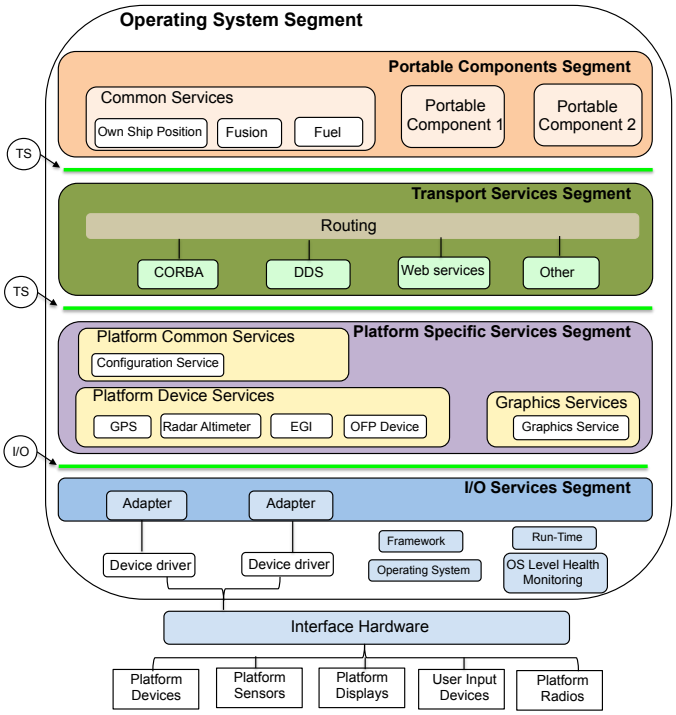

Fig. 1: Overview of the FACE architecture

At the transport services segment, many technological choices can be employed (e.g. CORBA, Web services, DDS, etc.). The selection of a specific middleware choice is important as it influences the properties of the system in what timeliness, efficiency, and (in the end) will provide compliance with different criticality levels. Despite the fact that still there are a number of active distributed systems in the avionics domain that use technologies such as CORBA, it is the case

that, at the moment, the most widely accepted one is DDS. It provides a well defined decoupled interaction model based on publish-subscribe. Also, it supports some degree of fine tuning of the communication relying on the actual capacity and behavior of the underlying network infrastructure.

B. Data Distribution Service for real-time systems

The Data Distribution Service (DDS) is an OMG standard that provides a publish-subscribe (P/S), i.e., a decoupled interaction model among remote components. DDS relies on the concept of a *global data space* where entities exchange messages based on their type and content. Such entities are *remote nodes* or *remote processes*, although it is also possible to communicate from within the same local machine.

Entities can take two roles; they can be *publishers* or *subscribers* of a given data type. Types are based on the concept of *topics* that are constructions that enable the actual data exchange. Topics are identified by a unique name, a data type and a set of QoS policies; also, they can use *keys* that enable the existance of different instances of a topic so that the receiving entities can differentiate the data source.

Applications organize the communicating entities into *domains*. Essentially, a domain defines an application range where communication among related entities (an application) can be established. A domain becomes alive when a *participant* is created. A participant is an entity that owns a set of resources such as memory and transport. If an application has different transport needs, then two participants can be created. A participant may contain the following child entities: *publishers*, *subscribers*, *data writers*, *data readers*, and *topics*. Publishers and subscribers are an abstraction that manages the communication part, whereas the data writers and data readers are the abstractions that inject the data.

One of the most successful elements of DDS is the set of quality of service parameters that it defines, namely *QoS policies*. In fact, not all of them are related to the temporal behavior of the communication. Most QoS policies provide other guarantees over the data transmission. A short summary of policies that influence the communication time (marked as t) and others affecting the overhead of the system (marked as o) are indicated in table I. The entities[2] to which they apply are also shown.

QoS policy	Entity	Description
Deadline (t)	DR	Max expected elapsed time between arriving data samples (unkeyed data) or instances (keyed data)
	DW	Max committed time to pulish samples or instances
Resource Limits (o)	DP	Limit to the allocated memory (message queues for history, etc.). It limits the queue size for History when the Reliability protocol is used.
History (o)	DR, DW	Stores sent or received data in cache. It affects Reliability and Durability (receive samples sent prior to joining) QoS policies
Latency Budget (t)	T, DR, DR	Indication on how to handle data that requires low latency. Provides a maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.
Timed based Filter (t)	DR	Limits the number of data samples sent for each instance per a give a time period
Transport priority (t)	DW	Establishes a given priority for the data sent by a writer. Underspecified: It is dependent on the actual transport characteristics and also supported by only some OSs.
Reliability (o)	DR, DW	Global policy that specifies whether or not data will be delivered reliably. It can be configured on a per DataWriter/-DataReader connection.

TABLE I: Subset of QoS policies affecting overhead

C. Middleware communication across partitions

The mainstream design of a critical partitioned system consists of isolating the communications of a system into one single partition. One of the partitions is selected as the *communications partition* (see figure 2) and is, then, responsible for the transmissions to and from other nodes. Partitioned systems are executed following simple hierarchical scheduling schemes. Each partition is assigned an execution time every given period, and the operating system is in charge of enforcing the assigned processor resource so that temporal isolation is fully guaranteed. As there is one single *communications partition*, transmissions take place exclusively during the time assigned to it. When an application or component residing in a given partition needs to transmit or receive data, it informs the communications partition via a shared memory space. When the kernel schedules the partition at its assigned time slot, the transmission takes place.

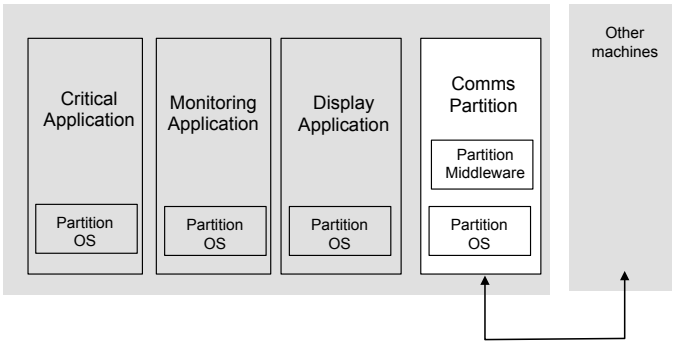

Fig. 2: Typical design of a mixed criticality system

This is the mainstream design approach in avionics, which has not relied on the usage of distribution middleware. Instead, a direct implementation of network protocols for ARINC 429 where used to support communication. Recently, standard-based middleware systems such as DDS are progressively considered into the actual implementations of critical partitioned environments; this has emerged with unusual vitality since the raising of FACE that searches for paths to easy system

[2]PU: Publisher, SU: Subscriber; DR: Data Reader, DW: Data Writer; T: Topic; DP: Domain Participant

integration. The reason is that the avionics industry has long realized the benefits of using matured commercial technologies as much as possible in order to improve component portability, interoperability, cost, integration time, and safe component reuse.

III. Designing a middleware enabled distributed partitioned system

A. System design

Figure 3 shows the major software blocks for the distributed monitoring system, focusing on the communication structure provided by the Data Distribution System which uses data topics. The system has two differentiated subsystems: *System A* that is a *meteo proximity server* (this system is connected to local –nearby– sensors that collect environmental readings) and *System B* that is the *Global Monitor* (this system receives the information from other nodes, including the meteo proximity server, and performs overall monitoring activities on the global scale).

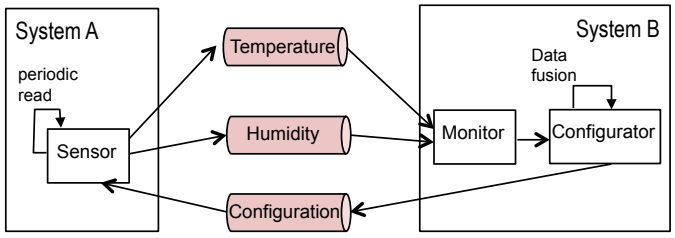

Fig. 3: Application software design

Two sensors (humidity and temperature) gather data samples that are passed to System A that is able to perform an initial basic processing of the data to take basic decisions on the sensors operation and detect faults. System A passes these data to System B that is able to perform more complex analysis over the data, and also it is able to make decisions on the configuration of the sensors and the overall system operation. The proximity sensor can do an initial processing of the samples and later, it sends the data to the global monitor.

Systems A and B run in two different physical machines that are partitioned emulating an ARINC 653 deployment. They are connected via an Ethernet link as compliant with AFDX. All partitions can integrate the communication middleware, so that any of them can send or receive data to and from other either local or remote partitions as shown in figure 4. This figure presents a complementary view with respect to figure 3 as here it is shown the partitions within each node and the software layers within each partition. The communication structure shown in figure 3 is encapsulated inside the *partition middleware* layer. In this way, communications may take place during the time slots that the kernel assigns to each of the partitions as will be presented in the following section III-B.

System A has two partitions: V_{A1} that monitors the sensor reads and performs basic analysis, and V_{A2} that performs data displays to the operators. System B has two partitions: V_{B1} that receives the sensor reads and performs complex analysis

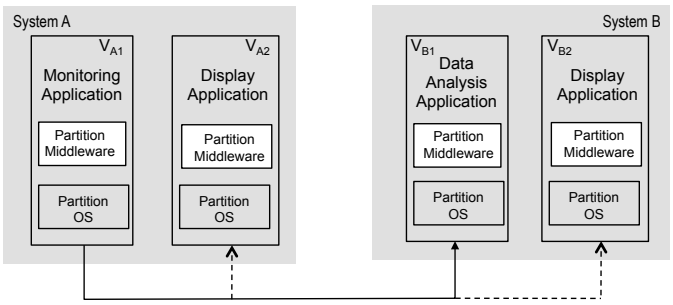

Fig. 4: Proposal architecture- Distributed monitoring application design in a partitioned system

to guide the system operation, and V_{B2} that performs data displays to the remote operators.

Partition V_{A1} receives the sampled data, and it sends them to V_{B1} through the appropriate topics: *temperature* and *humidity*, to send temperature and humidity values, respectively. Also, a *configuration* topic is used by System B to change the operation parameters of System A such as the sampling rate.

The topic based communication scheme between System A and Sytem B is shown in figure 3. Topics are associated to data types declared in a given programming language. The system defines a *dynamic topic* so that it is possible to update the its parameters (e.g. name or payload in this case). This favors flexibility and portability. This class contains a method for the creation and deletion of the topic. Code 1 shows the topic template.

Code 1: Topic structure template

```
struct DataSampleType{
 string<TOPIC_NAME> prefix;
 long sampleId;
 sequence<octet, TOPIC_MAX_PAYLOAD_SIZE>
     payload;
};
```

As the system handles sensor data samples, the needed topic parameters are *sequence number* and *payload*.

The communication settings for this application must be reliable. The subset of the QoS policies are shown:

- *RELIABILITY* is set to RELIABLE for guaranteed message delivery, and it is set both for data reader and data writer.
- *HISTORY* is set to KEEP_ALL that guarantees that samples will be retained until the subscriber retrives them. It is set for data reader.

Other properties as TRANSIENT need not be set as no late joiners are allowed. All connections are configured at the system start up time. Other parameters such as *Availability*, *Durability, Durability Service, DataWriterProtocol, DataReaderProtocol*, etc., ensure aspects as the ordered delivery of items, cache storage, usage of ACK/NACK and sending frequency. As our system is mapped to a UDP transport protocol over Fast Ethernet, it is sufficient to use the maximum allowed frame size as our application data messages payload

is of 1024KB. Also, IP routing is used as compliant with AFDX in a switched Ethernet setting; IP is enabled by the wire protocol of DDS that is RTPS (Real-Time Publish-Subscribe) [20].

RTPS is the protocol that enables interoperability across different implementations (vendors) of DDS. It was designed over the Internet Protocol (IP) multicast (one-to-many communication) and over an unreliable, connectionless transport such as UDP (User Datagram Protocol).

Apart from the default best-effort, RTPS also provides reliable communication for IP networks. RTPS offers distributed knowledge, preventing the centralised management of the system; this is an inherent fault tolerance mechanism which prevents the network from having a single point of failure.

The general implementation of the reliability mechanism for RTPS is based on the usage of acknowledgement messages (ACK). They are similar to a heartbeat mechanism and ACKNACK. Using a sequence number and a storage space (cache), it is posible to know which messages have arrived to the subscriber and which have not. Data writers have access to this cache. For each message, this cache stores the assigned sequence number, the history of the sent data values, and the history of whether the sample has been delivered to the reader.

B. Scheduling the partitions: hierarchical and ARINC 653 compliant

In IMA systems (that include FACE compliant systems), temporal partitioning is guaranteed though partition windows. The operating system kernel applies a deterministic scheduling algorithm based on a static schedule (defined in a static configuration file) that indicates the time windows that are assigned to each partition. This is defined in different operating systems that are ARINC 653 compliant such as VxWorks [22] or the MultiPartes approach [29].

The assignment of time windows to partitions (or virtual machines) defines a hierarchical scheduling model. Time windows enforce temporal isolation across the different partitions in the system. The timing overhead due to the middleware communication and transmission functions will be accounted for in the time window assigned to the partition(s) that execute(s) those middleware functions. In order to incorporate the transmission times to the schedule, the temporal bounds on the communication among the partitioned nodes have to be analyzed.

Here, it is presented a local schedulability framework with the intention of showing how the middleware overhead can be naturally integrated in the hierarchical scheduling model of partitioned systems. The middleware cost is considered and integrated within the partitions as follows. Let $\mathcal{V}$ be the set of n partitions of a system such that $\mathcal{V} = \{V_k\}$, $\forall k = 1..n$. The execution requirements of each partition are expressed as follows. A partition V_k requires to use the processor for C time units every period of T time units: $V_k = (C_k, T_k)$.

The execution life of a partitioned system based on ARINC 653 (see [22]) follows a hierarchical approach of two main levels. At the top level, the execution life of a system is defined by a loop over the same execution structure named *major frame*. The major frame is calculated as the hyperperiod of the periods of the partitions T_k, and let this hyperperiod value be MAF: MAF= hyp$(T_k), \forall k$.

The major frame is subdivided into a number (q) of time slots of equal duration called *minor frames*, and let their duration value be MIF. Each minor frame j is divided into a number (r_j) of time slots (S_k) of (possibly) different durations, each one assigned to a specific partition V_k. During its assigned time slot, a partition has uninterrupted access to common resources.

The actual time that the processor is active in a major and minor frames need not be MAF and MIF, respectively. Let F^M be the time that the processor is active during a major frame and let F_j^m be the time that the processor is active in minor frame j. It is given by equations (1) and (2), that summarize the simple scheduling scheme defined in ARINC 653 partitioned systems. Also, Figure 5 illustrates the scheduling in a graphical way.

$$F^M = \sum_{j=1}^{q} F_j^m \tag{1}$$

$$F_j^m \geq \sum_{i=1}^{r_j} S_{k,i} \quad \forall V_k \in \mathcal{V} \tag{2}$$

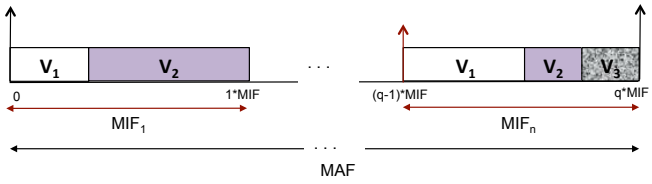

Fig. 5: Exemplification of hierarchical scheduling for ARINC-653 partitioned systems

The allocation of partition slots to each minor frame can be done in such a way that:

- The number of partitions executed in two different minor frames can vary. As a consequence, each minor frame structure can be unique.
- The specific partitions (partition IDs) executed in two different minor frames can be different.
- For efficiency reasons, a partition should be executed at most once in a given minor frame.
- Each partition must execute at least once during a major frame.

The calculation of the duration of the minor frames is straight forward as a divisor of MAF. The duration of the MIF has an impact on the number of context switches; in the avionics mainstream approach, the decision is application dependant.

C. Integrating the middleware overhead: the safety factor

Time windows ensure temporal isolation across partitions. This provides a natural baseline to integrate middleware communications in the partitions. In this case, the overhead caused by the middleware communication is integrated in the schedule by assigning additional resources for the communication functions as follows.

Let E^M be the spare time of one major frame, such that: E^M = MAF - F^M. There is a relation between the spare time and the maximum overhead of the communication as enabled by the middleware that is indicated in equation (3).

$$E^M \geq \rho \cdot c_{com} \tag{3}$$

where ρ is the *safety factor*, and c_{com} is the maximum overhead of the middleware communication. The value of ρ is an operator selection, so it is given based on application requirements. c_{com} is obtained off-line, and this can be done in different ways:

- real experiments; if actual tools are involved it is needed to perform real tests and experiments on the specific software layers that will be used. For this, it should be considered specific system conditions with values of interest for data payload and actual load.
- static code analysis and execution simulation; This is a problem of exponential complexity that does not adjust well to changes such as the uncertainty the network, varying payload, etc., This costly model lacks flexibility and does not take advantage of the partitioned scheme of ARINC 653.

In the current approach, real tests on a physical setting are undertaken. A middleware performance is analyzed off-line. The message size required by the specific application (1024KB in our case), the required load conditions, and the specific hardware. To be usable within a partitioned scheme for the above mentioned application, the specific tested implementation (DDS RTI Connext in our case) must yield:

- efficient values and stable behavior in different situations of load and payload transmission. The obtained middleware overhead values should fit within a feasible schedule, adjusting to the conditions indicated above, including the safety factor that is $E = 4 \times c_{com}$.
- the maximum obtained communication overhead c_{com} is associated to very exceptional situations. In this proposal, it is associated to an occurrence of a percentage (δ) of the execution results. In the present application, $\delta = 0.1$. As a consequence the system is overprovisioned for most of its execution, as required by the application domain.
- the partitions that make use of the middleware communication will integrate the maximum obtained overhead or cost of communication c_{com} within the partition time window value C_k.

IV. Results

Experimental results are presented with the goal of assessing the cost of using DDS in a partitioned context for supporting remote communication across local and remote partitions. We intend to validate the suitability of the proposed system design; we analyze the system in terms of the communication time from the side of the invoking node. Even for an unreliable setting, we have measured a partition-level reliable communication implementation, i.e., including the client response reception. The goal is to show that the chosen middleware provides communication bounds for this distributed partitioned system. Also, results show the stability of the middleware in the partitioned system, that is compared against the control group (a bare machine deployment in an unrealiable DDS configuration). Results of the execution of the implemented system show its feasibility, given the performance of the middleware in different scenarios over a sufficiently large number of trials to obtain the maximum values of the communication enabled by the middleware. The presented data is compiled from 1000 iterations to provide meaningful information. The communication results show the overhead of the whole processing stack of a partition [10].

The monitoring system is implemented with DDS RTI Connext 5.2.0 over two networked machines connected by a 100Mbps Ethernet link. The hardware of the physical nodes is a double core Intel E3400 at 2.6Ghz and with 2GB RAM. Results indicate the minimum (*min*), maximum (*max*), and average (*avg*) cases as well as the standard deviation (*std*). Results shown the real cost to show the efficiency of the scenario design and the overall performance.

A. Baseline performance experiments

The baseline performance is obtained by an intial experiment on a *favourable scenario*, acting as a control group against which to compare the reliable settings. Figure 6 shows the performance of the middleware on the favorable situation, i.e., a distributed setting with a *best effort* configuration set at the data writer and data reader entities of the distributed partitions.

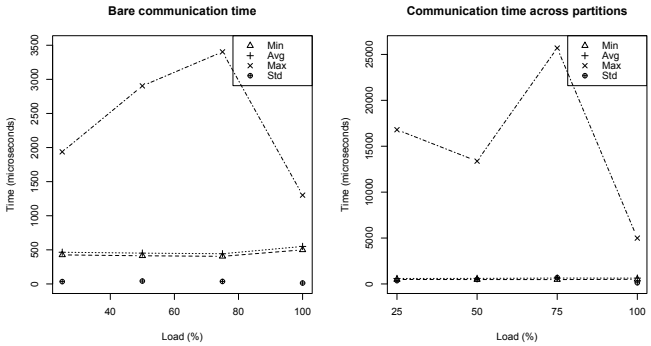

Fig. 6: Bare machine; best effort communication; varing load.

Figure 6(a) (at the left) shows a distributed setting for two remote bare machines, and Figure 6(b) (right side graph) shows a fully partitioned distributed setting. This scenario was analysed on, both, empty load conditions and with progressive

load up to 100%. The resulting data show that the communication cost was increased by 7x in the partitioned scenario. Nevertheless, it was shown that the average case times ranged from 3x to 7x lower than the worst case.

In this control group, it is shown the *maximum*, *minimum*, and *average* values, as well as the *dispersion*. The maximum value is associated to very exceptional situations, although possible. The average value is produced 99.89% of the time. Statistical dispersion provides confidence on the stability of the middleware behaviour. These measures must be studied for different situations of the system: load conditions (progressive load tests) and physical deployments (bare machine and partitioned). It should be noted the stable trend of the minimum and average values for both graphs (see two lower lines for each graph), as they are constantly around the same value regardless of the load conditions.

Figure 7 shows the results of a bare machine deployment of the emulated scenario for the distributed monitoring application. This scenario fully replicates the system and it also provides a DDS reliable communication configuration. The

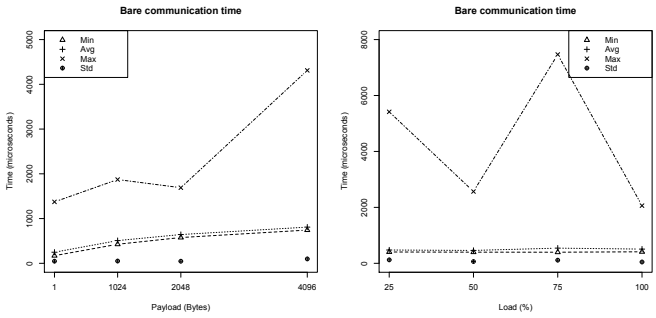

Fig. 7: Bare machine; reliable communication; varing load.

scenario of figure 7(a) (left side graph) is executed with no additional load, whereas 7(b) (at the right side) presents a progressive loaded system. Resulting times follow the expected pattern, and a similar phenomena as in the previous scenario is observed. Average behavior for (a) are between 3.7x and 5.33x smaller than the maximum times. Again, the communication proves to be very stable; dispersion is around $50\mu s$ for all the cases except for the largest message size that is $98.2\mu s$. For the scenario (b) that shows progressive load, the average times are between 11x and 4x smaller that the worst case. Nevertheless, the system shows to be stable, and the dispersion is between $47.2\mu s$ and $123.20\mu s$. As for the control group, it is worth noticing that trend of the minimum and average values for both graphs (see two lower lines for each graph) remains stable, i.e, constantly around the same value regardless of the load conditions.

B. Partitioned scenario

This section presents the experimental results on the partitioned scenario shown in tables II and III that correspond to system A of figure 4.

TABLE II: Requirements of the partitioned monitoring system

	V_{AP}	V_{DISP}
C	50	100
T	150	300
Communication frequency of V_{AP}	150	-

Periodically, partition V_{AP} sends a message on the sensor readings to a remote system (to system B of figure 4). The message has a size of $1024B$ and it is transmitted every $150ms$. The system requirements with respect to the middleware behavior are provided in table III. The safety factor imposes that the slack time E of the hierarchical schedule is, at least, 4 times larger than the maximum overhead of the communication transmission in its worst scenario.

TABLE III: System requirements set by operator calibration wrt middleware communication

Safety factor (ρ)	4
Occurrence probability (δ)	0.1
Payload size	1024B
Average load	50

A feasible hierarchical schedule derived from the requirements of the above partitions is shown in figure 8 that presents the initial schedule without considering the temporal overhead of the middleware enabled communication. The slack time, E, is $e_1 + e_2$, and the maximum overhead of the middleware must be allocated within the slack time. Figure 9 presents a feasible schedule that accounts for the time overhead of the middleware. In this case the slack time is reduced to $e_1' + e_2$.

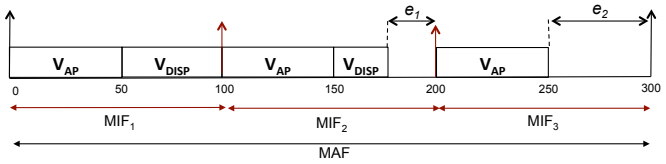

Fig. 8: Schedule without c_{mon}

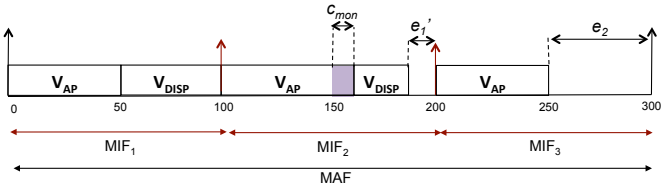

Fig. 9: Schedule with c_{mon}

The value of c_{mon} is obtained by the actual implementation and testing over different conditions of load and payload for the system, that do not only include the normal condition of the system. The results for the implementation of the fully partitioned sheme are presented in Figure 10. These experiments are performed to check different conditions. On the one

hand, the response of the system for various load situations is tested. On the other hand, the system is tested for different communication payload. Figure 10(a) (left side graph) shows the reliable communication setting on a distributed partitioned system deployment with no additional load. Figure 10(b) (right side graph) shows the same scenario with progressive load.

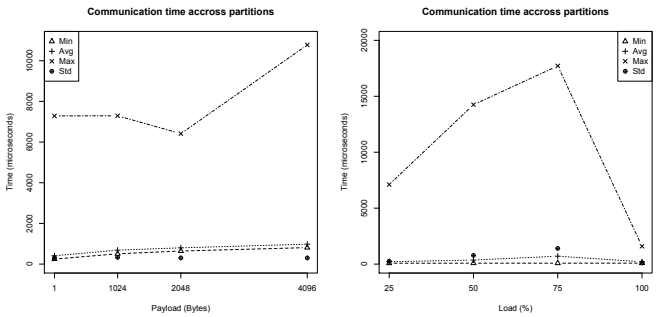

Fig. 10: Communication cost for a reliable configuration over partitioned systems

Results show that the system is very stable as standard deviation (std) is consistently around the same magnitude and in the order of $300\mu s$ for the empty scenario and between 212 and $1400\mu s$ for the progressive load. Moreover, the worst case is, for the empty scenario, between 8x and 18x larger than the average case. For the progressive load tests, the average case is between 8x and 39x smaller than the worst case. Also, it can be seen that the minimum and average cases are very close. Results show that the fully distributed partitioned environment yields a stable execution and the communication overhead of the middleware follows the expected pattern.

From the experiments, it is derived that the maximum overhead of the communication middleware, c_{mon}, is $18ms$. The spare time $E = 75ms$ and $\rho = 5$, therefore $E \geq 5 \cdot c_{com}$. So, $75 \geq 4 \cdot 18 \rightarrow 75 \geq 72$. values, as well as the dispersion. The maximum value is associated to very exceptional situations. The average value is produced 99.61% of the time, which yields that $\delta < 0.1$. Statistical dispersion provides confidence on the statibility of the middleware behaviour.

V. Background

IMA has been a very successful approach to transition from the former federated architectures to a more efficient design and final deployment into avionics systems. In this context, different standards are proposed to facilitate componentization, portability and interoperability at different levels of a system. In this way, ARINC 653 standard decouples the real-time operating system platform from the application software. For this purpose, it defines an APEX (APplication EXecutive) where each application software is called a *partition* having its own memory space. Each partition has dedicated time slots allocated through the APEX API, so that each partition can have multi-tasking and its own scheduling policy. Overall, the execution is embodied in a hierarchical scheduling policy where the top level is a cyclic schedule. Current work is to enhance ARINC 653 for multi-core processor architectures [7]. The underlying network is AFDX that uses commercial technology with redundancy to support safe transmission. The integration of networked and distributed systems follows ARINC 429 that is the data bus defining the characteristics of the data exchanges among the connected subsystems. It defines the physical and electrical interfaces and a data protocol for a local avionics network.

In other real-time systems, network scheduling typically relies on either an off-line and a-priori network schedule that defines the transmission plan for all the generated traffic (e.g. [5]); architectures such as TTA (Time Triggered Architecture) [17]; or distributed component models highly related to the hardware on-chip design such as Genesis [18].

Nevertheless, in the last decade the trend in the development of complex systems is to move to more productive ways of designing the communication and interaction. The avionics industry has developed FACE (Future Airborne Capability Environment) [21] standard to facilitate the development and easy integration of portable components. The communication middleware is given a key role as an interoperability enabler. Different technological choices can be used in FACE such as CORBA, Web services, or DDS, among others, as FACE does not bound itself to a specific technology. Each of these options are suited for different applications domains with varying criticality levels. The most popular technology at the moment is (probably) OMG's DDS standard [19] that has been applied in a number of domains such as remote systems control [11]. It provides an asynchronous interoperability via a publish-subscribe (P/S) paradigm that is data-centric, and the communication can be fine tuned through quality of service (QoS) policies.

Most recent works on the literature provide improvements to different aspects of the middleware such as service times making it aware of the underlying execution hardware [13]. On the performance side, there are some related works that contribute a thorough performance study of DDS for desktop virtualization technologies [10] but was not dealing with partioned systems; or the execution of DDS over a real-time hypervisor [23] although the actual network stack processing was not measured; or [6], [16] for network level P/S evaluation, and [25] for bare machine deployments. Overall, there is not sufficient analysis on the actual execution characteristics of specific middleware technologies in general partitioned environments. Moreover, there are no practical design models of partitioned systems that can comprehensively put forward the required software levels integration and there actual performance results. This paper contributes in this direction with a practical design of a distributed partitioned enviroment based on DDS, providing a study of the behavior of a partitioned system that communicates using this technology. Overall, it is remarkable that the values of the minimum and average cases fully adjust to a stable trend, and this is evidenced for both graphs (see the two lower lines for each graph). The new scenario shows that the average and minimum cases are kept constantly around the same value for the different load

conditions.

VI. Conclusion

The paper describes how communication middleware can be safely used in a time-critical environment, adhering to the application-level requirements. It provides a practical approach to the design of distributed partitioned systems that use DDS middleware to communicate remote partitions. Topics are defined to support the data centric model, that is exemplified for a distributed monitoring application. A simple scheduling model is provided that leverages the natural isolation given by ARINC 653. The communication overhead caused by the middleware and the partitioned setting is analyzed for a sufficient number of trials under different conditions that go beyond those exposed by the example application. Results show that the communication performance of middleware is stable even in presense of heavy load. The average case times are significanly smaller that the worst case (from 8x to 39x) and dispersion is 1.4ms for the worst possible scenario. The stability and communication overhead obtained is crucial in the approach to allow the integration into the partitioned systems. We exemplify the model for the case of the partitioned scenario, showing how it integrates naturally according to the proposed approach.

Acknowledgment

This work has been partly supported by the Spanish Ministry of Economy and Competitiveness through projects REM4VSS (TIN 2011-28339) and M2C2 (TIN2014-56158-C4-3-P).

References

[1] Airbus Deutschland GmbH AFDX. *Avionics Full Duplex Switched Ethernet.* `http://www.afdx.com` (Accessed 2016)

[2] – *ARINC Specification 653: Part 1, Avionics Application Software Standard Interface, Required Services*. `https://www.arinc.com/` (Access 2016)

[3] AEEC – Airlines Electronic Engineering Committee. *ARINC Specification 429, Part 1-15: Functional Description, Electrical Interface, Label Assignments and Word Formats*. `https://www.arinc.com/` (Access 2016)

[4] A. Burns, R. Davis. *Mixed criticality systems – A review*. 8^{th} edition. Report. Univesity of York. July 2016.

[5] R. Davis, A. Burns, R. J. Bril, J. J. Lukkien. *Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised*. Real-Time Systems, vol.35(3), pp. 239–272. April 2007.

[6] C. Esposito, D. Cotroneo, S. Russo. *On reliability in publish/subscribe services*. Computer Networks, vol. 57(5), pp. 1318–1343. April 2013.

[7] R. Fuchsen. *How to address certification for multi-core based IMA platforms: Current status and potential solutions*. Sysgo AG. http://www.sysgo.com (Online 2017)

[8] M. García-Valls, J. Domínguez-Poblete, I. Eddine-Touahria. *Using DDS middleware in distributed partitioned systems*. ACM Sigbed Review. 2017.

[9] M. García Valls, T. Cucinotta, C. Lu. *Challenges in real-time virtualization and predictable cloud computing*. Journal of Systems Architecture, vol.60(9), pp736–740. October 2014.

[10] M. García-Valls, P. Basanta-Val. *Analyzing point-to-point DDS communication over desktop virtualization software*. Computer Standards & Interfaces, vol. 49, pp. 11–21. January 2017. DOI: http://dx.doi.org/10.1016/j.csi.2016.06.007

[11] M. García-Valls, P. Basanta-Val. *Usage of DDS Data-Centric Middleware for Remote Monitoring and Control Laboratories.* IEEE Transactions on Industrial Informatics vol. 9(1), pp. 567–574. 2013.

[12] M. García-Valls, L. Fernández Villar, I. Rodríguez López. *iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time Systems*. IEEE Transactions on Industrial Informatics, vol. 9(1), pp. 228–236. February 2013.

[13] M. García-Valls, C. Calva-Urrego. *Improving service time with a multicore aware middleware*. 32^{nd} ACM/SIGAPP Symposium on Applied Computing (SAC). Marrakech, Morocco. April 2017.

[14] S. Groesbrink, S. Oberthr, D. Baldin. *Architecture for adaptive resource assignment to virtualized mixed-criticality real-time systems.* 4th Workshop on Adaptive and Reconfigurable Embedded Systems (APRES12), vol.10(1). ACM SIGBED Review, 2013.

[15] C. Gu, et al. *Partitioned mixed-criticality scheduling on multiprocessor platforms.* In *Proc.* of IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE), pp.1–6. 2014.

[16] A. Hakiri, P. Berthou, A. Gokhale, D. C. Schmidt, T. Gayraud. *Supporting end-to-end quality of service properties in OMG data distribution service publish/subscribe middleware over wide area networks*. Journal of Systems and Software, vol. 86(10), pp. 2574–2593. October 2013.

[17] H. Kopetz, G. Bauer. *The time-triggered architecture*. Proceedings of the IEEE, vol 91(1), pp. 112–126. 2003.

[18] R. Obermaisser et al. *Fundamental Design Principles for Embedded Systems: The Architectural Style of the Cross-Domain Architecture GENESYS*. IEEE ISORC 2009, pp. 3-11. 2009.

[19] Object Management Group – OMG. *A Data Distribution Service for Real-time Systems Version 1.4*. `http://www.omg.org/spec/DDS/1.4` 2015.

[20] Object Management Groug (OMG). *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, v2.2*. September 2014.

[21] The Open Group. *Future Airborne Capability Environment – FACE*. `http://www.opengroup.org/face` (Accessed 2016)

[22] P. Parkinson, L. Kinnan. *Safety-Critical Software Development for Integrated Modular Avionics.* VxWorks. White Paper.`www.windriver.com` (Accessed 2016)

[23] H. Pérez, J. J. Gutiérrez, S. Peiró, A. Crespo. *Distributed architecture for developing mixed-criticality systems in multi-core platforms*. The Journal of Systems and Software, vol.123, pp. 145–159. 2017.

[24] RTCA DO-297/ED-124. *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. `www.rtca.org` and `www.eurocae.org` (Accessed July 2016)

[25] T. Rizano, L. Abeni, L. Palopoli. *Experimental evaluation of the real-time performance of publish-subscribe middleware.* In Proc. of 2_{nd} International Workshop on Real-Time and Distributed Computing in Emerging Applications (REACTION 2013). Vancouver, Canada. December 2014.

[26] RTCA Inc. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA Inc. DO-178C. 12/13/2011.

[27] RTCA Inc. *DO-178B. Software Considerations in Airborne Systems and Equipment Certification*. RTCA Inc. DO-178B. 1992.

[28] R. Schantz, et al. *Towards adaptive and reflective middleware for network-centric combat systems*. Encyc. of Software Engineering. Wiley& Sons. 2002.

[29] S. Trujillo, A. Crespo, A. Alonso, J. Perez. *MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems.* Microprocessors and Microsystems, vol.38, pp.921–932. November 2014.

[30] S. Vestal. *Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance*. In *Proc.* of the IEEE Real-Time Systems Symposium (RTSS), pp. 239–243. December 2007.