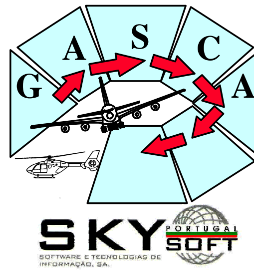

GASCA
"Generic Avionics Scaleable Computing Architecture"

João Carlos Negrão Ventura — *jcv@skysoft.pt*

José António Salvado Neves — *jasn@skysoft.pt*

Skysoft Portugal, S.A.
Taguspark, Núcleo Central, 337 ◆ 2780-920 Oeiras ◆ Portugal
Tel:+351-1-4228711 Fax:+351-1-4215161

Abstract: A GASCA system has two aspects, the architecture and the software. The GASCA architecture is composed of several modules running in several cabinets connected by the system, backplane and maintenance buses. A fundamental concept of GASCA is the partition, adopted from ARINC 653. The operating system software is the only software layer defined in GASCA, interacting with the application layer through the GASCA API. The project is still in the demonstration phase, and the OS, Configuration Manager, Fault Manager, Sample Applications and Distribution tool are presented.

The GASCA project is supported by the DGXII, and is being performed by Thompson-CSF Detexis (former Dassault Electronique) as project co-ordinator, Marconi Electronic Systems (former GEC-Marconi), Eurocopter Deutchland, Daimler-Chrysler Aerospace, DERA, NLR, Skysoft Portugal (former RTSN) and JAA as project partners. Alcatel and Lufthansa were involved in the beginning of the project but have since withdraw. The GASCA project co-ordinator is Mr. Jacques Draperi, with e-mail jacques.draperi@detexis.thomson-csf.com.

Introduction

Increasing commercial airline passenger traffic is leading to inefficiency in aircraft movement and congestion during flight and at airports. This has severe costs and safety implications. Customers, as well as operators must be assured of a correctly managed and controlled air traffic environment for continued growth.

The International Civil Aviation Organisation (ICAO) has recognised this, and in 1983 commenced a program to study, identify and assess new concepts and new technologies in the field of air navigation. It also made recommendations for the development of air navigation for international civil aviation over a period of 25 years. These recommendations led to the development of the Future Air Navigation System (FANS) concept which has embodied within it significantly improved Communications, Navigation and Surveillance (CNS) and Air Traffic Management (ATM) requirements. However, initial analysis of these requirements have indicated that current avionics architectures and equipment on-board aircraft will not be able to support the implementation of CNS/ATM functionality (and mainly those relative to Flight Management and Communication Management).

GASCA aims to define, to experiment and to validate through an advanced avionics architecture and system demonstrator that is capable of meeting these requirements. In line with airline operator needs, the architecture is capable of supporting a wide range of functionality including but not limited to CNS/ATM applications (such as Navigation, Flight Management, ATS Management, On-board Maintenance, Aircraft Condition Monitoring, Communication Management, …) and be sufficiently general purpose to allow concepts such as incremental upgrades, increased and selective functionality to be implemented in a cost-effective manner.

Architecture

A GASCA system is composed of cabinets that include different modules. Within a cabinet the modules communicate via a backplane bus, and the cabinets

---

*Proceedings of DASIA (Data Systems in Aerospace) '99 Conference, Lisbon, Portugal, 17-21 May 1999 (*ESA SP-447*)*

communicate between each other via the system bus. Furthermore, each cabinet is connected to a maintenance bus for maintenance actions.

Each cabinet contains a dedicated I/O module that manages the communication with the other cabinets.

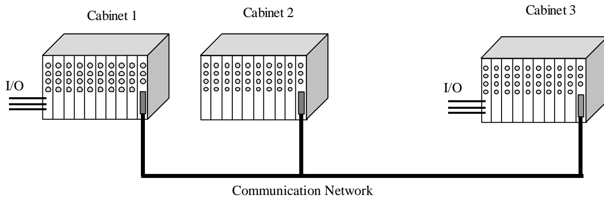

Figure 1: System Overview

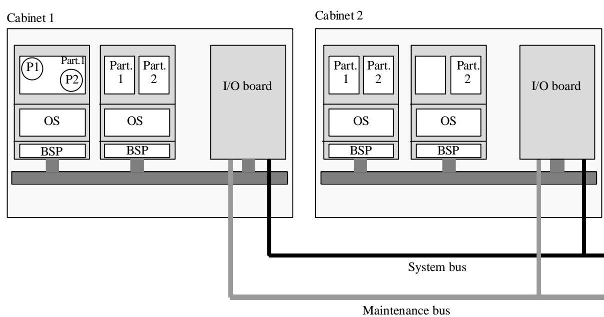

Figure 2: Cabinet Overview

A module contains one or more applications that run on the Operating System (OS) of the module. To ensure the integrity of each application, the OS uses partitioning principles (memory and time partitions) to schedule the applications.

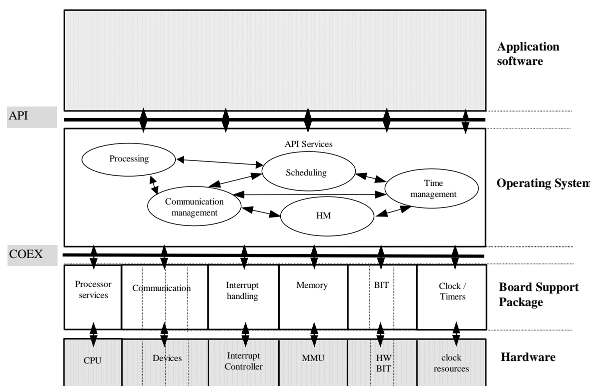

Figure 3: Module Overview

On a module, the software architecture is composed of three layers: the application layer, the OS layer and the Board Support Package (BSP) layer used to interface the OS to hardware.
The BSP layer is assumed to be provided by the equipment manufacturer and is not in the scope of GASCA, although it must support the COEX (core/executive) interface.

Partitioning

The GASCA concept of partitioning is derived from the ARINC 653 definition:

"One purpose of a core module in an Integrated Modular Avionics (IMA) system is to support one or more avionics applications and allow independent execution of those applications. This can be correctly achieved if the system provides partitioning, i.e., a functional separation of the avionics applications, usually for fault containment (to prevent any partitioned function from causing a failure in another partitioned function) and ease of verification, validation and certification.
The unit of partitioning is called a partition. A partition is basically the same as a program in a single application environment: it comprises data, its own context, configuration attributes, etc. For large applications, the concept of multiple partitions relating to a single application is recognised."

The partitioning model is considered extremely important to accomplish one major goal of GASCA: incremental certification.

Operating System

It is not the goal of GASCA to define a new operating system. However, in a GASCA system the OS must meet the following conditions:

- The GASCA API must be supported.
- The partitioning concept must be supported.
- For portability, the OS must be as hardware independent as possible.
- For modularity, the definition must be made by type of management.
- For openness, the OS must be able to admit extra software that makes an extended OS.
- The OS structure must be independent of the level of criticality of the module.

In a GASCA system, the OS must perform the following tasks:

- Partition management. This includes loading the partitions, construction of the cyclic major time frame, initialisation of each partition structure, scheduling the partitions and reporting any faults to the Health monitor.
- Process management. Within a partition, several processes can be running concurrently. A process is close to a «task» in Ada or to a «thread» in the POSIX terminology. The OS must schedule the processes and provide a set of synchronisation services.

- Time management. It is based on absolute time and includes time stamping, time control and deadline control.
- Memory management. Dynamic memory management services are to be provided for non-critical applications.
- Communication management. The communication model is specific to GASCA. The API services are generic and support every kind of communication (inter or intra partition communication). These services are message passing ones. The communication manager is an extension of the OS kernel, but extended communication managers can be added. The communication services include deadline considerations. Two types of messages are supported, queuing and sampling. In sampling mode, only the latest message is available for reading, in opposition to the queuing mode, where the messages are stored in a message queue.
- Health monitoring. The health monitor that is part of the OS is independent of the higher level Fault Tolerance Policy. Each module contains its own Health Monitor. There are three levels of error defined: module, partition and process level errors. For each level of error, a different recovery mechanism is proposed. For module and partition errors, the indicated recovery action is fetched from the appropriate table and performed. The error handler process within the partition manages process lever errors. Process errors can be propagated to partition level.

Fault-tolerance policies

In a distributed system, any number of faults can happen, either in the network or in the modules themselves.

The fact that these problems can occur forces any critical system to use some kind of fault-tolerance policy. Although no specific architecture was defined in GASCA, some guidelines are defined:

- Hardware fault monitoring should be a built-in feature. Hardware faults should be easily detected, and they should not lead to Byzantine failures.
- Critical functions should be replicated in order to minimise fault impacts. In the presence of faults, less critical functions can be degraded in order to maintain the critical function availability.
- Faults in shared resources should be handled by the shared resource itself, in order to make the fault transparent to the applications.
- API functions should be provided to access the fault tolerance policy mechanisms in order to provide the application developer with methods to handle fault classification, isolation and recovery.
- When a fault occurs, reconfiguration of the system, if needed, should be done in a manner transparent to the applications. This reconfiguration is initiated by a peer processing site that has all the information necessary to perform the reconfiguration. This implies the capability of storing in permanent storage the images of the application partitions and databases. At reconfiguration, the applications to be moved are then downloaded into the new modules and restarted in order to bring the entire system to a consistent state again.

GASCA API

The GASCA API is the interface between the GASCA OS and the applications. ARINC 653 was considered a baseline to build the GASCA API. The study made in GASCA shows that APEX is a good choice, nevertheless, the following must be pointed:

- ARINC 653 is in phase 1 and it only considers critical systems.
- In ARINC 653 phase 1, to address critical systems, no function is provided to manage memory. The memory allocation and release functions may be provided for non-critical systems. Some functions, like memory protection, are not included in GASCA because they are very dependent from hardware features.
- There are two ways to communicate in ARINC 653: Inter-partition and Intra-partition. It was chosen to unify both services.
- It was found that a function misses to build aperiodic tasks that want to wait for an event. The duration of the waiting is unknown and the task's deadline should be suspended. This notion is included in the two "wait to receive" message procedures.
- All the programming languages are potentially used in a GASCA system. Most of them have file services defined in the language. That is why no file services are added in the current GASCA API.

The API is split into the seven following packages.

Package GASCA_TYPES

No API calls are defined, only global types and constants.

Package GASCA_TIME

- timed_wait: is used by a process that wants to be suspended for a certain amount of time.
- periodic_wait: is used by a periodic process that wants to be suspended until its next activation.
- get_time: is used to get the current time.
- replenish: is used by a process that wants to replenish its deadline.

Package GASCA_PARTITION

- get_partition_status: is used to obtain the status of the current partition.

- set_partition_mode: is used to put the current partition into the normal operating mode, or by the error handler to stop or restart the partition.

Package GASCA_PROCESS

- get_process_id: is used by a process to obtain the process identifier related to a specified process name.
- get_process_status: is used by a process to obtain the current status of the specified process.
- create_process: creates a process and returns an identifier that denotes the created process.
- destroy_process: destroys a process previously created.
- set_priority: changes a process's current priority.
- suspend_self: suspends the current process if aperiodic, until the resume service is issued or the specified time-out value expires.
- suspend: is used by a process to suspend another process specified by its identifier, provided that this process is not periodic.
- resume: is used by a process to resume a previously suspended process specified by its identifier.
- stop_self: is used by a process that wants to stop itself.
- stop: is used by a process that wants to stop another process.
- start: is used by a process to start another process.
- lock_preemption: is used by a process that does not want to be preempted during the instructions executed after the call of this service.
- unlock_preemption: ends the current critical section of the process.

Package GASCA_MEMORY_MANAGEMENT

- memory_allocation: is used by the application to allocate a specified amount contiguous block of memory.
- memory_deallocation: is used to free a memory block previously allocated.

Package GASCA_COMMUNICATION

- create_message_port: creates a port related to a message.
- destroy_message_port: destroy the link between the port and its related messages and suppress the availability of use of the port.
- send_message: is used to send a message related to a port.
- wait_to_receive: is used by a process that wants to receive at least one message from a specified set of messages.
- wait_to_receive_all: is used by a process that wants to receive all the messages in a specified set of messages.
- read_message: is used to actually receive a message.
- clear_message: is used to clear all the messages received on a specified port.
- init_message_set: initialises a message set to be used in the wait_to_receive calls.
- put_message_in_set: puts a specified port into the message set.
- delete_message_from_set: deletes a port previously added in the set.
- get_message_from_set: returns one of the ports contained in a specified message set, and deletes this port from the message set.
- check_message_in_set: checks if a specified port is present in a message set.
- message_set_is_empty: indicates if a specified message set is empty or not.

Package GASCA_ERROR

- report_application_message: is used by the processes of the current partition to transmit a message to the Health Monitor for a recording purpose.
- create_error_handler: is used by a partition program at the start-up of the partition, to create an error handler process for this partition.
- destroy_error_handler: is used by the Health Monitor to remove an error handler process.
- get_error_status: is used by the error handler process to determine the error information from a faulty process.
- raise_application_error: is used by a process to signal a detected error and therefore invoke the error handler.

Current Work

The project is now reaching its final phase, the demonstrator has been defined and is almost ready for final integration.

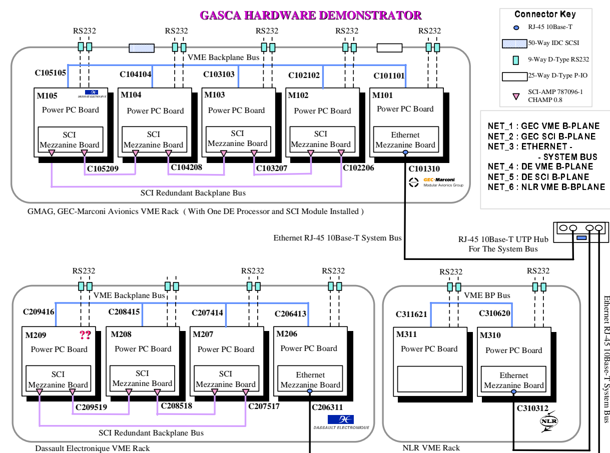

Figure 4: Demonstrator hardware

The demonstrator architecture is composed of several PowerPC Boards connected by a VME backplane bus and an SCI redundant backplane bus, in each cabinet. Each cabinet is connected to the others using an Ethernet system bus. No maintenance bus was considered necessary for the demonstrator.

The selected operating system for the demonstrator is a light POSIX kernel developed by Detexis that supports multitasking, threading, real-time, inter-process communication through shared memory and message queues and I/O. The standard C library is also provided.

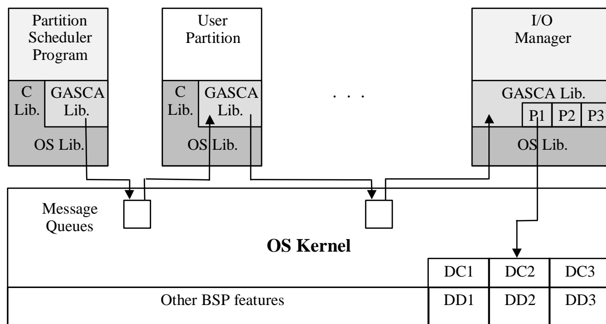

Figure 5: System software components

The GASCA API is implemented as a library on top of this kernel. Some changes were required to the kernel, in order to comply with the GASCA requirements, mostly related to health monitoring, but also dealing with the specifics of the GASCA process model and communication model. Currently, the GASCA API is also supported on top of Sun's Solaris operating system, with lesser functionality, but suitable for development, testing and debugging.

Some system applications will be running in all the modules:

- PSP: Partition Scheduler Program, launches and schedules the other partitions. It is also responsible for calculating the major cyclic frame, at system start-up.
- LCM: Local Configuration Manager, part of the fault-tolerance system.
- LFM: Local Fault Manager, also a part of the fault-tolerance system.

The I/O manager application will run in a separate module one in each cabinet, and is the only application that can communicate with the external modules. It serves as a gateway between the applications running in its cabinet and the applications running on other cabinets.

Fault-Tolerance

The fault tolerance architecture for the GASCA demonstrator has been developed by Marconi Electronic Systems and is composed of the Global and Local Configuration Managers, Global and Local Fault Managers, partition Error Handlers and a replication layer for critical applications.

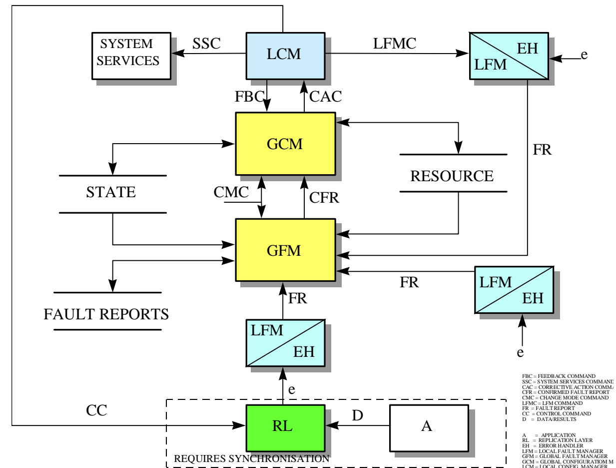

Figure 6: Overall Fault-Tolerance Architecture

The architecture depicts the FT services at local module interfacing with the system level FT services. The module level services consists of the Error Handler (EH), Local Fault Manager (LFM) and the Local Configuration Manager (LCM), which are responsible for handling the error and reporting the error, handling local services and providing common interface (to global FT elements) respectively.

The Global Fault Manager's (GFM) main task is to isolate the source of fault from all the individual fault reports in the system. It implements an algorithm which generates a list of candidates components which could all be a potential source of failure, and then using fault analysis by inferences, eliminates them until an unambiguous decision is found. On convergence to an unambiguous decision, a Confirmed Fault Report (CFR) is generated and reported to the Global Configuration Manager (GCM).

The GCM will keep track of the current system state and will update its state on a CFR or a change mode command (CMC). The GCM will co-ordinate the system reconfiguration based on current state and either the CMC or CFR messages. The reconfiguration at local level will take place through LFMs via LCM.

To cater for critical applications, the FT architecture will support N-Modular Redundancy (NMR) through services embodied in the Replication Layer (RL). The replication layer will support Single-Version Programming (SVP) which can be replicated N-times to form N separate channels running concurrently (often referred to as hot channels). The RL layer will ensure

that a single output is always maintained from N-hot channels.

The GFM and the GCM will run in one of the demonstrator modules. As this is a demonstrator system, there is no need to replicate the GFM and GCM. In a production system, a more refined architecture without a single point of failure would have to be addressed.

Demonstrator applications

Some simple applications were developed using the GASCA API for the demonstrator. These applications will demonstrate the correct functioning of the GASCA system. The applications are:

- Auto-Pilot: taking as input the next waypoint and the location of the system aircraft, it will issue commands to put it in the correct heading.
- Avoidance System: receiving location and heading information about the aircraft in the region around the system aircraft, it determines and informs about any possible collision in the near future.
- Flight Control: receives inputs either from the control stick and throttle or from the auto-pilot, it converts them into actuator commands.
- Flight Management System: calculates the aircraft location from beacon and sensor information and sends the next waypoint information to the auto-pilot.
- Utilities Management: calculates and informs about several parameters, such as fuel, engine, temperature and landing gears state.

Other dummy applications were also developed in order to consume some more system resources.

The following figure illustrates the connections between these applications and the environment. The applications were developed by Skysoft, and the environment simulator by NLR.

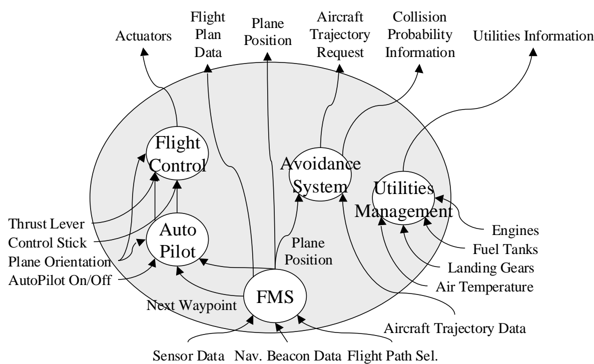

Figure 7: Demonstrator Applications Overview

Configuration and Distribution Tool

An important part of the GASCA demonstrator system are the application blueprints, which define the partitions to be run in each module, and also the message channels between them. The GASCA OS uses these blueprints to know where and when to start each partition, and also to create and reserve each message channel.

A set of blueprints has to be generated for each possible configuration. In the GASCA demonstrator, seven different configuration scenarios have been developed each containing either a module or a network failure. In a production system, a configuration would have to be defined for each possible failure. This means that for *m* modules and *n* networks, 2^{m+n} configurations have to be defined each involving up to m^p choices, p being the number of partitions to be distributed. This clearly requires the use of an automated blueprint generation tool. This tool has been developed, with a subsystem allowing the generation of the different distribution scenarios. For each scenario, the best configuration is the one that minimises the cost factor:

$$Q_i = \frac{SF^2 \bullet w_{SF} + IF^2 \bullet w_{IF} + UF^2 \bullet w_{UF}}{w_{SF} + w_{IF} + w_{UF}}$$

where SF, the separation factor, that tries to minimise the number of critical applications in a module is:

$$SF_i = \sum_{cl=1}^{5} \frac{\frac{1}{m}\sum_{j=1}^{m}\left(\frac{\text{\#apps level cl on module j}}{\text{Total apps level cl}}\right)^2 \cdot w_{cl}}{w_1 + w_2 + w_3 + w_4 + w_5}$$

IF, the intercommunication factor, that tries to minimise the message traffic in the network is:

$$IF_i = \frac{\sum(\text{Weight of edges in reduced communication graph})}{\sum(\text{Weight of edges in communication graph})}$$

UF, the resource utilisation is given by:

$$UF_i = \frac{U_{CPU} * w_{CPU} + U_{Mem} * w_{Mem} + U_{I/O} * w_{I/O}}{w_{CPU} + w_{Mem} + w_{I/O}}$$

where U_R, the usage of resource R is:

$$U_R = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\text{Total R used on module i}}{\text{Maximum available R on module i}}\right)^2$$

This algorithm was designed by NLR and was developed by Skysoft for the demonstrator. Several parameters such as partition ordering (partitions that send messages to others have to be scheduled first),

period feasibility (assuring the module is able to schedule all the partitions with their desired period) and different network capacities were identified, but not included in the algorithm as that would require a distribution tool too complex for the demonstrator.

System Performance Model

In order to validate and prove the conclusions of the GASCA project, a system performance model (SPM) of all the GASCA system components was developed by DERA. The development of the SPM was, in itself, a useful task. It brought an insight into the problems and challenges faced in the construction of a real GASCA. It remains a useful tool for examining the protocols and mechanisms involved in the concept. The SPM has a further role within GASCA, which is to estimate the performance of a particular aircraft system based on the architecture. This model will provide a proof of the GASCA concept, which would otherwise be purely hypothetical. The SPM will require validation data, measured from the demonstrator, to be input into the model.

The demonstrator exists in order to evaluate the rules and guidelines on which GASCA is based, and the SPM will support it in these assessments. The SPM may also be used beyond the limits of this demonstrator, and thus assess a greater range of systems and system limits.

References

ARINC 653, Avionics Application Software Standard Interface. January 1997.

GANL0003, WI 2.1.2, Software Partitioning and Distribution, NLR.

GADE0034, WI 2.2.5, API Definition. Detexis.

GADE0042, WI 2.9.1, Report on the review. Detexis.

GADE0046, WI 2.9.8, Preliminary architecture guidelines. Detexis.

GART0015, WI 3.1.1, Software Definition Report. Skysoft.

GADE0045, WI 3.1.3, OS definition report. Detexis.

GADE0060, WI 3.3.8, System Software documentation. Detexis.

GAGM0060a, System Software Document Fault Tolerant Components Volume 2, Marconi Electronic Systems.

GAGM0023, Proposed Static Configurations To Demonstrate Fault Tolerance. Marconi Electronic Systems.

GAGM0026, The GASCA Fault Tolerance Data Structure Designs. Marconi Electronic Systems.

GADR0007, WI 3.5.5 – 3.5.8 GASCA Systems Analysis, DERA.