Building High-Integrity Distributed Systems with Ravenscar Restrictions *

Santiago Urueña Juan Zamorano

Department of Computer Architecture and Technology (DATSI)
Technical University of Madrid (UPM)
E28660 Boadilla del Monte, Spain

E-mail: Santiago.Uruena@upm.es, JuanRafael.Zamorano@upm.es

Abstract

The Ravenscar profile was a qualitative leap in the development of single-processor hard real-time systems with certification requirements. But nowadays more and more safety-critical systems are distributed, so a new Ada profile is needed for multi-node applications. This work discusses the restrictions and additions to the language needed to certify and obtain the required predictability and timeliness in a high-integrity hard real-time Ada distributed application.

1. Introduction

Nowadays, several types of High-Integrity Systems (HIS) —specially safety-critical ones— have hard real-time requirements and must execute in an embedded distributed hardware. A distributed system is not only needed when different parts of the system are physically distant. It also provides better fault-tolerance, isolates applications with different criticalities among the nodes, and gives more processing power than a single CPU.

The Ravenscar profile was designed for HIS with strict temporal requirements [6], and where response-time analysis (RTA) methods [10] are needed, easing the development and certification of those types of systems. It was one of the major additions to Ada 2005 [1], and there are multiple commercial implementations [3]. However, the profile was targeted for mono-processors and thus no special support was designed for the development of distributed systems.

Although the Distributed Systems Annex (DSA) is not forbidden in the Ravenscar profile defined in Ada 2005 [9, § D.13.1] it presents some problems that make this Annex E inadequate for hard real-time distributed systems [14]. This is not surprising because the DSA was designed for general purpose distributed systems, although the implementators are allowed to extend and adapt the annex with new functionality and rules [9, § E.2(14), E.2.1(8.b), E.5(26), E.5(27.1/2)].

Therefore, Ravenscar applications can currently use the DSA in a "portable" way only for non real-time communication with other nodes. The goal of this paper is to restrict the DSA following the philosophy of the Ravenscar profile to enable the certification of the middleware, to achieve the required degree of predictability and timeliness, and to discuss the most important additions needed for the development of distributed hard real-time safety-critical systems.

This paper covers distributed applications running on a physically distributed system (different computing nodes connected through a bus or a local network), or on the same node (logical partitions, like ARINC-653). The existence of a reliable hard real-time communication network is of paramount importance. However, this paper is not aimed at multi-processor systems, except if each partition can execute only in the same CPU (i.e. no 'task migration').

This paper is organized as follows. Section 2 describes the related work, including the scheduling theory. Section 3 discusses the advantages and disadvantages of different distribution mechanisms used by the software industry. Section 4 defines a set of restrictions for building safety-critical distributed systems, while section 5 talks about the implementation requirements. Section 6 presents two use cases. Finally, section 7 summarizes the main conclusions of this work.

2. Related work

This paper builds upon current advances in scheduling theory for distributed hard real-time systems. Tindell and Clark [24] extended the response time analysis techniques used for single processors to distributed systems, introducing the concept of holistic schedulability. Later, Palencia

*This work has been funded in part by the Spanish Ministry of Science and Technology (MCYT), project TICTIC2005-08665-C03-01 (THREAD), and by the Council for Education of the Community of Madrid and the European Social Fund.

and González Harbour [16] improved the technique to reduce the pessimism of transactions.

Work about adapting the Ada distributed system annex for real-time systems includes past IRTAW sessions [14][7] and two different implementations: RT-GLADE [13], a research real-time implementation of the DSA, and PolyORB [25], a commercial real-time middleware with different distribution models, including RT-CORBA and the DSA.

Finally, there is also some research about the specific topic of Ravenscar compliant distributed systems. On the one hand, some middlewares are implemented using the Ravenscar profile —PolyORB, mentioned above, and DEAR-COTS [18], a distribution framework mainly designed for fault tolerance— and on the other hand work by Audsley and Wellings [4] in IRTAW 2000 about using the DSA and Ravenscar for High-Integrity Systems.

3. Selection of a distribution mechanism

Different distribution mechanisms for Ada are currently used in industry, some of them for specific environments (e.g. SOIS [19] for the space domain, or MPI [23] for scientific computations) while others are multi-purpose, like the Real-Time Common Object Request Broker Architecture (Real-Time CORBA), the Data Distribution Service (DDS), or the Distribute Systems Annex (DSA). The standard distribution mechanism of Ada has different advantages and disadvantages with respect to other middlewares.

3.1. DSA advantages

The Distributed Systems Annex is integrated in the language since Ada 95, and was designed for a wide variety of distributed systems. It is easy to learn [11], and a full distributed application can be developed very quickly adding only some categorization pragmas to specific library units, and specifying the location of each partition using the configuration file. Automatic consistency checks at the start of the application ensure the binary compatibility of each partition [9, § E.3(6)].

Every distributed application is also a valid centralized program, therefore the same application can be first compiled and tested as a single binary and later the exactly same source code can be compiled as a distributed program [9, § E(7)]. This is very useful during the development stage, because debugging a distributed application is far more difficult than a centralized one. Although this can also be achieved using other distribution mechanisms it would require the modification of the source code (and thus bugs can be introduced in the process).

Other advantages of the distribution model (categorization pragmas versus the API of other middlewares) are the compiler checks among partitions for code verification and optimization [9, § E(8)], and for external tools like the SPARK examiner [5], which does not need to be modified to analyze the application. It should be investigated whether the DSA facilitates writing a tool to obtain the temporal model of a system from the source code.

The DSA provides multiple communication paradigms including message passing, distributed shared memory [12], remote procedure calls, and distributed objects. The Ada specification specifies a standard communication subsystem (System.RPC), but implementations are free to generate calling stubs that leverage other underlying middlewares [9, § E.5(27.b/2)] (e.g. DSA implementations exits for CORBA, web services [25] or Java RMI [17]). Each partition has an independent run-time system thanks to the purity rules enforced by the categorization pragmas.

3.2. DSA disadvantages

However, the DSA has also some problems. It was not designed for real-time communication, in opposition with DDS, RT-CORBA, or SOIS. It has also limited vendor support, and there are only a few research and commercial implementations, while CORBA or SOIS have even implementations for safety-critical software.

Although the Ada 95 language designers created a modular scheme to encourage that the compiler and communication subsystem were created by different vendors, in practice there is no interoperability among the implementations. And consistency checks make difficult the implementation of open systems and dynamic services.

Like in other distribution mechanisms (e.g. Java RMI), remote tagged objects cannot migrate to other nodes (they must be limited) so remote dispatching operations are always served by the node that created such tagged object. However, in RMI there is a default serialization for all types, but in the DSA the application programmer must provide the code for the marshalling and unmarshalling of data involving some access types—e.g. a linked list.

The purity rules are very strict, and only preelaborated types can be transferred. For example, no standard time type can be used in a remote operation, the application programmer is forced to define one. In practice, purity rules difficult the reuse of code not designed for a distributed application. However, the same applies to the rest of distribution mechanisms.

Finally, the DSA does not provide the Publish/Subscribe [20] communication paradigm, in opposition with DDS. This is an efficient real-time communication paradigm that allows multicast communications, and thus it is very interesting for control systems (especially since the schedulability analysis of multi-event synchronization can be achieved [8]).

3.3. Distribution mechanism chosen

Some of the above disadvantages are not intrinsic to the definition of the DSA but implementation dependent, like interoperability or real-time communication support. The ARM allows the addition of new rules, categorization pragmas, or interfaces of new distribution services. But it is worth noting that if those services are not standarized vendors have little motivation to implement them.

However, some of those disadvantages are not a problem for a HIS. For example, although CORBA offers better interoperability among different vendors this is not usually a core concern in a HIS because the developers have more control over the whole software stack. Although complex distributed high-integrity systems are usually developed by more than one contractor, each safety-critical subsystem is made by only one.

Reuse of past code written in different languages is also desirable, but for HISs the testing and certification steps are more costly. As said above, the DSA simplifies testing because exactly the same source code can be tested as a distributed or centralized program, and certification is easier thanks to the whole application checks and because less code is written due to its higher abstraction level compared to the others communication mechanism (on the other hand, if more code is generated the programmer has less control of the code, so this should be further investigated).

Some DSA problems found in Ada 95 has been fixed in the technical corrigendum 1 (like some defects related with heterogeneous systems [2]) and in Ada 2005 (more implementation freedom because it is not required to use the partition communication subsystem interface defined in System.RPC). In summary, the DSA is a good foundation for the development of distributed systems, namely HIS ones. In the next section we will discuss the specific problems that must be fixed in the DSA to be able to develop a hard real-time distributed HIS.

4. Restrictions

The Annex E is not disallowed in the Ravenscar profile as defined in Ada 2005. Moreover, at least one of the DSA implementations —PolyORB— can be configured to be Ravenscar compliant. However, the language standard is not designed for distributed real-time systems, so a set of changes is needed to adapt the Annex E to safety-critical distributed systems with hard deadlines.

Some features needed for basic real-time communication were proposed in the past, like messages priorities [14] or non-blocking asynchronous RPCs [7]. These were successfully implemented in RT-GLADE [13], a research real-time DSA prototype. In the future, another desirable addition would be Publish/Subscribe, but the compatibility with the current distribution model should be further investigated.

But for high-integrity systems additional restrictions are needed if the code must be certified, and also to predict if all deadlines will be met. We will restrict the DSA according to current response time analysis theory for distributed systems. The objective is to introduce the minimum restrictions to retain the required programming expressiveness and to permit the reuse of existing code.

4.1. Compulsory restrictions

The restrictions were designed to obtain the required degree of predictability and timeliness needed in a hard real-time system, and to simplify the implementation of a safety-critical middleware, thus easing its certification. In some cases, a restriction could also be introduced to increase performance. And because implementors do not want to maintain more than one specialized run-time, the restrictions must also be compatible with the existing Ravenscar profile. This is the set of restrictions proposed:

- **No remote access types**: A connection for each remote operation is created at elaboration time between the sender and receiver partitions. It is never closed, and new connections should not be established after elaboration time to avoid an excessive blocking time (e.g. similar to the No_Local_Protected_Objects, or No_Task_Allocators restrictions). Therefore, the exact number of interconnections must be known at compile time (actually when the distributed application is configured [9, § E.1(15)], just before the binary for each partition is generated). This can be hard to predict if remote-access-types (access types to remote subprograms and to remote class-wide-types) are allowed. Another advantage is that the set of resources needed for all connections can be predicted at compile time.

- **No concurrent remote calls**: For predictability reasons, a remote operation cannot be called while processing a past invocation (to the same remote subprogram). This implies there is no wait queue in every remote operation, easing the response time analysis. This is similar to the existing restriction Max_Entry_Queue_Length => 1. Another advantage derived from this rule is the absence of loops in the call graph, so no distributed deadlock can occur [21] if there is one RPC handler for each operation of the interface (static allocation of incoming operations, i.e. there is no thread pool). Violations to this rule cannot be detected by the compiler, only at run-time or with tools similar to the SPARK examiner.

- **Coordinated elaboration**: A distributed application cannot start until all its partitions had been elaborated.

The DSA is designed for general purpose distributed systems, where some partitions can start executing before others, for example in a client-server scheme [9, § E.1(13)]. However, in a real-time system it is not acceptable to enqueue a remote call until the invoked partition completes its elaboration [9, § E.4(14)]. It also implies that there is a static number of partitions at compilation time. This restriction is similar to the sequential elaboration policy specified by the Ada 2005 pragma Partition_Elaboration_Policy , but it should be noted that this pragma (from Annex H) is not required by the Ravenscar profile.

One implication of the first restriction is that it would disallow the distributed object paradigm. Other concern about no dynamic connections is how to achieve fault tolerance, one of the potential advantages of a distributed system. Transparent partition replication can be the answer (integrating a framework for fault tolerance directly in the DSA implementation instead of in each application [18]), but this issue must be further investigated.

The second restriction is needed for predictability reasons, as done to protected objects when removing the entry queue. And if only one task can be associated with a protected entry, the same also applies with each remote subprogram. Note that local deadlocks where avoided in Ravenscar thanks to the Priority Ceiling Protocol, so the absence of all types of deadlocks is an interesting property for High-Integrity Systems.

It should be noticed that partition termination is already disallowed: In the full DSA a partition can terminate either because all their tasks have finished or when its environment task is aborted. But this cannot happen in this high-integrity DSA because both restrictions are already enforced by the Ravenscar profile (No_Task_Termination and No_Abort_Statements). No partition termination implies that a remote subprogram call cannot be cancelled [9, § E.4(13)], simplifying the implementation of the middleware.

4.2. Optional restrictions

Other restrictions were considered but later discarded because —although useful for some types of high-integrity systems— are not essential to perform a response time analysis of the system or to simplify the implementation of the run-time:

- *Synchronous calls*: if only asynchronous communication is allowed then the RTA is simplified, but this will reduce the programming expressiveness. Audsley [4] proposed to disallow synchronous calls to avoid excessive blocking time, but newer RTA methods reduce the pessimism introduced in that situations [16]. However, only asynchronous operations should be used if the distributed application is implemented in SPARK. Otherwise, an exception can be raised if there is a communication error while performing a synchronous call.

- *Nested calls*: if a synchronous remote subprogram cannot perform another (blocking) remote call before returning to the caller (a chain of calls) the response-time analysis is greatly simplified. However, the programming expressiveness will be also greatly reduced, and although no nested calls is a sufficient condition to avoid distributed deadlocks, they are already avoided if concurrent calls are disallowed.

- *Complex interpartition communication*: if unconstrained types or complex data structures (e.g. linked list) are used as parameters in a remote operation, it could be impossible to calculate the size of the maximum message. But these types (if correctly used) do not necessarily introduce any schedulability problems, and the programming expressiveness would be greatly reduced if this restriction is introduced.

Ravenscar deals with concurrent code, and disallowing the transmission of these types would be equivalent to restrict a sequential construct. The application programmer should be allowed to have dynamic size messages, but depending on the level of certification required these types can be disallowed with the aid of external ASIS tools.

It is worth noting that the programmer must provide the adequate marshalling and unmarshalling code ('Write and 'Read attributes) for types composed by *non-remote* access types (see example 1). Therefore the programmer is aware of the serialization costs, and the run-time does not have to handle the serialization of complex data (like recursive types [22]). Note that remote access to wide-access types has no serialization problems because they must be limited. They are not disallowed due to serialization costs but to avoid dynamic connections.

4.3. Supported features

To summarize, the distribution features supported are:

- **Passive partitions**: shared passive or pure packages including atomic and volatile variables, and protected objects (without entries).

- **Static remote subprogram calls**: remote type packages as restricted above, and remote call interface library units.

- **Unconstrained parameters**: unconstrained types and (non-remote) access types are allowed in remote calls.

- **Synchronous and asynchronous communication**: Synchronous communication for active and passive

partitions, and pragma Asynchronous to enable asynchronous communication. Pragma All_Calls_Remote is also allowed, useful mainly for code debugging.

5. Implementation requirements

It is desirable that a task invoking a remote operation does not delegate the message generation (including data marshalling, message partitioning, composition of message headers, and even message queueing) to another task to avoid priority inversion. The network is usually non-preemptable, so total priority inversion is in general not possible but it can be bounded. The receiver should then process each call with the priority specified in the message.

Each remote operation should be processed in the called partition by a specific thread, including each instantiation of a generic remote call interface (probably each generic instantiation will be in different active partitions, but when more than one is located in the same partition a common thread for each operation is not allowed). Therefore, the ARM requirement to have a reentrant RPC handler is no longer needed in this restricted DSA.

The Program_Error exception is sent back to the caller in the case the destination thread is still processing another call, as done in the Ravenscar profile for tasks trying to access an entry of a protected object in which another task is already waiting. This cannot be detected at the calling partition, only when the calling petition arrives to the server, wasting some bandwidth. However, this should only happen in the testing phase because it implies that the program is erroneous.

It should be noticed that each partition can still have an independent run-time system. No clock synchronization is needed because the communication is message oriented [15, p. 1.27], but of course a mechanism to obtain a certain degree of common time is desirable in a real-time system. This should be further investigated.

The implementation must document the communication process, specifying if any step is delegated by another task in the caller or called partition. It must be further investigated the metrics that should be documented by the implementation.

6. Examples

6.1. Fault-Tolerant inter-node communication

In the first use case the Flight Management System communicates with different nodes, either using a high-speed network, or through a bus (when the bandwidth requirements are low). The communication links are replicated for fault-tolerance: To recover from transmission errors, and to tolerate a broken link.

In the code of example 1 the Flight Management System communicates with the Flight Plan Manager and with the fuel-level sensor. The fuel-level sensor has low bandwidth and CPU requirements, executing over a microcontroller (minimal run-time system, no tasking). The sensors are also replicated. The Flight Plan Manager sometimes has high-bandwidth requirements because in this example it must transfer the complete planned route as a linked list to the Flight Management System.

In our fictitious DSA implementation the middleware handles transparently the replicated networks. But it should be noticed that the replicated sensors are managed by specific application code and not by the DSA implementation.

6.2. Criticality segregation

Usually, the software of a high-integrity system has different criticality levels. For example, Level A code is considered mission critical, while Level B code will not lead to catastrophic events if the software fails. A lower criticality level implies less certification requirements, and thus different verification and validation costs.

In this use case two hypothetical applications of different criticality levels execute in the same hardware node, a common approach in Integrated Modular Avionics (IMA). The RTOS provides a different memory space and CPU budget for each one, and a shared memory region for communication between them. In the DSA terminology, each application is an *active partition*, while the shared memory region is a *passive partition* [9, § E.1(2)].

As can be seen in the example 2, the Flight Management System writes the telemetry data in the passive partition (pragma Shared_Passive), while a task of the lower criticality partition updates the displays. In our hypothetical DSA implementation the Level A partition can be configured to have R/W access to the passive partition and read access only to the Level B partition. Therefore both applications are completely isolated (so they can be certified at different criticality levels) while the communication is very fast and completely asynchronous.

7. Conclusions and future work

This position paper has discussed the changes needed in the Ada Distributed Systems Annex (DSA) for developing safety-critical distributed systems. Although currently the DSA cannot be used in a distributed system with hard real-time communication requirements, it is argued that the Annex E is more adequate for this kind of High-Integrity Systems than other industrial middlewares.

This paper briefly describes some of the real-time capabilities needed for basic real-time communication, and restricts the DSA to enable certification and to obtain the

required degree of predictability and timeliness. The resulting profile is compatible with Ravenscar, and it is believed it has enough programming expressiveness for the development of complex safety-critical hard real-time embedded distributed systems.

Finally, some topics like the documentation requirements and fault-tolerance (probably through transparent replication) must be further investigated. And a prototype is needed in the future to validate the proposed distribution mechanism, and to prove whether it can be successfully implemented and certified.

References

[1] Ada Rapporteur Group. Ada Issue 249 — Ravenscar profile for high-integrity systems. *Ada Letters*, XXV(3), September 2005.

[2] Ada Issue 208 — What is the meaning of "same representation" in all partitions?, August 1999.

[3] L. Asplund, B. Johnson, and K. Lundqvist. Session summary: The Ravenscar profile and implementation issues. *Ada Letters*, XIX(25):12–14, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[4] N. Audsley and A. Wellings. Issues with using Ravenscar and the Ada distributed systems annex for high-integrity systems. In *IRTAW '00: Proceedings of the 10th international workshop on Real-time Ada workshop*, pages 33–39, New York, NY, USA, 2001. ACM Press.

[5] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[6] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3 edition, 2001.

[7] J. J. Gutiérrez and M. González Harbour. Towards a real-time Distributed System Annex in Ada. *Ada Letters*, XXI(1), 2001.

[8] J. J. Gutiérrez, J. C. Palencia, and M. González Harbour. Schedulability analysis of distributed hard real-time systems with multiple- event synchronization. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pages 15–24. IEEE CS Press, June 2000.

[9] ISO SC22/WG9. *Ada 2005 Annotated Reference Manual. ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*, 2006. Available on `http://www.adaic.com/standards/ada05.html`.

[10] M. Joseph and P. Pandya. Finding response times in real-time systems. *BCS Computer Journal*, 29(5):390–395, 1986.

[11] Y. Kermarrec. CORBA vs. Ada 95 DSA: a programmer's view. In *SIGAda '99: Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, pages 39–46, New York, NY, USA, October 1999. ACM Press.

[12] P. Ledru and S. G. Shiva. Interpartition communication with shared active packages. In *TRI-Ada '96: Proceedings of the conference on TRI-Ada '96*, pages 57–62, New York, NY, USA, 1996. ACM Press.

[13] J. López Campos, J. J. Gutiérrez, and M. González Harbour. The chance for Ada to support distribution and real-time in embedded systems. In A. Llamosí and A. Strohmeier, editors, *9th International Conference on Reliable Software Technologies — Ada-Europe 2004*, number 3063 in LNCS, pages 91–105, Palma de Mallorca (Spain), 2004. Springer-Verlag.

[14] S. A. Moody. Session summary: Distributed Ada and real-time. In *IRTAW '99: Proceedings of the ninth international workshop on Real-time Ada*, pages 15–18, New York, NY, USA, March 1999. ACM Press. Chairman: Michael González Harbour. Rapporteur: Scott Arthur Moody.

[15] J. C. Palencia Gutiérrez. *Análisis de planificabilidad de Sistemas Distribuidos de Tiempo Real basados en prioridades fijas*. PhD thesis, Universidad de Cantabria, 1999. Supervisor: Michael González Harbour.

[16] J. C. Palencia Gutiérrez and M. González Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *RTSS 1999: Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 328–339, December 1999.

[17] L. Pautet and S. Tardieu. What future for the distributed systems annex? In *SIGAda '99: Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, pages 77–82, New York, NY, USA, 1999. ACM Press.

[18] L. M. Pinho and F. Vasques. Using Ravenscar to support fault tolerant real-time applications. *Ada Letters*, XXII(4):47–52, 2002.

[19] C. Plummer and P. Plancke. The spacecraft onboard interfaces, SOIS, standardisation activity. In *DASIA 2002 — Data Systems in Aerospace*, 2002.

[20] R. Rajkumar, M. Gagliardi, and L. Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings of the First IEEE Real-Time Technology and Applications Symposium (RTAS'95), 1995.*, pages 66–75, Los Alamitos, CA, USA, May 1995. IEEE Computer Society.

[21] C. Sánchez, H. B. Sipma, Z. Manna, V. Subramonian, and C. Gill. On efficient distributed deadlock avoidance for real-time and embedded systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.* IEEE Computer Society, April 2006.

[22] D. Tejera, A. Alonso, and M. Á. de Miguel. Predictable serialization in Java. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'07)*, May 2007.

[23] K. Thomas. Parallel programming in Ada 95 and MPI. *Ada User Journal*, 21(2):143–152, July 2000.

[24] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2–3):117–134, April 1994. Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems).

[25] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106–119, Palma de Mallorca, Spain, June 2004. Springer Verlag.

Example 1. Internode communication. Of course, a Remote Call Interface is assigned only to one active partition. A Remote Types unit is needed in this example because access types cannot be declared in a Remote Call Interface.

```
-- Node 1
package Sensors.Fuel is
   pragma Remote_Call_Interface;

   type Fuel_Level is delta 0.001 range 0.0 .. 100.0;
   function Current_Fuel_Level return Fuel_Level;
end;

-- A Remote Types unit is replicated in every partition that includes it (node 2 and 3).
with GPS;
with Ada.Streams;
package Routing is
   pragma Remote_Types;

   type Flight_Plan is private;
   procedure Flight_Plan_Write (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
                                Item   : in  Flight_Plan);
   procedure Flight_Plan_Read  (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
                                Item   : out Flight_Plan);
   for Flight_Plan'Write use Flight_Plan_Write;   -- user-defined marshalling
   for Flight_Plan'Read  use Flight_Plan_Read;    -- user-defined unmarshalling
private
   type Flight_Plan is record   -- Linked list
      Waypoint : GPS.Coordinates;
      Next     : access Flight_Plan;
   end record;
end;


-- Node 2
with Routing;
package Flight_Plan_Management is
   pragma Remote_Call_Interface;

   procedure Planned_Route (Route : out Routing.Flight_Plan);
end;


-- Node 3
with Sensors.Fuel;
with Routing;
with Flight_Plan_Management;
procedure Flight_Management_System is
   Fuel  : Sensors.Fuel.Fuel_Level;
   Route : Routing.Flight_Plan;
   -- ...
begin
   loop
      -- ...
      Fuel := Sensors.Fuel.Current_Fuel_Level;
      -- ...
      Flight_Plan_Management.Planned_Route (Route);
      -- ...
   end loop;
end;
```

Example 2. Criticality segregation. A Shared Passive unit can be assigned only to one partition because it has state and can declare public variables. It is preelaborated, and can depend only on Pure units or other Shared Passive packages.

```
-- Shared memory area partitions Level A and B
with Instruments;
package Telemetry is
   pragma Shared_Passive;

   Current_Altitude  : Instruments.Altitude;
   Current_Latitude  : Instruments.Latitude;
   Current_Longitude : Instruments.Longitude;
   Current_TAS       : Instruments.True_Airspeed;
   pragma Atomic (Current_Altitude);
   pragma Atomic (Current_Latitude);
   pragma Atomic (Current_Longitude);
   pragma Atomic (Current_TAS);
end;



-- Partition Level B
with Telemetry;
package body Displays is

   task body Display_Manager is
      -- ...
   begin
      loop
         -- ...
         Print_Display1 (Telemetry.Current_Altitude,
                         Telemetry.Current_Latitude,
                         Telemetry.Current_Longitude,
                         Telemetry.Current_TAS);
         -- ...
      end loop;
   end;
end;


-- Partition Level A
with Telemetry;
with Sensors.Altitude;
procedure Flight_Management_System is
   -- ...
begin
   loop
      -- ...
      Telemetry.Current_Altitude := Sensors.Altitude.Current_Altitude;
      -- ...
   end loop;
end;
```