Middleware for a Distributed and Hot-redundant Software in Ada 2012

Vincent Monfort

Systerel Paris, 3 Rue Danton, 92240 Malakoff; email: vincent.monfort@systerel.fr

Abstract

In railway control systems, distributed, available and reliable software is a recurrent need. As a consequence a middleware is a convenient solution to handle these aspects and provide guarantees to the application software. In this context, we developed a new Ada 2012 middleware for execution of distributed and hot-redundant software which must conform with EN50128 standard. This article presents the particular needs of such a middleware, the technical challenges and solutions, and a feedback on the Ada 2012 language and tools.

Keywords: Ada 2012, SMP, middleware, distributed software, hot-redundant software, railway control system, critical software, EN50128.

1 Introduction

For the development of its new generation of underground railway Integrated Control Center, Alstom Transportation has decided to develop a new Ada 2012 middleware after it used an Ada 83/95 one since 20 years. The reasons to develop a new middleware from scratch were the need of multi-platform and symmetric multiprocessing support on the one hand and the advantages of using the latest features of the Ada 2012 language and development tools to simplify middleware architecture on the other hand. The middleware main characteristics are a generic and high level interface to host a supervision software and to hide the mechanisms of communication, distribution (not using Annex E [1]) and hot-redundancy services provided to the application software. Moreover it must guarantee performance and high availability to the application software which in addition must conform to the EN50128 standard.

2 Context

A simplified view of an underground railway system can be described as follows, on the one side there are the wayside equipment (signaling, switches, presence sensors, etc.) and rolling stock equipment, on the other side there are the control system receiving statuses from the equipment and sending commands to the equipment. In this example (see figure 1), the control system is composed of distributed and redundant machines:

- The server is in charge to normalize status and command data, this server is duplicated on 2 redundant machines SRV1 and SRV2,
- Several control station machines (SC1, SC2 and SC3) are connected to the server (distribution).

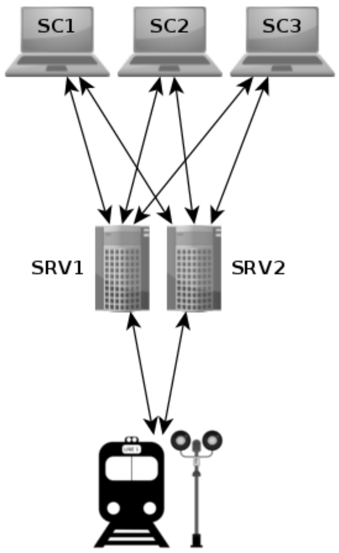

Figure 1: Example of a railway control system architecture.

In such a system, the goal of a hot-redundancy mechanism is to guarantee availability of the server services by switching from SRV1 to SRV2 (or vice versa) in a short time in case of failure of SRV1 (resp. SRV2). Each of the control station machines needs to be notified about the status changes from the wayside and can send commands to the wayside. As a consequence the middleware must reach these objectives to comply with an underground railway system needs.

3 Middleware principles

The next sections describe the main principles of the middleware which are based on the configuration of the application architecture and the execution of the application composed of application functions in different processes.

3.1 Application software architecture

In the context of the middleware, the application software architecture is composed of processes which can be executed on different machines (each process on one machine) and redundant processes which can be executed on two machines (each redundant process on two machines). Each process runs one or several application functions containing the application code of the software and which can communicate with other determined functions. The description of this architecture must be done statically for the application, through an XML configuration file, in order for the middleware to hide the mechanisms of communication between the processes, distribution and hot-redundancy.

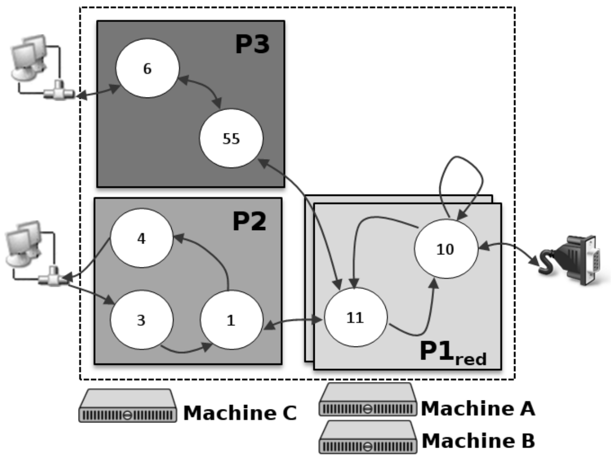

Figure 2: Example of an application architecture configuration.

Figure 2 describes an example of application architecture. It describes the machines, the processes (as squares), the application functions (as numbered circles) and configured exchanges between these functions. More particularly:

- Processes P2 and P3 are hosted on machine C: P2 runs functions 1, 3 and 4, P3 runs functions 6 and 55,
- Process P1 is redundant and then hosted on 2 machines A and B: P1 runs functions 10 and 11.

The corresponding XML configuration sample for process P1 is the following:

```
<process name = "P1">
  <hosts>
    <host name = "MachineA"/>
    <host name = "MachineB"/>
  </hosts>
  <functions>
    <function id="10"/>
    <function id="11"/>
  </functions>
  <processes_to_connect>
    <process_to_connect name = "P2"/>
    <process_to_connect name = "P3"/>
  </processes_to_connect>
</process>
```

In addition the functions can be configured to be executed on a particular CPU core with a relative priority:

```
<function id = "10" priority = "high" cpu = "1"/>
```

3.2 Execution of application functions

The execution of the application hosted by the middleware is message oriented, which means execution is based on message exchanges between functions. Since application architecture is configured, a function can send messages to (resp. receive messages from) another function without the need to take care of its process nor machine location. The exchanged messages must be defined by the application as a derived type of the abstract message type:

```
type Fid_T is range 0 .. 255; -- Function identifier type

type Message_T (From : Fid_T;
                To   : Fid_T) is abstract tagged private;

-- Application defined type for messages from function 10 to 11
type Msg_10_To_11_T is new Message_T(From => 10, To => 11)
with record ... end record;
```

A function execution is orchestrated by the reception of messages which triggers application code execution, it is then possible to execute application treatment, store data in middleware specific concurrent data structure and send messages to other functions. Each function defines the contents of the procedure, with predefined signature, called on message reception and can send messages using the predefined Send procedure:

```
-- Signature of procedure to be defined by each function
procedure Process_Message (Message : in Message_T'Class);

-- Predefined procedure to send messages
procedure Send (Message : in out Message_T);
```

An example of a function implementation can then be the following:

```
procedure Process_Message
      (Message : in Message_T'Class) is
   MyMsg : Msg_10_To_11_T; -- Declare a new message
begin
   ...                -- Treatments of incoming message
   MyMsg.Data := ...; -- Set the message contents to send
   MyMsg.Send;        -- Send the message
end;
```

The middleware is then in charge to deliver the message to the correct function and guarantee the delivery to the target function running in the configured process and machine.

Note: middleware guarantees are provided for a controlled and nominal application execution. It means application code does not lead to crash and can afford to treat the amount of messages it sends (if it is not the case it can be detected using a configuration option setting a limit of messages in queue).

3.3 Details of the hot-redundancy mechanism

The redundancy mechanism is mainly based on a Master/Slave mode for a redundant process, which means a redundant process instance is either active (Master mode) or passive (Slave mode) on an application functional point of view. If the Master redundant process instance has a failure, then the Slave instance will switch to Master mode and continue the treatment of messages where it was stopped. Figure 3 shows an example in which the Master instance of process P1 failed on machine A and the Slave instance switched to Master mode on machine B. In order to meet the requirements of hot-redundancy, the middleware provides the guarantee that message processing by a function is atomic and no message is processed twice or dropped in regular circumstances (nominal application behavior). These properties can be enforced thanks to a few mechanisms:

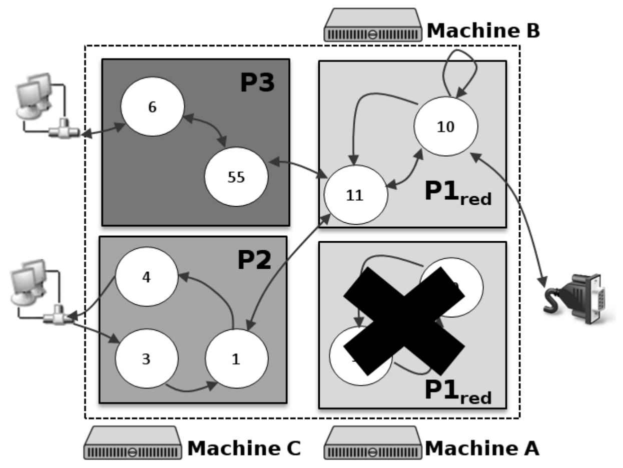

Figure 3: Example of redundant process switch from Slave to Master mode.

- Messages sent to a target function in a redundant process are transmitted to both redundant process instances,
- Messages treatment state is synchronized between the two redundant instances of a function,
- Application code must use specific redundant data structure which data are synchronized and modified in an atomic way regarding a message treatment.

As a consequence, on the application side, there are only a few constraints to implement a function in a redundant process. The application code must use only the provided redundant data structures, which are similar to classical containers such as vectors and maps, and must not store data locally or send a message referencing local contents.

4 Feedback on the Ada 2012 language and tools

This section presents feedback on the latest Ada language version and associated development tools (GNAT Pro tools). In the first place, the new middleware benefits from the use of Ada 2012 features and the first remarkable benefit is a code size reduction of about 80% compared to the precedent middleware version.

4.1 Features use

Most of the new Ada 2012 features were used to develop the middleware. The next sections describe how the new features have been used and justify the reasons for which some features were not used.

4.1.1 Multi-task and task-safe treatments

The middleware architecture makes an important use of tasks since each process uses at least the following tasks:

- client connection task: in charge to send messages from functions to the network,
- server connection task: in charge to receive messages from the network to be transmitted to target functions,
- function task: each application function execution is managed by a different task,
- state manager task: in charge of managing the redundancy state of the process.

In order to manage the messages exchanged between these tasks the *Task-safe queues* were used in combination with the *Holders* in order to store in the queues the abstract tagged ancestor message type. Indeed, the function tasks use the client queue to add messages to be sent by the client task, and the server task uses the function task queues to transmit received message to target functions.

Moreover *Synchronized barriers* are used to synchronize these tasks after initialization phase.

4.1.2 Multi-processor affinity

The *Multi-processor affinity* and *Task priority* feature are used by the middleware to configure the core tasks (client, server, etc.) and the function tasks to run on a particular CPU with a relative priority. This is important since the applications developed are intended to be used on servers with at least 8 cores.

4.1.3 Contract programming

The *Preconditions and postconditions* were widely used for the middleware development first, and then to expose the expectations and obligations of the API for the hosted application. Contract programming was really relevant to ensure quick and efficient development integration and validation.

4.1.4 Expressiveness

The use of *Conditional expressions*, *Quantified expressions* and *Expression functions*, but also of *In-Out function parameters* and *Iterators*, really improved the expressiveness and readability of the middleware code. These features contributed a lot to the code size reduction.

4.1.5 Unused features

Few of the new Ada 2012 features were not used for the middleware but we can notice the following ones.

Type invariants are not used since middleware types do not have strong internal constraints and the *Preconditions and postconditions* were widely used to express constraints between different parameters. This is certainly due to the middleware nature of the software.

Subtype predicate are also absent since only integer subtypes were defined and used only the range definition as constraint.

Ravenscar for multiprocessor systems was not used since the middleware does not use a Ravenscar profile.

In addition, as indicated in the introduction, the existing Annex E [1] was not used for distributed aspects since it was not suitable to the redundant mechanisms and encapsulation in the middleware. There was also a guarantee that we could handle the possible technical and performance issues independently to reach our goals.

4.2 Development tool use

GNAT Pro tools were used to develop in Ada 2012 and reach our requirements in terms of portability, quality and performance.

4.2.1 Operating System dependency

In order to be independent of the Operating System, the GNAT Pro libraries were used.

First of all, the GNAT sockets, used as streams, are an important feature for the middleware. Sockets are used for message exchange between processes or process instances, these messages can be application messages but also internal messages to initiate or maintain communication, synchronization of redundant process instances and so on. Associated to the stream mechanism, it permitted a lightweight and readable implementation of the message exchange on the network.

Then other Operating System dependencies were avoided through the GNAT OS_Lib interfaces.

4.2.2 Code quality

The GNATCheck tool was used to enforce the compliance of the code with the coding rules defined for the development process with EN50128 standard. Most of the coding rules were automatically verified using it.

4.2.3 Other utilities

Moreover several standard libraries were used to implement the middleware and participate also to keep the code clean and reduce its size. The XMLAda library is used to parse the application architecture configuration file, the Traceback and Source_Info libraries allow to report precise diagnostic traces and the MD5 tool is used to guarantee the integrity of the middleware version.

4.3 Difficulties encountered

After all the advantages exposed above, we also want to describe the difficulties and issues encountered during the development.

4.3.1 Development tool bugs

Several bugs showed up while using the GNAT Pro 7.2 version. They were related to the use of Ada 2012 features. Most of the bugs were compiler ones leading to crash bug box in various cases (expression function, type declaration with discriminant and derived type, anonymous type use, access to private record structure with discriminant and child unit), a few were on compiled code (non-evaluation of protected barrier, double execution of instruction) and on the GNATCheck tool (expression functions).

However all these problems have been fixed since the GNAT Pro 7.4 version.

4.3.2 Real time timers

The real time timers `Ada.Real_Time.Timing_Events` implementation is not suitable for the middleware. Indeed these timers are implemented using one task to check the expiration of all timers and execute the associated callback. Consequently, execution of one callback can postpone the following timer expiration. This last point was not acceptable since application code could have modified middleware behavior by delaying internal timers. Moreover the protected procedure callback interface was not convenient since it was forbidding to use the task-safe queues used for sending messages. Even if these timers could be suitable for embedded environment, it was not the case for the native environment and middleware needs.

As a consequence the timers were re-implemented for the middleware needs by providing non protected callback interfaces for timer expiration executed by independent tasks.

4.3.3 Performance issues

Stream sockets

Due to the combination of serialization and streaming through TCP sockets, we faced a performance issue for which the network performance was heading to 2 MBytes/sec with full CPU usage. Indeed for the serialized type, the smallest serializable sub-elements are sent as independent TCP messages which was not suitable to send our defined abstract message type. Moreover since concrete message types are defined by the application it is not suitable to let the application re-define the serialization of the type.

This problem has been solved in several steps. First we used a stream memory buffer implementation which is a stream type which can be streamed itself into another stream. Then, since it was implemented with a byte array, we had to redefine its serialization to avoid it to be sent element by element (default array serialization behavior except for the String type). Finally we used this stream as an intermediate stream to send a message type object to a TCP stream socket generating only one TCP message. This solution improved performance by a factor of 50 and brought back normal CPU usage.

Task-safe unbounded priority queues

Exchanges of messages in the middleware intensively use `Ada.Containers.Unbounded_Priority_Queues` but a performance issue was present in the implementation when using it with many messages with the same priority in the queue. Finally we participated to fix it in GNAT Pro (NF-17-OB05-042).

5 Conclusion

This middleware development was the first important industrial project using Ada 2012 for Systerel. It has convinced us that it is a major evolution of the language. The new Ada features allow for a quick development and finalization of a middleware containing a sensitive hot-redundancy feature. Finally, the first railway product application hosted by the middleware is now in production and is robust and efficient.

References

[1] ISO/IEC 8652: 2012(E), *Ada 2012 Reference Manual.*