Real–Time Teaching with Java: JPR^3 *

Diego Alonso and Juan A. Pastor and Bárbara Álvarez

diego.alonso@upct.es
Universidad Politécnica de Cartagena (Spain)

Abstract. This paper describes a development platform built around a digital railroad scale–model: JPR^3 (Java Platform for Realtime Railway Research). The laboratory equipment and software aims to achieve two goals: help and motivate students of real–time systems and as support for postgraduate students. Students find the scale–model really challenging and are very motivated by it; thus it's easy for them to really learn and practice all the concepts of real–time systems. But it's not only for students use: it also serves as a research platform for postgraduate students, thanks to the possibilities offered by the scale–model. Java has been chosen as the programming language codify the platform and the implementation of the system is described in this work.

1 Introduction

Although many undergraduate courses in computer engineering acquaint students with the fundamental topics in real–time computing, many do not provide adequate laboratory platforms to exercise the software skills necessary to build physical real–time systems. The undergraduate real–time systems course at Technical University of Cartagena is practical and carries out in a laboratory with a Digital Model Railroad Platform, where students can apply the real–time concepts explained at class [1] and see them work in a real enviorment.

Thanks to the Real–Time Extension [2], Java now offers a wonderful API for real–time systems' teaching, because there is a clear relation between real–time concepts and Java objects. Plus the fact that students can see it work in a real–time operating system, the result is a complete "real–time experience" for them. Also, Java has been chosen because students are already familiar to it, since Java is studied during the first courses. This way, they can focus on applying real–time techniques rather than in learning a new programming language.

But in the process of real–time learning, the railway scale–model is also of great help, not only because its a real, physical system, but also because it motivates the students, who find it very challenging. The total length of the track plus the complexity added by the possibility of using the turnouts, results

* This work was supported in part by the Spanish Ministry of Education (with reference ACF2000–0037–IN) and the Regional Government of Murcia (Séneca Programmes with reference PB/8/FS/02)

in a complex circuit to control in which problems with several levels of difficulty (and risk) can be simulated.

This paper is organised in five more sections. Section 2 gives a complete description of the laboratory equipment, both hardware (Sect. 2.1) and software (Sect. 2.2). In Sect. 3 the real–time problems related to the mock–up are outlined, and the Java implementation of the server–side of JPR^3 is described in Sect. 4. An example of a real practice is presented in Sect. 5. Finally, Sect. 6 summarised the content of the paper and outlines future plans for JPR^3.

2 The Platform at the Laboratory

This section presents a brief description of the equipment present at the laboratory, focusing on the railroad scale–model and the software control platform. The railroad scale–model has been developed around a commercial system designed and built by Märklin[3]. Specifically, it is based on the Märklin Digital System. Figure 1 shows a panoramic view of the railroad platform once finished.

Fig. 1. Overview photograph of the scale–model

On the other side, the control architecture is run on a normal Intel Pentium computer, placed next to the scale–model and connected to it by an RS–232 serial wire.

2.1 The Digital Model Railroad

Märklin commercialises beautiful all–time locomotives and all kind of accessories to simulate a real railway network. The railroad scale–model placed at the laboratory is formed by the following elements and accessories:

- ⋆ Five digital locomotives, capable of moving in both directions and with special functions, such as play the bell, turn–on the lights, or even throw smoke.
- ⋆ Sixteen digital turnout switches (where three tracks join) with manual and automatic control.
- ⋆ Six digital semaphores, which are only passive elements, i.e., the locomotive doesn't stop by itself if the semaphore is in red.
- ⋆ Twenty one double reed contact sensors to manage and control the position of the different locomotives in the mock–up.
- ⋆ Around a hundred railway tracks, both straight and curved, that make up our particular railway network.

The Märklin Digital System uses the tracks as power and control lines for all elements present in the scale–model, so the number of wires is minimum and new elements can be easily added. Moreover, it uses the centre of the tracks as the main conductor line, so the polarity of the signal is independent of the direction of the movement of the locomotives. But this kind of communication, based on friction, has a great drawback: it's very noisy. So the transmission speed is set to a very low value and every command is sent several times to ensure its correctly received.

All the active elements of the scale–model (turnouts, semaphores and locomotives) have a unique identification number and carry a device to decode the control commands that travel by the tracks. Because of the noise in the system, each decoder needs to decode, at least, three times the same command for it to proceed with it. This non-desired feature greatly increases the latency of the system.

The reed contacts are placed before and after every turnout around the mock–up, in order to monitor the traffic on the railroad. Each reed contact is really composed by two switches, which are activated depending on the direction of the locomotive that is stepping through it. To get the state of the reed contacts, three Märklin S88 Decoders are used. Each one provides a reading of the status of up to eight reed contacts, resulting in a total of sixteen sensors.

The scale–model can be manually operated by means of the Märklin 6021 Control Unit, the core of the Märklin Digital System. This module is in charge of both converting the control orders to electric signals, that are transmitted through the rails, and of reading the state of the Märklin S88 Decoders. Finally, to be able to control the scale–model with a computer, a module that provides a RS–232 serial interface with the Control Unit is used (see Fig. 2).

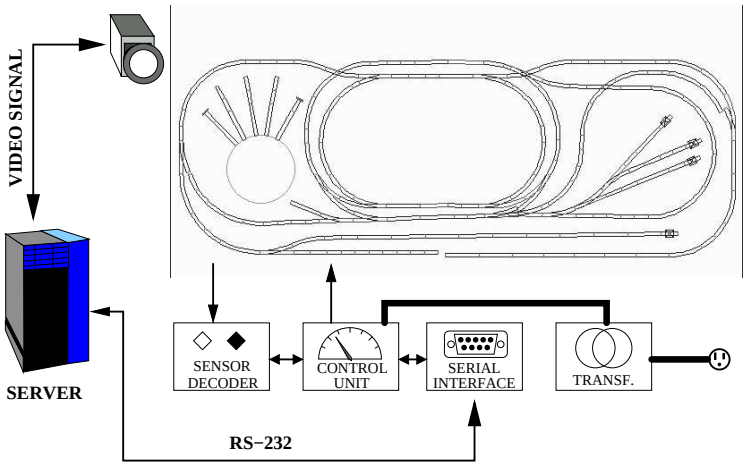

Fig. 2. Diagram of the configuration of the scale–model

2.2 The Software Platform: JPR^3

The software design of the JPR^3 was started following a four view design approach [4][5]. The initial development of the platform was guided by three objectives:

1. The application has to run in a host system that doesn't make use of the Real–Time Java extension. The user has to be able to configure whether the Real–Time API has to be used or not.
2. The architecture has to be modular and easily extendable, so new features could be added (such as the use of some video camera or a simulator of the mock–up).
3. The architecture *should* be distributed, so different clients (such as an automatic control module or a web client) could make use and monitor the mock–up.

With all these objectives in mind, the application was developed following the schema shown in Fig. 3. This paper presents only the, what is called, *server–side* of the application, which is, after all, the only that really has real–time constraints.

To control the elements of the mock–up, the Märklin Serial Interface provides several commands to send to the Control Unit. Section 3 presents all the issues related to the implementation of the communication with the scale–model, which, as we will see, is not trivial. The available implemented control commands are:

- ◇ Stop the mock–up
- ◇ Start the mock–up
- ◇ Change turnout track
- ◇ Read reed contacts
- ◇ Manage semaphores
- ◇ Use locomotives functions
- ◇ Change locomotive speed
- ◇ Change locomotive direction

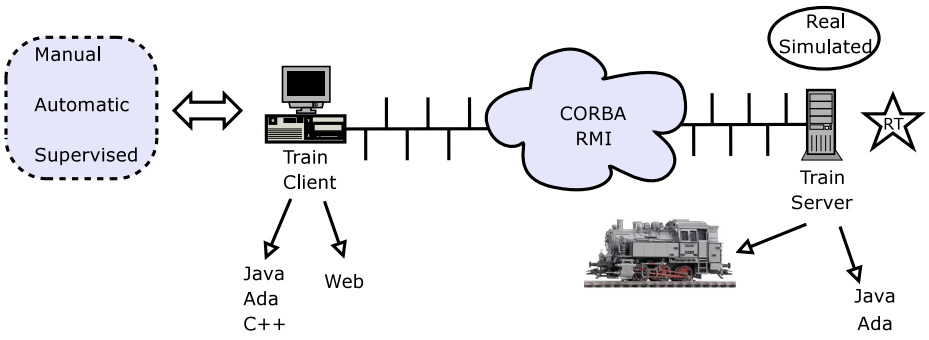

Fig. 3. Architecture overview

3 Real–Time Characterisation of the Scale–Model

The greatest time constraint imposed by the scale–model is due to the communication system. As said before, the Märklin Digital System provides a great advantage (there are practically no wires) but at a great cost (the communication is noisy, commands are sent several times and it works at a very low speed). Moreover, a small *unknown* delay has to be introduced between two consecutive commands, because, otherwise, the last command could be *lost in its way* and thus completely ignored by the Control Unit.

Although the available set of commands is reduced, as seen in last section, it is obvious that not all commands have the same priority. Commands such as *stop* and *start* have a greater priority over the rest. Indeed, the emergency-stop or stop-all order has to executed at fast as possible, to avoid collision between locomotives. Also the scale–model ignores all commands until the start one has been received, making the sending of other commands useless.

The other important time constraint is imposed by a simple fact: *locomotives do not have to crush!*. This desirable objective means in practice that there has to be a free track between two locomotives. In this case, the word *track* groups all tracks between two reed contacts. As said in Sect. 2, there's a reed contact sensor before and after every turnout element of the mock–up, and they are the only available source of information to know where a locomotive may be. We say *may be* to mean that we only know that some locomotive has stepped through one reed contact in a given direction, thus it is only known that the locomotive is somewhere over the track.

Having said that, the safety condition for the system is the following: the frequency of the reading of the state of the reed contacts has to ensure that no locomotive could have activated two different reed contacts between two consecutive readings. So, given the maximum locomotive speed and the shortest track

(group of tracks between two consecutive reed contacts) the minimum period between reads is around 500 msec.

4 Java Implementation Issues

Once the architecture of the application was finally defined and the server–side finished, it was time to test it. At that time, we chose the solution proposed by Timesys[6]: a modification of the Linux Kernel and an implementation of the Virtual Machine over it. So, by now we are using version LEK-X86BSP-4.1-155.3 of Timesys' Linux Kernel and version RTJ-X86BSP-1.0-547.1 of the Java Virtual Machine. Everything has been installed, as said before, in the Intel Pentium computer present at the laboratory.

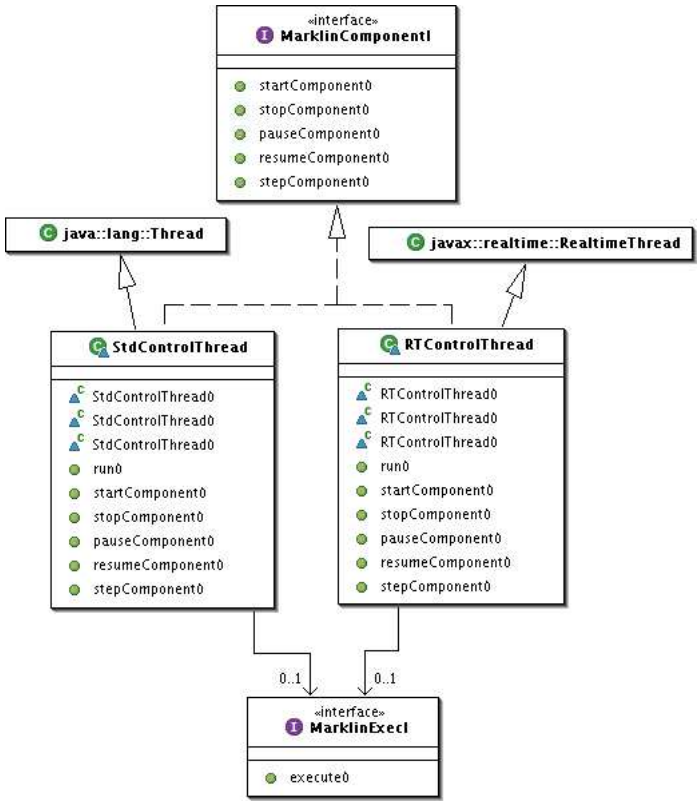

Fig. 4. Execution layer class diagram

As said in Sect. 2.2, the most important objective seeked by the development of the application was to make it as highly configurable as possible. The goal was

to be able to configure the kind of threads it would use (normal Java threads or RealtimeThreads [7]). This choice, as well as many more, are made during the application start–up by means of configuration files. Figure 4 shows an UML diagram of the execution layer. Among these configurable characteristics, two will be named:

- The use of the Java Real–Time extension.
- The time constraints of the different threads, such as the deadline, period, priority, ... Some values are obviously ignored when non real–time execution is selected.

When the initial creation and configuration step is over, every component shown in Fig. 3 runs in it's own thread (whether real or non real–time one). Each one of these threads supervises and controls every object and thread created in the component it's in charge of. This way, the platform can be easily and safely stopped.

Figure 5 shows the configuration of the component we have named "Mock–Up Interface". This component is the core of the server–side of the application, because is in charge of managing the communication between the mock–up and the different possible clients.

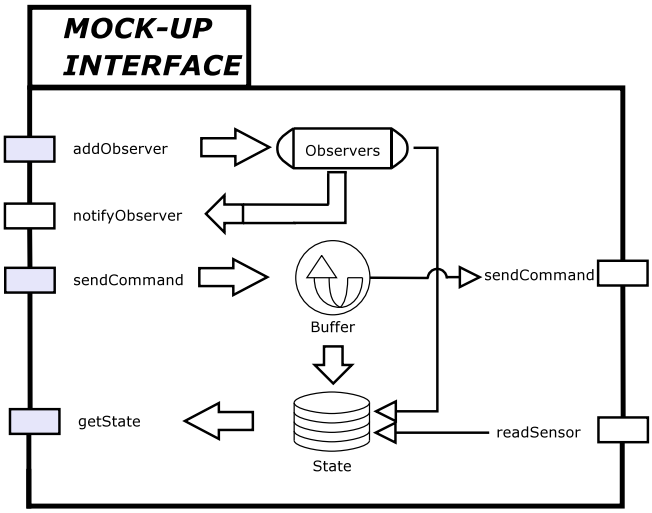

Fig. 5. Inner configuration of the mock–up interface

As can be seen in the figure, the constraint imposed by the latency of the communication is solved by the use of two circular buffer with different priorities. This was necessary to ensure that high–priority commands are executed nearly when they are send to the component. The application consider only two such commands: *emergency stop* and *release of the system*, whose importance can be

derived by their names. Besides the circular buffer, the server also follows an observer pattern [8] to notify every client of the state of mock–up.

When the initital configuration and creation of objects is finished, three threads are kept running in the server component:

- One thread, with the highest priority of all, to send commands to the mock–up and read the sensors.
- Another thread to notify the observers of the state of the mock–up that it has changed.
- One thread to execute different control strategies, developed by the students, when in automatic control mode.

5 A Practical Exercise

The laboratory has been used in courses such as Concurrent Programming and Real-Time Systems. These courses are placed in the last year of the Industrial Electronics and Control Engineer graduate curriculum, and they cover a good percent of ACM/IEEE-CS recommendations in computing technologies [9]. In particular, the course Real-Time Systems has seven lectures which concentrated on the following topics:

1. Characteristics of real–time systems and introduction to the Real–Time Specification for Java.
2. Concurrent programming.
3. Scheduling schemes (cyclic executive and priority–based models).
4. Reliability, fault tolerance and low–level programming.

The practical exercises must provide opportunities for these theoretical concepts testing. With the basic infrastructure described in the above sections, a highly comprehensive set of programming practices were developed. One of them is briefly described in this section, simply to give an overall idea of the laboratorys possibilities.

The railroad platform simulator focuses on modularity. Students should be able to design an application module using simulator module interfaces. An example of practical exercise is carried out by the students starting from the following requirements specification. Once the students know the railroad platform, we fix the turnout switches in order to have a lineal problem, as can be seen in Fig. 6.

It is necessary to avoid that a train collides with other train. Evidently, because of the turnout switches are fixed, we have a one–way railroad and it is not possible that a train meets to other train in a turnout switch or two trains collide. In order to avoid accidents, it has always to have a "free stretch"[1] between two trains. In accordance with this idea, if a train (`train A`) has gone over a reed contact sensor which represents the beginning of a track and other

[1] We define a "free stretch" as the space of tracks between two reed contacts

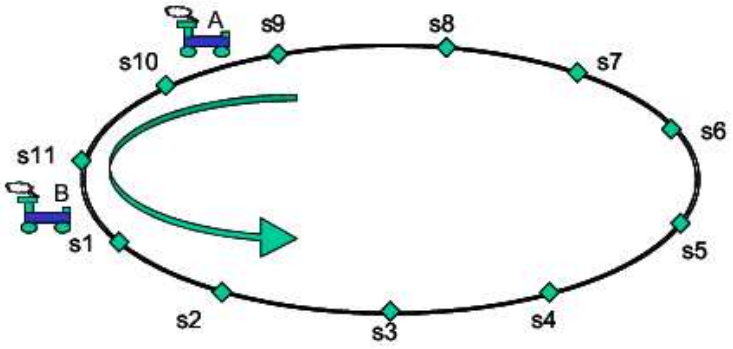

Fig. 6. A practical scheme

train (`train B`) is on the following track then we have to stop the first train (`train A`) until the second train (`train B`) leaves the occupied track. In this moment, the stopped train must recovery its initial speed.

As Fig. 6 describes, the `train A` is over a stretch between s9–s10 reed contacts and can go on towards the stretch delimited by s10–s11 reed contacts. However, this train must be stopped when standing on s11 reed contact if the `train B` has not stood on s1 reed contact.

To avoid the constant execution of stopping and starting procedures, it is recommended to increase the speed of the forward train (in our example `train A`) around 20% until having a "free stretch" between the trains again. From this moment, the forward train travels with its initial speed[2].

The students must implement a program for monitoring the state of the different sensors and modifying the speed of the trains in order to avoid collisions between them as the above specification describes. Each train must be implemented as RealtimeThreads. In this way, the control is distributed and other manager task will centralize the occupation of the tracks. Some real–time characteristics of the railroad platform are: critical safety operations (to stop a train, to stop the whole system, to change the state of a turnout switch, ...) need to have a very high priority; the time needed to stop a train must be bounded; the detection of a possible collision between trains must be in certain limits, and so on. In addition, the times of standing on each reed contact sensor must be registered.

Finally, the student must take into account possible fails and manage them using the mechanism of exceptions. In this way, the exercise allows to practice the different topics reviewed during the course as concurrent programming, scheduling schemes, fault tolerance, etc ...

[2] We suppose in this example that the students know the simulator and its interface to reading all sensorial information accurately

6 Conclusion and Future Work

JPR^3 aims to help students exercise their knowledge of concurrent programming, real–time systems, control application, etc ... as well as research tool for postgraduate students. The platform was designed to be easy to use and configure for students, but also extensible so new features could be added in a painless way.

Obviously, the research and development of JPR^3 does not end here. We've already started up a number of activities to extend its functionalities.emphasiseese, we wish to emphasize the following ones: design of a simulator of the mock–up (so students could test their control strategies without using the real mock–up), use of video cameras to carry out visual supervision and control of the locomotives (cameras can also be placed *inside* the locomotives) and, finally, make it a real distributed application by using a communication middleware, such as CORBA and/or RMI.

Although we have chosen a particular implementation of the Virtual Machine, we also plan to test JPR^3 with other implementations of the JVM and real–time operating systems.

References

1. Burns, A. Wellings, A.: *"Real–Time Systems and Programming Languages"*. Addison–Wesley (2001)
2. Real–Time Specification for Java v1.0 *http://www.rtj.org/rtsj-V1.0.pdf*
3. Märklin Trains homepage *http://www.marklin.com*
4. Hofmeister, C. et als.: *"Applied Software Architecture"*. Addison–Wesley (2000)
5. Powel, B.: *"Real–Time Design Patterns"*. Addison–Wesley (2003)
6. TimeSys Corporation homepage *http://www.timesys.com*
7. Dibble, P.: *"Real–Time Java Platform Programming"*. Prentice Hall (2002)
8. Gamma, E. et als.: *"Design Patterns: Elements of Reusable Object Oriented Software"*. Addison–Wesley (1995)
9. CM and IEEE-CS. Year 2001 model curricula for computing (Computing curricula 2001). The joint IEEE computer society/ACM task force. [Online]. Available: http://www.acm.org/education/curricula.html.