The Design of a Real-Time Event Manager Component

Damien Masson
Université Paris-Est
LIGM, UMR CNRS 8049
ESIEE Paris
2 boulevard Blaise Pascal
BP 99 93162 Noisy-le-Grand CEDEX
d.masson@esiee.fr

Serge Midonnet
Université Paris-Est
LIGM, UMR CNRS 8049
Université Paris-Est Marne-la-Vallée
5 boulevard Descartes
77454 Marne-la-Vallée CEDEX 2
serge.midonnet@univ-paris-est.fr

***Abstract*—We review the issues with the conception of real-time event based applications and propose an event manager component design. We start from the design proposed by the Real-time Specification for Java to handle asynchronous events and we extend and generalize it to form an event manager component. This component manages the relations between the triggering of distributed events, the events handlers and the resource allocations (servers and processors). The allocation process consists of setting individual event handlers to servers and servers to processors. The component uses the description of the events relations to generate a state automaton used to release their handlers. The feasibility analysis is processed at a component level and takes into account the events relations and the association of the servers with the processors (affinity).**

I. Introduction

The aim of this paper is the design of an event based real-time application in a high level programming language. Indeed, the idea which was taken as a joke ten years ago to program real-time application in Java is becoming a reality nowadays. The Java community requested a real-time specification in 1996 and a first release of the RTSJ was published in 2000. This first version specifies a set of classes to represent the scheduler, the real-time tasks and handlers of asynchronous events, the time and timers, memory regions and monitors to control shared resource accesses.

A new Specification Request (JSR-282) was recently presented in order to specify a new release (version 1.1) for RTSJ, and a new revision of the Reference Implementation (RI-1.1 alpha6) is provided by Timesys. This release improves the processing of asynchronous events (data can be associated with an event triggering) and describes the affinity of schedulable entities and processors in multiprocessors environments.

In a previous work [1], [2], we focused on the implementation within the RTSJ of advanced mechanisms. We want here to go further with the design of an high level software component which homogenize different solutions for the handling of asynchronous event in real-time systems.

The component approach allows to encapsulate mixed and complex mechanisms such as the inter events relation expressions, resources allocation, feasibility analysis or system dimensioning.

This paper presents the existing approach within the RTSJ, defines a design for a component we call the *Event Manager* and proposes configurable CORBA-idl interfaces to implement it. All pieces of the component we propose are implemented as part of different work. The aim of this component is so to make them run together.

Section 2 presents the existing model of the RTSJ to represent asynchronous event, its limitations, and the need to at least another entity: the task server. Section 3 introduces our *EventManager* and Section 4 formalizes it. We conclude and present future works in Section 5.

II. Component Design and Abstractions

We introduce in this section the design we propose to manage real-time events. We start by the model proposed in the RTSJ and we explain the need for an higher abstraction model.

A. The RTSJ Model

In order to generalize the `Thread` facility of regular Java programs, RTSJ proposes the notion of schedulable object (SO) through the interface `Schedulable`. A SO is an object which can be scheduled by a scheduler. A suitable context for the execution of such an object is created using scheduling and release parameters attached to each SO. `Schedulable` is an interface with getter and setter methods to access and modify scheduling parameters (priority and/or importance) (class `SchedulingParameters`) and release parameters (cost , period, deadline) (class `ReleaseParameters`). There is also methods to attach the SO to a particular scheduler or to add or remove it to the feasibility analysis.

Two main concrete classes implementing this interface are proposed: `RealtimeThread` which also extends the class `java.lang.Thread`, and `AsyncEventHandler` (AEH) which represents the logic to execute when an associated event, modeled with the class `AsyncEvent` (AE) is triggered. A `RealtimeThread` is always associated with one and only one thread, but `AsyncEventHandlers` can share a thread if a thread pool mechanism is implemented (this is a possibility offered by the specification but this is not mandatory).

We focus this work on the asynchronous events and their handlers which is the RTSJ model to implement event based applications. RTSJ proposes a many-to-many relation $(n-n)$ between events and handlers. Practically, this means that it is possible to associate to an event trigger an unbounded number of handlers and that a given handler can be bounded to several events.

1) AsyncEvent (AE): This class model an asynchronous event. No release parameters are directly associated with an event, but a method `createReleaseParameters()` permits to obtain a suitable instance of `ReleaseParameters` in order to associate it to an handler. The default behavior is to return an `AperiodicParameters`.

Methods `add/removeHandler()` permit respectively to add and remove an AEH to the list of associated handlers. The method `fire()` triggers the event. In practice, it means that all associated handlers are released and start to compete for execution. Methods `bindTo()` and `unbindTo()` permit to associate (or disassociate) the event to a kernel-level interruption like a key press, in order to automatize the call of the `fire()` method.

2) AsyncEventHandler (AEH): An AEH is a SO modeling the logic associated with an event triggering. It so implements the methods described in the `Schedulable` interface used to set and get the SO real-time properties:

- **job scheduling** The `SchedulingParameters` subclass `PriorityParameters` is used to set the priority and the importance of the object. Only fixed priority preemptive scheduling policies are required by the RTSJ.
- **job activation** The RTSJ proposes three `ReleaseParameters` subclasses to describe the activation model of a SO: `PeriodicParameters`, `AperiodicParameters` and `SporadicParameters`. With the sporadic model, one can set up the minimum inter arrival time (MIT) between two instances of a task. Four policies are possible to handle a MIT violation between two event triggering: *except* which throws an exception, *replace* which replaces the first happening by the new one, *ignore* which ignores the new one and *save* which enqueues the new arrivals.
- **memory region** The RTSJ proposes memory areas outside the classic Java heap, which are never garbage collected. A SO is associated with a memory region (the heap, physical memory or a scoped memory). We do not study this aspect in this paper.
- **scheduling policy** A SO is associated with an unique object `Scheduler`. An instance of `Scheduler` manages the execution of SOs and implements a feasibility algorithm.
- **feasibility analysis** The other methods concern operations used to analyze the schedulability of the set of SO associated this the same instance of `Scheduler`.

B. Advance Event Handling: the Event Server

Reducing all non periodic traffic to sporadic traffic is too limited. To handle pure aperiodic traffic in real-time system, three solutions exist. The first on is the background scheduling (BS): lowest priorities are reserved for the non periodic traffic. This solution is easily implementable with the AE/AEH model. However, it does not permit to obtain good mean response times for aperiodic handlers. The second solution, the use of *task servers*, is introduced in [3] response times. A task server is a task with a limited budget and a policy to replenish this budget. The interference of this task has to be known or at least bounded. Then the aperiodic traffic handling is delegated to it. The AE/AEH model is not sufficient to implement task servers. It is possible within the RTSJ to assign common release parameters to a group of SOs. These special release parameters are modeled by the class `ProcessingGroupParameters` (PGP). Unfortunately this behavior is underspecified [4]. A contribution to extend the PGP semantic has been led in [5] but this approach relies on modifications on the specification.

The third solution is the use of a slack stealer. The slack stealer rely on an algorithm to compute the slack: the maximal amount of time available at instant t to execute aperiodic tasks at the highest priority without endangering the periodic tasks. It is a special kind of server which use the slack as its capacity. As the time complexity of slack computation algorithms is too high, approximation algorithms are proposed [6]. Modified algorithms and a set of classes to write task servers and slack stealer with RTSJ is proposed in [1].

We introduce here the *EventServer* (ES) entity which regroups both task server and slack stealer approaches. The model to handle aperiodic request is now composed by three entities: AE/AEH/ES.

C. Toward an Event Manager

The aim of this paper is to propose a higher abstraction level to design event handling. Indeed, the ES is not sufficient to model more complex relations between events triggering and handler releases. We want to design a software component able to endorse the responsibility of the event gesture from its happening to the execution of its handler within a server running on a processor. Such an entity can be parameterized by choosing between policy allocations for events on handlers, and for handlers on processors. The first gain for the developer is the ability to use those high level mechanisms together without having to acquire an expertize for all of them. One other utility of a centralization beside a manager is to simplify the end-to-end time-analysis.

III. The Event Manager Component

We describe in this section a new software component we call an *Event Manager*. This component takes as an input the events and as an output the tasks assigned to processors. It endorses four responsibilities: managing the relations between events triggering and handlers releases, servicing the handlers in assigning them to task servers, allocating the servers on a processor set, and performing a feasibility analysis of the system.

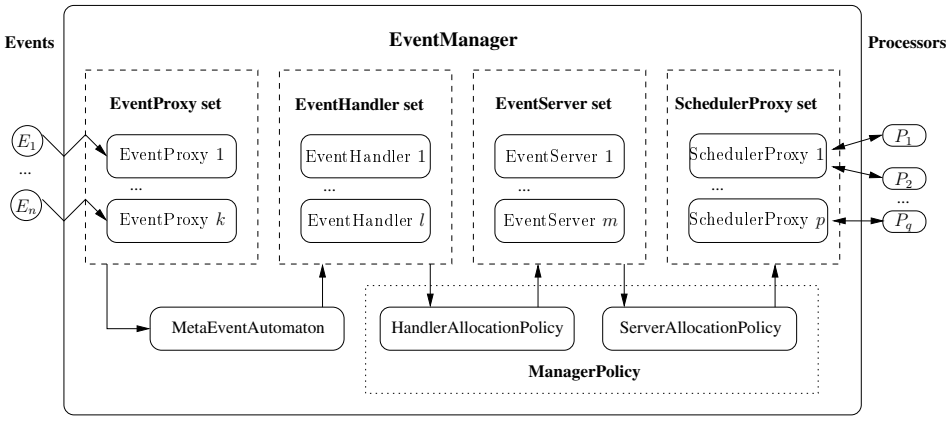


Fig. 1. Distributed-event manager on multiprocessor architectures

A. Relations Between Events and Handlers

In the model presented in the last section, there is no entity aware of the relations between AE, AEH and servers. However, when several AEH are bounded to the same AE, their release model will be the same. And when several handlers are assigned to the same task server, their interference on the other tasks in the system are linked. The first gain having a centralized manager is so to keep knowledge of these relations and to use it for the analysis.

Moreover, with a centralized entity, we can define more complex relations between events and handlers. Such an entity allows us to manage the handlers in a more powerful way than the basic association $n - n$ of the RTSJ model. We can for example imagine a system where a treatments should start only if two distinct events have occurred. Even more complex relations between the events happening can be described using logical operators. The manager can use an internal automaton in which an event happening corresponds to a transition. When the automaton reaches certain states, handlers are released. The logical operators and a language to produce the automaton is described in [7] and we will detail them in Section IV-F.

We can note that this model applies with both asynchronous events and sporadic/periodic events. The entity *EventProxy* designates the manager view of the event. *EventHandler* designates their handlers and *MetaEventAutomaton* the automaton.

B. Handlers Service

When the automaton reaches a state which corresponds to the release of an handler, this handler must be added to a server. According to a changeable policy, the manager will choose a server from a pool, or start a new server, and add the handler to its queue. The handler has release and scheduling parameters in order to permit the policy to choose a server, and the server to serve it. A special server can be used to scheduled sporadic tasks as soon as they are released. The servers are modeled by the entity *EventServer* and the policy by *HandlerAllocationPolicy*.

C. Processors Allocation

If the application is targeted for multiprocessors architectures and/or distributed environments, a processor must be choose to execute the server. The processors available can be viewed as a resource allocated to the event manager. We can distinguish two kinds of processor: some are reserved for the manager, others are shared with others SO or others event managers.

The theory of real-time multiprocessor scheduling is divided in two main approaches: global scheduling and partitioned scheduling. In the first one, the tasks are allowed to migrate from a processor to another at runtime. Whereas in the last one, the tasks are assigned statically to an unique processor. Once assigned to a processor, each processor schedules its own tasks set independently.

The processors are accessed by the manager through the *SchedulerProxy* entity. A scheduler proxy can be bounded to a scheduler which control one or several processors. The first case corresponds to a global scheduler and the second either to a local scheduler of a partitioned-based scheduled system or to a mono processor system.

D. Feasibility Analysis

The manager must provide interfaces to statute on the feasibility of the real-time constraints on the handlers and the servers. There is no entity responsible for this role, it is assumed by each component of the manager.

If a *SchedulerProxy* represents a scheduler reserved for the manager, well known results on feasibility analysis can be reused to decide the feasibility of the server set. However, if the scheduler is shared with other SO, the feasibility can no longer be analyzed by the component. For that purpose, the component must provide a bound on the interference the servers it serve can produce on other SO.

The automaton, which models the dependency relations between the events, can be used to transform the model in an equivalent – in the sense of the feasibility – of the starting system.

In our model, handlers have release and scheduling parameters used to choose the server where it will be executed. Some handlers can have deadline, in that case an admission control has to be performed before choosing the server. To choose the server among the ones which pass the admission control, a policy has to be chosen.

As resumed by Figure 1, a manager (EM) is so composed by a pool of servers (ES), a set of proxies which represent events (EP), a set of handlers (EH), a set of scheduler proxies which represent schedulers (SP), an automaton which describes the relation between EP set and EH set and a policy to assigned EH to ES and ES to SP.

IV. Formalizing the Event Manager

We have introduced all the elements of our component: the event manager. We will now detail and formalize them using interface declaration, and discuss about the analysis of a system composed with one or several event managers.

A. EventManager (EM)

The *EventManager* interface has methods to add and remove elements in each set and getter and setter for the automaton and the policies.

For the sake of clarity, we do not detail the remove methods corresponding to add ones and getters corresponding to setters.

```
public interface EventManager{
 void addEventProxy();
 void addEventServer();
 void addEventHandler();
 void addEventServer();
 void addSchedulingLocation(boolean reserved);
 void setMetaEventAutomaton(MetaEventAutomaton
      e, Set<EventHandler> handlers);
 void setMetaEventAutomaton(String
     metaEventDescription, Set<EventHandler>
     handlers);
 void setManagerPolicy(ManagerPolicy mp);
 void setManagerPolicy(HandlerAllocationPolicy
      hap, ServerAllocationpolicy sap);
}
```

B. EventProxy (EP)

An *EventProxy* is the manager view of an event. It can be the local event itself if the event is local, or associated to a distant event.

Methods `bind/unbindTo()` permit to connect the proxy with a physical event on the local system or to a distant event. Method `fire()` triggers the event. Note that there is no method to add or remove handlers since this association is made by the `MetaEventAutomaton` component of the manager.

```
public interface EventProxy{
 void fire();
 void bindTo(String happening);
}
```

C. EventHandler (EH)

An EH is the most simple entity, it represents a logic associated to the EM. It inherits from the interface *Servable* which represents an object which can be handle by a server, by opposition with a Schedulable (SO). This logic must be wrote in the method *handleEvent()*. When added to a manager the EH is associated with one server with respect to the required *ManagerPolicy*. Then EH is enqueued in the server queue when the automaton reaches the appropriate state.

```
public interface EventHandler::Servable{
 void handleEvent();
}
public interface Servable{
 void setReleaseParameters(ReleaseParameters
     rp);
 void setSchedulingParameters(
     SchedulingParameters sp);
}
```

D. EventServer (ES)

An ES is a SO in the RTSJ sense. It maintains a queue of servable which can be sorted by arrival dates (FIFO). It is caracteryzed by a *ServicePolicy* (Deferrable, Polling, Slack Stealer...). Note that a servable could have release and scheduling parameters, but this is not mandatory. If it has such parameters, the queue can also be sorted by priorities or deadlines... This depends on a *QueuePolicy* which can be (but not limited to) *FixedPriority* (FP) or *Deadline Monotonic* (DM). When the deadline of a servable is reached, its priority can be increased or decreased (a max priority value must have been computed to ensure the feasibility), the job can be dropped... This depends on a *DeadlineMissedPolicy*. If its WCET is consumed but its execution not completed, the server can cancel it, or let it continue, this depends on a *CostOverrunPolicy*. Finally, when its period (or minimal inter-arrival time) is violated, a *MITViolationPolicy* (like the one of the RTSJ) can be applied. A server also has to provide a method to let know if a given SO can respect its release parameters. We will see in Section IV-G that it can be used by the *EventHandlerAllocationPolicy* to choose a server to execute a given handler.

```
public interface EventServer{
 void addEventHandler(eventHandler eh);
 void setServicePolicy(ServicePolicy sp);
 void setMITViolationPolicy(MITViolationPolicy
      mvp);
 void setDeadlineMissedPolicy(
     DeadlineMissedPolicy dmp);
 void setCostOverrunPolicy(CostOverrunPolicy
     cop);
 void setQueuePolicy(QueuePolicy qp);
 boolean isAdmissible(Schedulable s);
}
enum ServicePolicy{
 POLLING,DEFERRABLE,SLACK_STEALER;
}
enum QueuePolicy{
 FIFO,DM,FP;
}
```

```
enum DeadlineMissedPolicy{
 DROP, INCREASE, DECREASE, IGNORE;
}
enum CostOverrunPolicy{
 DROP, INCREASE, DECREASE, IGNORE;
}
enum MITViolationPolicy{
 REPLACE, IGNORE, ADD;
}
```

E. SchedulerProxy (SP)

A *SchedulerProxy* is the manager view of a scheduler, which can be either a mono processor scheduler, a scheduler responsible of one processor on a partitioned-based multiprocessor architecture, or a global scheduler. An instance of SP manages the execution of servers on the scheduler it represents, according to a scheduling policy. It also implements a local feasibility algorithm. The feasibility algorithm determines if the known set of server is a feasible system. This feasibility analysis will also depends on the scheduling policy of the SP. The scheduler represented by the SP can be reserved for the manager, but it can also be shared with others SO or others managers. Then the method *isFeasible()* only concern the set of server, and an additional method *getMaxInterference()* is needed to permit the feasibility analysis.

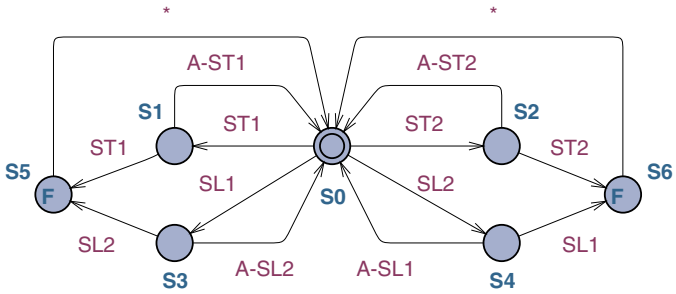
```
public interface SchedulingLocation{
 void addEventServer(EventServer es);
 boolean isFeasible();
 Interference getMaxInterference();
 SchedulingPolicy setSchedulingPolicy();
 boolean isReserved();
}
enum SchedulingPolicy{
 RM, DM, EDF;
}
public class Interference{
 RelativeTime period;
 RelativeTime wcet;
}
```

F. MetaEventAutomaton (MEA)

The MEA is the automaton which controls an internal state that records all past fired event. The handler is activated only when the conditions described in the meta-event description are satisfied.

The relations between the event proxies are described using a logical expression. This expression uses a set of logical operators (AND, OR, THEN, NEXT, NOT, TIMES). The automaton is produced either using a description language (*.acf) and then compiled (*rtmec*) with a *compil-type* argument set to *local* for single node multiprocessor and set to *distributed* for distributed multiprocessor manager[1] or using the *EventManager* as a MEA factory to create a MEA corresponding to the expression *expr* (see the *EventManager* idl description). Figure 2 presents a *MetaEvent* example.

[1]java -jar rtmec.jar -compil-type={local,distributed} description.acf




```
MetaEvent UrgencyStop{
SensorLight:SL1, SL2,
SensorTouch:ST1, ST2
EventCondition ((SL1 NEXT SL2) OR
(SL2 NEXT SL1) OR
(ST1 TIMES 2) OR (ST2 TIMES 2));
};
```

Fig. 2. Urgency Stop Automaton Example: In this example the *MetaEvent* describes the relations between four events in a control application for a robot. The *SensorTouch* events and *SensorLights* relations indicate the mandatory stop condition for the robot.

G. ManagerPolicy (MP)

The MP has two distinct roles: to manage the assignments of the handlers on the servers, and to manage the assignments of the ES on the SL. The first is handled by an *HandlerAllocationPolicy* (HAP) and the second by a *ServerAllocationPolicy* (SAP).

```
public interface MetaSchedulingPolicy{
 void setHandlerAllocationPolicy(
     HandlerAllocationPolicy hap);
 void setServerAllocationPolicy(
     ServerAllocationPolicy sap);
}
```

1) HandlerAllocationPolicy (HAP): There is several possibilities to assign the handlers on the servers. We can select the appropriate server because of its policy, of its priority, of its current charge, of the response time it can warranty at instant t, ... The role of an HAP is to choose from the set of server in the manager the one to serve a given handler, or to start a new server. The admission control method in the interface *EventServer* can be used to arbitrate the decision. The pool of server can also be viewed as a given limited resource, and use in a similar way than a pool of processors in multiprocessor scheduling theory. Then several policy can be applied. The placement issue can then be reduced to the *Bin-Packing* problem which has been largely studied. Many heuristics exist to solve it [8]. The most classical we can cite are *First-Fit*, and *Worst-Fit*. *First-Fit* places a new object in the leftmost bin that still has room while *Worst-Fit* places a new object in the emptiest existing bin. First-Fit reduces the number of used servers and Worst-Fit reduces the load of each server (it uses all the server set).

```
public interface EventServerPolicy{
 AllocationPolicy getPolicy();
```

```
  EventServer assignEventHandler(EventHandler
      eh);
}
enum AllocationPolicy{
  FirstFit ,WorstFit;
}
```

2) ServerAllocationPolicy (SAP): The role of the SAP is to choose a scheduler proxy for a given server. Then the server will be scheduled according to the scheduling policy of the proxy. There is several way to choose the proxy: again, we face a *Bin-Packing* problem, and well known heuristic can be proposed: *First-Fit* and *Worst-Fit*.

```
public interface SchedulingLocationPolicy{
  AllocationPolicy getPolicy();
  SchedulerProxy assignEventServer(EventServer
      eh);
}
```

V. Conclusions and Future works

We have presented the existing model of the RTSJ to represent asynchronous events, its limitations, and the need to extend it to allows the programming of a higher level handling of event. We had justify the need and the advantages of setting a centralized entity, the *EventManager*, which has a global view of the system from the event happening to the release on a processor of its handlers within a server. We proposed interfaces to implement this manager and justified our choices. It is important to note that each part of component proposed in the paper is implementable and implemented, except the global scheduling which is a work in progress.

We also want to propose, as soon as possible, all these implementations in a well designed package with runnable examples.

This will permits us to conduct extensive measures in order to validate the usability of our software component for hard real-time applications.

As part of future work, we want to study the possibility to introduce temporal behavior in the meta event automaton. This could allow us to deal with release parameters violation (like the minimal inter-arrival time) before the release of the handler, which could enhanced the performance of the component.

References

[1] D. Masson and S. Midonnet, "RTSJ Extensions: Event Manager and Feasibility Analyzer," in *6th Java Technologies for Real-Time and Embedded Systems (JTRES'08)*, ser. ACM International Conference Proceeding, vol. 343, Santa Clara, California, Sep. 2008, pp. 10–18, (9 pp.).

[2] ——, "Userland Approximate Slack Stealer with Low Time Complexity," in *16th Real-Time and Network Systems (RTNS'08)*, Rennes, France, Oct. 2008, pp. 29–38, (10 pp.).

[3] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proceedings of the Real-Time Systems Symposium*. San jose, California: IEEE Computer Society, Dec. 1987, pp. 110–123.

[4] A. Burns and A. Wellings, "Processing group parameters in the real-time specification for java," in *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, vol. LNCS 2889. Springer, 2003, pp. 360–370. [Online]. Available: http://www.cs.york.ac.uk/rts/cgi-bin/bibtex/bibtex.pl?key=R:Burns:2003g

[5] A. Wellings and M. Kim, "Processing group parameters in the real-time specification for java," in *proceedings of JTRES 2008*, 2008.

[6] R. I. Davis, K. Tindell, and A. Burns, "Scheduling slack time in fixed priority pre-emptive systems," in *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93)*, 1993, pp. 222–231.

[7] C. Curcio and S. Midonnet, "Resource Reclamation using Meta-Events in a Real Time Java System," in *Poster session of 3rd International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS'07)*. Hsinchu, Taiwan: IEEE Press, Dec. 2007, electronic version (6 pp.).

[8] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," pp. 46–93, 1997.