The Analysis and Evaluation of Design Patterns
for Distributed Real-Time Java Software

Angelo Corsaro
Strategic and Technological Planning
SELEX SI
Via Tiburtina Km. 12.400
00131 Rome, Italy
Email: acorsaro@selex-si.com

Corrado Santoro
Department of Computer and
Telecommunication Engineering
University of Catania
Viale A. Doria, 6 - 95125 - Catania, Italy
EMail: csanto@diit.unict.it

Abstract

The Real-time Specification for Java (RTSJ) introduces a new memory model featuring some programming constraints that impede the “as-is” use of many well-known design patterns. In this context, this paper describes and evaluates two design patterns, developed by the authors for distributed real-time Java software, that are able to overcome the limitations imposed by RTSJ. The first pattern, RTJ-Leader-Follower, is a RTSJ-compliant version of the well-known Leader/Follower pattern. The second pattern, called Scoped Tunnels, provides a new communication mechanism for RTSJ threads executing in different and incompatible memory areas, thus making possible the realization of the standard pattern half-sync/half-async for efficient network I/O handling. The paper presents both a qualitative and quantitative evaluation of these patterns, showing, above all, that they are able to provide a safe execution environment.

1 Introduction

The development of distributed real-time and embedded (DRE) systems using the Java language is becoming a reality [5, 22]. The Real-Time Specification for Java (RTSJ) [3], proposed to face this challenging scenario, provides a set of classes intended to facilitate the development of DRE systems in Java; moreover, some RTSJ-compliant virtual machines are now available [1, 21, 20, 15, 10]. However, despite this availability of tools, RTSJ introduces some programming constraints that make the development of Java DRE systems a non-trivial activity. In fact, in order to avoid unpredictable latencies introduced by the garbage collector, RTSJ’s memory management is based on a model that makes use of non-heap memory areas—the so-called scoped memories—featuring a predictable object allocation time; such areas do not require tracing garbage collection since objects allocated in a “scope” are automatically deleted when the last thread exits the scope itself. To support this kind of structures, some limitations on object references are introduced, thus making hard, or even infeasible, the implementation of several well-known design patterns for middleware and network software [12, 17, 16]. This paper analyzes the impact of the RTSJ programming model on several patterns typically used in the design and implementation of distributed middleware and applications, and proposes two RTSJ-compliant solutions. The papers refers to two well-known design patterns, the leader-follower [17] and half-sync/half-async [17], which are the most often used in the design of network servers [17]. The paper states the problem, explaining the issues that make infeasible the cited patterns in RTSJ, and proposes some qualitative evaluation criteria. Then two RTSJ-compliant solutions are provided, called rtj-leader-follower and Scoped Tunnels. Such solutions are then evaluated using the derived criteria and a performance analysis, both showing the effectiveness of the proposed patterns for the realization of RTSJ-based DRE systems.

The paper is structured as follows. Section 2 provides an overview of the leader-follower and half-sync/half-async patterns, showing the reasons why they are infeasible with RTSJ; some parameters able to evaluate a RTSJ-compliant solution are also introduced. Section 3 and 4 present and evaluate the patterns rtj-leader-follower and Scoped Tunnels for half-sync/half-async, respectively. Section 5 reports some related work. Finally, Section 6 concludes the paper.

2 Background

2.1 Leader-Follower

The leader-follower [17] is a well-known design pattern used to efficiently handle concurrent events coming from various sources. In the design of network
software, this pattern is used to receive incoming requests, and suitably processing them\(^1\).

The basic version of this pattern, which is sketched in Figure 1, faces the inefficiency caused by a on-the-fly creation of a new thread for each new event to be handled, by using a set of threads, created at programm startup and placed in a thread pool. The functioning is based on (1) picking a leader thread from the thread pool, and selecting it to handle an incoming event; when (2) the event is captured, the leader (3) elects a follower thread to become the new leader, and then (4) properly handles the event. In this way, during processing of an event, another (new) event can be concurrently detected and handled.

A possible Java implementation of such pattern foresees to start all the threads of the pool in advance and making these to run an infinite loop aiming at (i) suspending, waiting for being selected as leader; (ii) waiting for the event; (iii) signaling the thread pool to promote a new leader; (iv) passing the event to the handler; and (v) suspending again.

Using this pattern in RTSJ world would imply to run threads as (NoHeap)RealtimeThread objects, according to the urgency of the event to be handled, and thus associate each thread with a scoped memory area. However, in this case, threads would run the infinite loop in a scoped memory; therefore, since threads will never end (exiting the scope), the objects created during event processing (and allocated in the scope) will be never collected.

2.2 Half-Sync/Half-Async

The Half-Sync/Half-Async is an architectural pattern [17] used to decouple synchronous operations from asynchronous ones in a concurrent I/O system. The pattern, which is sketched in Figure 2, is based on an asynchronous process that, triggered by the occurrence of an event\(^2\) (1a in Figure 2), handles it reading the associated data and dispatches the latter (2a) to a synchronous (user) process that has previously issued a blocking request for new data (1b); once data is dispatched, the synchronous process can thus obtain the information (2b) and suitably process them. This pattern, very common in network I/O software, is based on decoupling synchronous and asynchronous sides by means of a queue filled by the asynchronous process and emptied by the synchronous one.

In a real-time environment, events and data must be handled and processed according to their urgency, thus the execution of the synchronous and asynchronous processes must be performed according to proper deadlines. Once again, this means that, in a RTSJ environment, the interacting processes must be instances of (NoHeap)RealtimeThread classes and thus associated with suitable scoped memory regions. If the system is designed such that threads belong to the same memory region, the queue can be implemented using an object pool approach [4, 2, 16]; so, in this case, the pattern is still feasible in RTSJ. However, placing the threads in the same memory region is a constraint too strong for the design of a concurrent system; the general case is instead the one in which threads belong to different (and often incompatible) memory regions: in this case, communication of objects between these domains cannot be performed, since it would soon cause a violation of the RTSJ memory model constraints. Indeed, the main concern is that any object exchanged in the queue is created in the context of a thread (the asynchronous one) and must be reclaimed, after its usage, in the context of another thread (the synchronous one). And this is not possible with RTSJ.

2.3 Design and Evaluation Criteria

As reported above, the design patterns illustrated need to be suitably modified in order to be feasible in a RTSJ environment. In facing such a refactoring, any design solution should take into particular account not only the traditional design criteria of object-oriented software engineering [12, 17, 19] but also some other issues specifically related to RTSJ and real-time [5, 22]. To this aim, we derived some criteria that design solutions should obey and that can be summarized in the following objectives:

O.1 Correctness. The solution must be able to handle data with different criticality, according to the associated time constraints.

---

1 This pattern is used in many well-known network servers, such as CORBA implementations [14] or the Apache Web Server, etc.

2 e.g. a socket connection, a new message from the network, etc.
O.2 Compliance. The solution must be compliant to RTSJ and in particular to its memory model.

O.3 Safeness. The solution must avoid memory leaks and uncontrolled growth of allocated memory.

O.4 Efficiency. When concurrency is a requirement, the solution must be effective and efficient in that it must provide a high degree of parallelism and reduce latencies and context switches.

O.5 Transparency. The solution must provide a programming environment that allows the designer to deal with the original Java semantics (i.e., automatic memory management) and ignore RTSJ memory management details.

3 The RTJ-Leader-Follower

This Section deals with rtj-leader-follower, our proposal to make the leader-follower pattern RTSJ compliant. Given that the pattern was first presented in [7], we give here a description of the main design choices, followed by an evaluation of its performances and adherence to objectives listed in Section 2.3.

Context. The same of the traditional leader-follower [17], given that events are time-critical and their handling must be performed according to the associated time constraints.

Problem. The problem to solve is to handle time-critical events, using RTSJ software, taking in particular into account the objectives reported in Section 2.3.

Solution. The main issue to be solved to make event handling possible, according to the principles of the leader-follower pattern, is the organization of threads and scoped memories, in order to avoid any violation of RTSJ constraints (objective O.2) and the occurrence of memory leaks (objective O.3). According to these concepts, event handling must be performed inside a scope that must be entered by the handler before performing processing and exited when handling is due: this ensure that all objects, created during event handling, are automatically reclaimed at the end of processing operations. Each thread of the pool must thus be associated with a scoped memory area—we call it handler scope—to be used for the purpose above. However, such an association does not suffice to meet all the objectives. In fact, according to the principles of leader-follower, the threads of the pool are all created at startup time, and each of them runs an infinite loop. In this scenario, only the event handling operation must be executed inside the handler scope, thus requiring a structures that decouples the thread from event handling, not only from the operational point of view but also for scoped memory association.

Structure. On the basis of the argumentation above, the schema proposed for the rtj-leader-follower, which is depicted in Figure 3, is based on using, for each element of the pool, three main objects: the EventHandlerLauncher, the HandlerScope and Event Handler, all allocated at startup time in a scoped memory area, called PoolScope, where the pool itself is also allocated. The PoolScope also constitutes the main execution environment where the operations of the pattern take place. The EventHandlerLauncher is a Runnable object, started with a (NoHeap) Real-timeThread, that implements the infinite loop described in Section 2.1. According to Figure 3, once an EventHandlerLauncher is selected from the pool and is woken up (operations 1 and 2 in Figure 3), it enters the associated EventHandler in the assess HandlerScope (operation 3). The EventHandler, in turn, uses the EventSelector object to obtain the next event to process (operation 4). EventSelector is a user-defined object that implements the specific event retrieval code; it can be, for example, the acceptance of a new socket connection, the reception of a new message from the network, etc. Once the event is obtained, the EventHandler signals the pool to elect a new leader in order to process another event (operation 5), if present, thus ensuring a high degree of parallelism. Then the EventHandler processes the event and, once handling is over, it exists the HandlerScope (operation 6), thus deleting all the objects created during processing, and then gives back the control to the EventHandlerLauncher. The latter, finally, returns itself to the pool (operation 7) and suspends itself once again, waiting for a new wake up, when eventually selected.

Implementation. The complete class diagram of the RTJ-Leader-Follower pattern is reported in Figure 4; the reader can refer to [7] for the complete description of the interactions among objects (i.e. the sequence diagram). As the Figure shows, event selection and handling are made flexible by means of classes EventSelector and EventHandler.

---

Figure 3. The RTJ-Leader-Follower Pattern
The former is defined as an interface: the user can thus write her/his own code\(^4\) implementing event retrieval, given that a method `getNextEvent()` is exposed. The latter—EventHandler—is an abstract class, to be extended by a ConcreteEventHandler, offering a callback method (`pollEvent`) that must be overridden to implement user-defined event handling; EventHandler offers also other useful methods to adjust the (real-time) execution priority (`setPriority`) and to trigger new leader election (`electLeader`).

### 3.1 Objective Compliance

Having derived, in Section 2.3, some objectives to be met by RTSJ-compliant design patterns, let us analyze here how the rtj-leader-follower described above is able to comply with all of them.

**O.1 Correctness.** The criticality of an event can be determined only after that the event itself is received and its type is known. Therefore, in our solution, all EventHandlerLaunchers run at a programmable priority. When the event is then received, the ConcreteEventHandler has to call the `setPriority` method to adjust execution priority according to the event type, now identified. Notice that different leader-followers could be associated with different network endpoints, so to create priority lanes \([11]\) to limit head of line blocking on non-priority preserving transport protocols, such as TCP/IP.

**O.2 Compliance.** The solution provided is strongly based on RTSJ structures; it can be also proved that RTSJ constraints (i.e. single-parent rule and object reference constraints) are not violated.

**O.3 Safeness.** Object creation is only allowed when user-defined objects, i.e. ConcreteEventHandler and EventSelectorImpl\(^5\), execute in a HandlerScope. Exiting the scope when the processing is completed ensures that all created objects are reclaimed, thus avoiding memory leaks.

**O.4 Efficiency.** The solution provided follows the same design principles of the leader-follower pattern \([17]\). Therefore, since the latter is designed to meet this objective, the same can be said for the rtj-leader-follower.

**O.5 Transparency.** As reported above, the user has only to implement his/her ConcreteEventHandler and EventSelectorImpl classes and is not forced to use RTSJ services or abstractions\(^6\).

\(^4\)And even a complex package with a class hierarchy.

\(^5\)The class implementing the EventSelector interface.

\(^6\)Unless there are application-specific requirements, e.g. starting other threads to perform the processing of an event using parallel activities, but this is not a common case.
Figure 6. ScopedTunnels and the circular queue

i.e. the time between the instant before leader wake-up (operation 2 in Figure 3) and the instant before getting the event (operation 4 in Figure 3). The same test has been performed using a non-RTSJ implementation of the leader-follower pattern and measuring, also in this case, the time before waking-up the leader and the instant before getting the event. This measures indicate the cost payed for the use of the chain of “handler” objects, as well as the overhead introduced by entering a handler in the scope.

The results, reported in Figure 5, clearly show that the performances of the non-RTSJ solution plus those of scope entering are about the same of the performance of the RTSJ-based pattern. This means that the most of the time is spent in entering the scoped memory, while the overhead of using the chain launcher/handler is not significant. Finally it is worth noticing that, excluding the effects of garbage collection, the predictability of the two solutions is practically the same for jRate, while it is slightly worse for the RI’s RTSJ-based implementation. This stems from the less predictable scoped memory enter time provided by the RI. The main differences between jRate and RI performances w.r.t. the management of scopes, as described in [6], does not depend on the compiled nature of jRate-based applications, but on the optimization and cache friendliness of jRate’s scoped memory implementation. Indeed you should consider that the RI implementation of scoped memories is native as well.

4 Scoped Tunnels

This Section deals with the half-sync/half-async pattern and presents a suitably adaptation to make it compliant to RTSJ.

Context. The same of the traditional half-sync/half-async [18, 17], given that data must be processed according to the associated time constraints.

Problem. The problem to solve is to perform, using RTSJ software, time-critical data transfer from an asynchronous thread to a synchronous one, also meeting the objectives reported in Section 2.3.

Solution. The solution we propose exploits (once again) the concept of object pools. A set of scoped memory areas—each called ScopedTunnel—is created in the immortal memory; each area is destined to hold a single data item produced by the asynchronous thread and, as the name suggests, acts as a mechanism to transport data from asynchronous to synchronous side. As Figure 6 illustrates, these areas are organized in a classical (synchronized) circular queue: each time the asynchronous thread has produced a data item, it places it in the first free ScopedTunnel of the queue, using a mechanism that will be described later on. This data item, which in principle can be any Java object, is associated to the ScopedTunnel through the portal and thus can be accessed by means of the getPortal method.

On the other side, the synchronous thread picks the first ScopedTunnel from the queue, enters in the scope and obtains the data item through the portal object; then it consumes the data and, after processing is over, exits the scope causing the data item object to be (correctly) deleted, as well as any other object used during processing; finally, the ScopedTunnel that has been used till now is returned to the queue, to be made available back again to the asynchronous thread. The particular and interesting aspect of this solution is the idiom used by the asynchronous thread to place the data item in a ScopedTunnel; such an idiom is derived from the memory tunnel pattern proposed in [7] and is based on allowing a Java object to be written into a scope by performing a cloning operation: this idiom permits to “break the barriers” imposed by RTSJ memory model limitations, without provoking sensible consequences. This operation is made possible by introducing a new class, called ScopedTunnel and derived from the RTSJ ScopedMemory, which offers a suitable cloneToPortal method. This method re-creates the object given as parameter in the memory space held by the ScopedTunnel, by performing a deep copy of the attributes. This copy operation is not the same of a classical cloning performed in Java\(^7\), because some issues related to the constraint of both the RTSJ and the real-time environment must be considered. In

\(^7\)By means of the method Object.clone()
particular, the solution must avoid the introduction of unpredictable latencies, loss of performances and violations of RTSJ constraints. To this aim, the following rules are applied:

a) The data object to be cloned cannot be referenced, in the source code, as an interface. In fact, using interfaces hides the real class of the object that will be cloned, so the cost of performing the operation cannot be estimated (hence the WCET) through a static byte code analysis.

b) The data object can contain references only to constant-sized arrays of primitive Java types (i.e., ints, floats, chars, etc.), which are allocated in the same memory area of the data object itself. This required to ensure adequate and predictable cloning times, avoiding any violation of RTSJ rules.

Such constraints are indeed not so strict because the cloning mechanism of Scoped Tunnels is essentially designed to allow the exchange of (short) messages, which can be thus encapsulated into e.g. String objects, arrays of chars or ints, etc., and not for the exchange of more complex object graphs.

Structure. The structure of the half-sync/half-async pattern with Scoped Tunnels is depicted in Figures 7 and 8, by means of UML class and sequence diagrams. As it can be noted, both asynchronous and synchronous activities are implemented through a (NoHeap)RealtimeThread that, in turn, enters a Runnable object in a scope; the latter is entailed with the task of performing the real processing at the async- or synchronous side, so that, once such an activity is concluded, exiting the scope allows created objects to be automatically reclaimed. While, at the asynchronous side, the scope area is a classical ScopedMemory object, the synchronous thread uses the ScopedTunnel: by entering this scope, not only the data item can be found there and accessed through the portal, but the scope can be also used as execution domain for the synchronous processing activity.

Other Solutions. Solutions different than that of cloning can be still used, but they do not feature the same characteristics, flexibility and general validity of our proposal.

A first alternative entails to create the data item object directly in the ScopedTunnel area through the newInstance/newArray method. However, this solution is not generally valid since it cannot be applied if the data item object is created by a Java library function, through the use of the standard new opcode, following the invocation of a suitable method performed by the AsyncHandler.

A second alternative is the WedgeThread pattern, described in [2, 16], which, in our scenario, implies to force AsyncHandler to not exit from its memory scope until the SyncHandler enters it; this allows all objects created from the former handler to be made available also to the latter, which, by exiting the scope at the end of the activity, can delete all. The main concern of this solution is that, in order to allow a single data item object to be made available to the other side, all the objects temporarily created by the AsyncHandler during its processing must be kept in the scope, even if they are now useless. So this solution features inefficiency since it uses (and wastes) more memory than the quantity really required.

4.1 Objective Compliance
Let us state, in this Section, how the Scoped Tunnel solution is able to meet the design objectives listed in Section 2.3.

O.1 Correctness. Data handling, from both asynchronous and synchronous sides, is performed in the context of the execution of (NoHeap)RealtimeThreads, thus real-time characteristics are ensured.

O.2 Compliance. It is easy to check that RTSJ memory model constraints are not violated. The main concern is instead the use of a non-standard (for RTSJ) mechanism to perform object transfer, that is the tunnelling. However, it is designed in such a way as to be compliant with RTSJ, in fact the constraints introduced in object cloning through the tunnels ensure determinism and adherence to RTSJ memory model principles.

O.3 Safeness. Basically the ScopedTunnel idiom does not introduce memory leaks. Moreover, as it has been dealt with many times in the previous Subsection, the pattern in Figure 7 is structured in such a ways as to ensure deletion of unused object when processing is completed.

O.4 Efficiency. The solution is based on pre-allocating the various objects at startup and then using them. The only inefficiency is indeed the copy operation required to transfer the data item object through the tunnel. But it has to be noted that, even if another solution like the WedgeThread could not require such a copy, it introduces inefficiency in memory usage, as illustrated above.

O.5 Transparency. Data handling at the synchronous side is performed by means of a ConcreteSyncHandler object implementing the handleData method (see Figures 7 and 8). The programmer writing the code of such a method has only to take care of processing the data item object received as parameter. All the other activities, i.e. getting the scope, entering the handler, getting the portal, etc., are performed by all the other objects of the pattern and thus are transparent for the programmer.
4.2 Performance Evaluation

We wrote a prototype implementation of the Scoped Tunnel using jRate and employing native code for the cloneToPortal method. Two different versions of the Scoped Tunnel have been implemented. The first version provides a polymorphic object cloning mechanism, meaning that it can be used to transfer any generic objects. This choice, even if it provides great flexibility, has a strong impact on performances. In fact, a cost is paid each time an object is copied, since the cloneToPortal method has to determine, at runtime and for each copy, the object layout in order to access the attributes\(^8\).

However, when tunnels are used in contexts where the object exchanged is often always of the same class, the knowledge, at design time, of the class of objects that will be transferred allows not only a WCET analysis, but also the advance preparation of the structures needed to perform object copy, thus improving performances. For these reasons, we implemented also a typed Scoped Tunnel that allows cloning of objects belonging to a precise class, which is specified by means of an additional suitable method.

Using this prototype implementation, we measured the time required to perform a cloneToPortal operation vs. the size of the copied object, for both non-typed and typed tunnels. We also compared the performances of using a traditional reference-passing mechanism rather than object copy. The results\(^9\) are reported in Figure 9.

As the graph clearly shows, tunnel’s performances depend on object size with a linear and predictable law. Typed tunnels performances are very interesting if compared to both the non-typed version and the simple reference passing: indeed, about 77% of CPU time is saved with respect to the non-typed version. Moreover, the cost of copying a 1000 bytes-sized object, which is 18 times the reference passing time for non-typed tunnels, becomes only 4 times for typed tunnels.

5 Related Work

There are many papers dealing with various aspects of the RTSJ memory model, but a few of them deal with idioms and patterns for engineering distributed applications with the RTSJ. As reported in \([16, 5, 2, 9, 8]\), the issues introduced by RTSJ memory model require a deep investigation on how well-known design patterns can be adapted, or redesigned, in order to meet RTSJ constraints. For this reason, the aim of this paper is to cover the issues related to network software design by introducing new software engineering guidelines. As a reference, we report here a brief description of similar approaches comparing them with the proposals dealt with in this paper.

In \([4]\), the concept of object pools is introduced as a means to perform object allocation and deallocation manually. This approach makes possible the realization of many recurring patterns that, due to RTSJ constraints, are infeasible with automatic memory management; however, the use of object pools has two major drawbacks, (1) it requires classes to provide initialization methods other than the constructors, thus not allowing the use of Java software or libraries already implemented and/or available only in bytecode.

---

\(^8\)In our prototype implementation, accessing object attributes is done by obtaining (each time) their offsets in the C++ object v-table, and by using the internal reflection API of GCJ.

\(^9\)The test-bed is the same of the rtj-leader-follower pattern.
format, and (2) it delegates memory management to developers, thus creating the potential for memory leaks, and jeopardizing one of the most attractive Java feature—automatic memory management. The solutions proposed in this paper exploit pools, without however delegating to the user the memory management. Instead, objects can be automatically reused (not deleted and eventually recreated), while offering to user code a safe environment, able to handle object collection automatically.

In [16] some design patterns for RTSJ have been introduced. A part of the Wedge Thread, which has been already dealt with in Section 4, that paper proposes the Handoff pattern to allow the exchange of an object between two different memory areas. The approach is similar to that of Scoped Tunnels, but works only when the two scopes are sibling and share the same parent area. This is an additional constraint that is not present in our approach.

6 Conclusions

This paper has dealt with the problem of engineering Java real-time network software using RTSJ. Two well-known design patterns have been analyzed—the leader-follower and the half-sync/half-async—showing their “as-is” infeasibility in RTSJ due to memory model constraints. The relevant solutions for the RTSJ world are then presented and evaluated using both qualitative and quantitative metrics. Qualitative evaluation has been performed by using some parameters derived from the intention of offering a programming environment that is safe—according to Java semantics—compliant to RTSJ and able to meet real-time requirements. Quantitative evaluation has been instead performed by executing some performance tests using a prototype implementation. As a result, the proposed patterns are able to overcome the constraints imposed by RTSJ memory model, provide good performances and offer a flexible means for the design of real-time Java network software.

References


[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.


