J Supercomput (2014) 70:1588–1616
DOI 10.1007/s11227-014-1259-x

A simple distributed garbage collector for distributed real-time Java

Pablo Basanta-Val · Marisol García-Valls

Published online: 12 July 2014

Abstract The use of real-time distribution middleware programmed with high-level languages like Java is becoming of increasing interest in next generation applications. Technology like Java's Remote Method Invocation (RMI) paves the way towards these new distributed horizons. RMI offers many high-level abstractions useful for distributed application programmers to reduce their development times. One of these abstractions is a distributed garbage collector (DGC) that removes unreachable remote objects from the distributed ecosystem. However, in real-time Java, distributed garbage collection is underspecified and it introduces unbounded indeterminism on end-to-end real-time Java communications. This article analyzes this problem proposing a simple characterization for a predictable real-time distributed garbage collector (RT-DGC). The approach requires support from the middleware infrastructure that implements the abstraction but it also introduces bounded overhead. The article provides insight on the performance that RT-DGC offers to a distributed real-time Java application and the extra overheads due to the intrinsic cost of this abstraction.

Keywords Real-time · Java · Distributed garbage collection · Middleware · DRTSJ · RSTJ

This work has been partly supported by the iLAND project (ARTEMIS- JU 100026) funded by the ARTEMIS JTU Call 1 and the Spanish Ministry of Industry (www.iland-artemis.org), ARTISTDesign NoE (IST-2007- 214373) of the EU 7th Framework Programme, and by the Spanish national project REM4VSS(TIN 2011-28339).

P. Basanta-Val (✉) · M. García-Valls
Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Leganes, Spain
e-mail: pbasanta@it.uc3m.es

M. García-Valls
e-mail: mvalls@it.uc3m.es

1 Introduction

Current complexity in real-time systems development is increasing dramatically demanding new programming abstractions to reduce development and maintenance costs [1–4]. On one hand, ancient monolithic and small-size systems are being interconnected (e.g., [5,6]) to new applications that have to offer predictability to end-to-end communications. On the other hand, the number of lines of code of a typical real-time system is also increasing due to new application requirements that want to offer enhanced functionality. This state is also true even in industrial applications (e.g., [7]) that demand new abstractions to reduce their development and maintenance costs and to introduce flexible deployment and reconfiguration capabilities [8].

To deal with this problem, real-time practitioners may opt for high-level programming languages that reduce the development and maintenance cost of their applications. One of these languages is real-time Java [9] that targets applications that have complex requirements: hard real-time, soft, and general purpose performance mixed in a complex distributed real-time system.

Currently, the efforts in the real-time Java community are grouped into three main specifications: the Real-time Specification for Java (RTSJ) [10], the Distributed Real-time Specification for Java (DRTSJ) [11], and the Safety Critical Java Specification (SCJS) [12]. The first targets to single virtual machines and deals with issues related to centralized resource management, DRTSJ offers mechanisms to interconnect different virtual machines in a predictable way, and SCJS provides enhanced predictability for high-integrity applications.

So far, the three main specification efforts have evolved at different rates. RTSJ has implementations ready to be used to develop applications and currently it is optimizing its model with enhancements addressed in the JSR-282 [13] community process. SCJS is in a public draft revision, with prototypes that support the high-integrity models proposed in the specification. However, DRTSJ [14,15] is moving slower and the community lacks reference specifications, and implementations on which distributed applications develop.

The list of issues that DRTSJ has to face is extensive: [5,9,16]. Among them, the most addressed paradigm is the remote invocation (RMI) [17]: most researchers have produced specific contributions for this distributed abstraction. For RMI, researchers [16,18] proposed extensions to the API to interconnect different real-time networks, manage end-to-end priorities, and offer end-to-end predictable memory management models. For instance, new services such as a distributed event handler (DEH) and the synchronous scheduling service (SSS) were proposed to manage distributed events and to deal with new networked models. Lastly, other services, such as the naming service and the distributed garbage collector service defined in Java's Remote Method Invocation (RMI)[19], have not addressed real-time performance issues in depth.

Up to date, leading effort DRTSJ describes distributed real-time garbage collection as an interesting support [16]. However, the real-time Java community has not addressed this benefit–cost relationship in depth. The goal of the article is to provide a simple real-time distributed garbage collector (RT-DGC) approach for distributed real-time Java. DRTSJ, the main specification effort, profits from the results defined in this article to base in them its distributed garbage collector.

RT-DGC techniques are also relevant to other real-time Java's RMI approaches that have removed distributed garbage collection from their profiles and have identified real-time garbage collection as a part of their underlying support (i.e. [16,20,21]). The results included in the article help them to understand the extra cost that the use of a RT-DGC has in their infrastructures.

The rest of the article describes the RT-DGC algorithm proposed from different perspectives. Section 2 deals with related work focused on distributed garbage collection and real-time Java. Section 3 introduces the basic behavior included in Java's RMI. Section 4 proposes a simple approach to include a RT-DGC service based on three basic functionalities to manage the distributed garbage collector. Section 5 analyzes the impact of this service on a distributed real-time Java middleware, addressing programming interfaces and architectural issues. Section 6 evaluates the RT-DGC algorithm on a networked hardware infrastructure. Finally, the article ends with conclusions and future work (Sect. 7).

2 State-of-the-art

2.1 Algorithms for automatic memory management

The algorithms described for supporting RT-DGC have a directed connection to previous work on classic automatic memory management algorithms (compiled in [22,23]). The rest of this section connects this work and the technique proposed in Sect. 4.

The proposed algorithm is based on a real-time version of a previous algorithm described in the context of Network Objects (described in [24]) which was adopted in Java's RMI [25,26]. These types of algorithms are based on reference counting. From a real-time performance perspective, this is advantageous because distributed garbage collection algorithms do not require scanning all memory references to remove unreferenced objects. However, one of the limitations of reference counting is that these algorithms are not able to detect loops in remote object references and that they provide less garbage collection than root-scanning garbage collectors. This type of limitations is shared in common with the proposed distributed real-time garbage collector, which is based on reference counting techniques.

The algorithm proposed in this article has been readjusted considering distributed real-time Java constraints. Two types of adjustments have been carried out on the original skeleton of the Java's DGC. One is introducing a predictable resource management and a real-time Java characterization on the DGC algorithm; this type of support is not currently available in Java's RMI. Another is producing a simpler distributed garbage collector computational model, more efficiently designed (see Sect. 3). Current Java's RMI has internal messages (e.g., `DGCAck`) that improve the performance in cases where there is a failure in the network or a computational node. The proposed RT-DGC algorithm does not use the `DGCAck`, resulting in a simpler computational model. The failure detection functionality is managed by the distributed real-time Java platform (that provides mechanisms [17] to detect and recover failures in a distributed application).

Previous work in distributed garbage collection (see [23]) was more focused on performance issues (i.e., the number of messages and efficiency) than in the predictability of the DGC algorithm. This is the specific contribution of this work to the state of the art: a characterization of a DGC algorithm as a predictable facility integrated into a global system. Previous work was designed for general purpose applications and their garbage collectors do not have a running priority honored in each node.

The proposed approach is relevant for other centralized algorithms that provide applications with real-time performance like Metronome [27,28] and other approaches [29–33]. Basic real-time garbage collectors are based on Baker's incremental model described in [31]. Using this model, researchers [29] explore the definition of a time-triggered approach to the real-time garbage collector, deducting worst-case response times for the application. Based on these strategies Metronome collects heaps using time-based or work-based approaches taking 50 % of the available time. However, none of these centralized efforts was extended to participate in a process that comprises several virtual machines.

From the point of these real-time garbage collectors, the proposed techniques are interesting for extending its support to a networked environment, where remote objects are automatically collected when they are no longer in use. In order to provide a real-time characterization useful for the distributed garbage collector as a task that creates its own garbage to send/receive messages from other nodes. It also needs a mechanism that prevents a remote object from disappearing before all their remote references.

The integration of centralized garbage collectors with centralized garbage collectors seems simple because the distributed algorithm used in DGC algorithms creates new local references in the local nodes that prevent from the destruction of a remote object in a remote node. This simple interface is compatible with efforts like the algorithm included in IBM's real-time Java infrastructure [27] and the real-time garbage collector offered by Jamaica [28]. However, the tools that determine the allocation rates of the tasks of the system have to be modified to take into account the new requirements of the distributed garbage collector.

Another related piece of work is the use of efficient memory allocators that include techniques such as TLSF [34] which offers time-predictable allocation with low fragmentation. These techniques are less relevant than real-time garbage collectors in the scope of this article, which are the most natural choices when designing a real-time Java application.

2.2 Distributed real-time Java and distributed garbage collectors

The second part of the related work refers to distributed real-time Java technology and its interpretation of a RT-DGC service. Some previous approaches have identified this type of support as interesting, while others have forbidden its use (see [17]).

Approaches like RTZen [20] and other CORBA's approaches towards having a distributed garbage collector are set aside of the discussion because RT-CORBA does not support garbage collection as a key service. From RTZen's perspective, this type of service is not as relevant as in Java's RMI model. The technique is useful to extend RTZen with mechanisms that collect remote services that are no longer in use.

Regarding leading effort DRTSJ, real-time Java researchers identified having a predictable distributed garbage collector as a goal for distributed real-time Java in any of their integration levels [35]. In DRTSJ, distributed applications exchange remote object references which traverse several nodes. Currently, DRTSJ defines three integration approaches: L0, L1, and L2. L0 refers to a plain Java's RMI running on real-time Java virtual machines without end-to-end real-time remote invocations. L1 refers to a system with predictability on real-time remote invocations. Lastly, L2 refers to the possibility of having enhanced functionality on RMI, such as distributable threads. Some extensions defined for the L2 integration level include a distributed event handler and distributed remote objects which are used by distributable threads. The support given by the RT-DGC algorithms described in the article offers a more predictable approach for these L2 facilities.

Researchers from the University of York ([16]) worked in a framework for distributed real-time Java. Among the set of challenges they identified, having a predictable DGC mechanism is in the improvements list. However, they do not provide practical approaches solving these issues. Thus, the proposed solution offers a practical approach to address integration between DGC mechanisms and York's real-time RMI framework.

Researchers from Universidad Politécnica of Madrid ([21]) defined rules for different distributed real-time Java platforms. All their solutions forbid the use of a distributed garbage collector and some of them also the destruction of remote objects. Therefore, the approach described in the article is useful for flexible frameworks that support remote object deallocation, meanwhile its high-integrity infrastructure cannot use this support.

Lastly, this work impacts on previous contributions on the DREQUIEMI framework ([8,36]). DREQUIEMI identified a real-time garbage collector as an internal service for remote invocations. However, the work was silent on how this type of functionality should be added. The article provides algorithms for supporting a predictable DGC service in RMI (Sect. 4), which is compatible with the remaining services described for DREQUIEMI.

3 Background on the Java's RMI distributed garbage collector

This section describes the distributed garbage collector included in Java's RMI (see [37–39]), analyzing different drawbacks and practical issues in its behavior.

In essence (Fig. 1), the distributed garbage collector in RMI consists of two methods `clean` and `dirty` which are invoked from the core of the middleware. The `dirty` message takes as an input the identifiers of the remote objects that are going to be renewed and the leasing time. The `clean` message requires also an identifier of the remote virtual machine.

The operational rules of the distributed garbage collector are the following:

- Each time a node receives from another node a reference from a local node, the node sends a `dirty` message to the remote server to indicate that there is a new

```
package Java.rmi.dgc;
public interface DGC extends Remote{
 void  clean(ObjID[] ids, long sequenceNum, VMID vmid, boolean strong)
   throws RemoteException;
 Lease  dirty(ObjID[] ids, long sequenceNum, Lease lease)
   throws RemoteException;
  }
```

Fig. 1 `Java.rmi.dgc.DGC` interface included in current Java's RMI

reference in the system. The `dirty` message keeps the reference for a certain amount of time. And after that time, the reference has to be renewed with another `dirty` message. The `dirty` message takes as an input the identifiers of the remote objects that are going to be renewed and the leasing time.

– Each time a local remote reference to a node disappears, the node sends a `clean` message to the owner of the remote node to indicate the reference is no longer in use.

3.1 Use case

To exemplify the behavior of the DGC algorithm, Fig. 2 introduces a simple scenario with three nodes: Node 0, Node 1 and Node 2. Node 0 has a remote object named `Ro`, Node 1 has a remote reference (`Rr(ro)`) to this remote object and Node 2 has no references to `Ro`.

Under this initial assumptions (see Fig. 2), Node 1 has to perform periodical renewals. This renewal process is shown in the step (0) of Fig. 2.

From that time on, the example shows how to move the remote reference from the Node 1 to Node 2. The process consists of four steps to be complete:

1. The first step is an invocation from Node 2 to Node 1. This invocation is carried out at application level by means of an application-dependent remote method. In the example, the method has the following signature: `Remote getRemoteObject()`.
2. Internally, the method returns a reference to a remote object (`Rr(ro)`)that moves from Node 1 to Node 2. After that, the local node notifies the owner that it received this message.
3. The notification to the owner node is done via a `dirty` message.
4. Lastly, the method notifies to the Node 1 that it has received the reference. This goal is supported with a `DgcAck` message, sent from Node 1 to Node 2. From that time on, the server has to renew the remote reference periodically before its time out.

The example also includes a case where the remote object is removed (see Fig. 2). In the example, the cleaning happens when the two remote references in Node 1 and Node 2 disappear. In that case, each node has to send a `clean` message to the corresponding node. When there is no remote references to the remote object, `Ro` becomes ready to be collected.

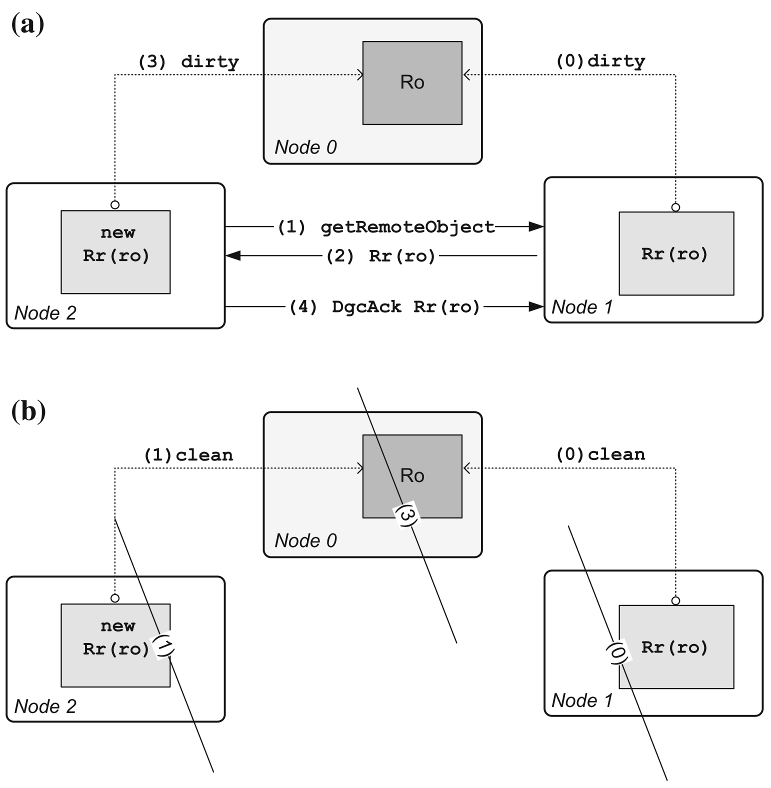

Fig. 2 An example of three nodes interacting with the Java's RMI DGC. **a** Creating a new remote reference, **b** removing remote references and destroying a remote object

3.2 Enhancing predictability in the Java's RMI distributed garbage collector

After analyzing the distributed garbage collector from a real-time perspective, the following two issues have to be considered:

– Enhanced predictability: The current definition of the distributed garbage collector is insufficient from the point of view of a real-time infrastructure like real-time Java. The list of issues that have be addressed includes the lack of worst-case execution times for the garbage collector process, lack of relationships with the CPU (e.g., the possibility of defining priorities for different nodes) and a proper characterization for the leasing process.
  Our predictable version of the distributed garbage collector (see Sect. 4) provides the distributed garbage collector with this type of functionality.
– Efficiency and optimizations: Another important issue is the efficiency of the distributed garbage collector. The example shows that the distributed garbage collector may be rather inefficient due to the number of messages exchanged (e.g., the `DgcAck` messages sent to acknowledge the reception of a remote reference).

The predictable version of the distributed garbage collector (Sect. 4) removed some of these messages changing the responsibility of notifying that there is a new remote reference to the node that enables the transference of the remote reference.

4 A predictable distributed real-time garbage collector

This section describes a simple real-time garbage collector in the context of a distributed real-time model which communicates via real-time Java remote invocations. The model is derived from an analysis of Java's RMI distributed garbage collector (see Sect. 3). These invocations are supported by a priority-driven scheduling infrastructure and its corresponding worst-case execution time. The integration is carried out by characterizing the two messages exchanged by the distributed garbage collector and the leasing model.

The following two major constraints are introduced in this section to describe the real-time distributed garbage collector:

– The proposed garbage collector is focused on an RMI layer. Other elements of the architecture (such as an optional operating system that maps this threads to operating system concurrent units: threads, and/or interrupt handlers) are set aside of the discussion. The model is focused on RMI specific issues and describes interaction between different Java RMI actors.
– The interactions between the different garbage collectors in a distributed system are described in isolation to explain the collection algorithm and its characterization as a real-time task. This characterization is required to use different scheduling techniques (such as response time analysis for priority-driven systems, see [40]).

4.1 Remote invocation model

The goal of the remote invocation is to invoke a remote method allocated in a remote server from a local client. The basic model for communications used in this section (see Fig. 3) assumes that each Java's RMI communication comprises seven logical steps in each remote invocation (based on the model described in [18]):

$$(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7) \qquad (1)$$

In the first step (namely step 1 in Fig.1), the middleware takes input data (`data` in Fig. 3) via a stub remote object. After taking the control, the application data are serialized (step 2) and sent to the server. Once at the server node (step 3), the data sent from the client are deserialized and the remote object is up-called (in step 4). The remote object receives data from the client and generates the response information (`res` in Fig. 3) returned to the client. These data are serialized at the server (step 5) and deserialized at the client (step 6). At the end of the invocation, the returned data (`res`) are at the application level (after step 7).

The latency of the network is integrated in step 2 and step 5 as a part of the internals of the middleware. In addition, if the type of the network is known, the model may be

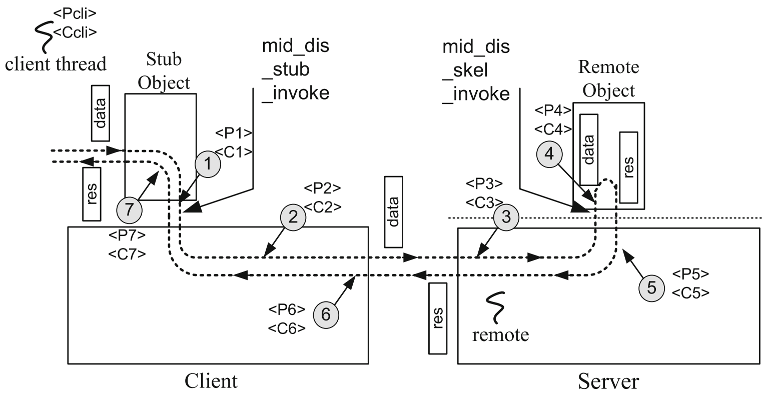

Fig. 3 Remote invocation: communication model and priority characterization for the remote invocation process

extended with this information. This type of approach is valid for producing a simple model that may be extended with particular network models. This type of network may be a dedicated RS232 link, a Switched-Ethernet LAN, or the Internet. In all these networks, the transmission model and cost have to be integrated with the remote object model proposed in this section (as a subsequence in step 2 and step 5).

4.1.1 Real-time task characterization

Before venturing into the definition of a predictable model for the remote invocation, we introduce the task model used in the distributed system. The task model consists of a set of n independent transactional tasks:

$$\Gamma = \tau_1, \ldots, \tau_n \tag{2}$$

that may be running in m independent processors:

$$\Pi = \pi_1, \ldots, \pi_m \tag{3}$$

In this model, each transactional task is characterized with a period: T_i, a maximum deadline: D_i, and set of seg_i different segments sequentially executed. Each segment consists of a worst-case execution time: C^{seg}, a priority: P^{seg}, and an execution node: π^{seg}:

$$\tau_i = (T_i, D_i, [C_i^1, \ldots, C_i^{\text{seg}_i}], [P_i^1, \ldots, P_i^{\text{seg}_i}], [\pi_i^1, \ldots, \pi_i^{\text{seg}_i}]) \tag{4}$$

Most application deadlines: D_i and inter-arrival times: T_i are given by the application requirements. Also, worst-case execution times: C_i^j in each segment may be empirically estimated (measuring its value) or alternatively by means of worst-case execution time techniques such as those described in [41]. Regarding the running nodes for each segment: π_i^j, they are typical application requirement too.

The transactional model requires a priority P_i^j to be defined for each segment. Unfortunately, in a distributed real-time system, there is not an optimal priority assignment policy that may be universally used [42]. Therefore, in the context of this article, we propose the following strategies:

1. A global priority per each τ_i task:

$$P_i = P_i^j \qquad \forall j \in [1, \mathrm{seg}_i] \tag{5}$$

   This type of approach requires one priority for each task shared in all transactions. Its main benefit is that it simplifies the computational model, resulting in the following task model:

$$\tau_i = (T_i, D_i, [C_i^1, \ldots, C_i^{\mathrm{seg}_i}], P_i, [\pi_i^1, \ldots, \pi_i^{\mathrm{seg}_i}]) \tag{6}$$

   Notice that this simplification does not address the problem of assigning global priorities to the different tasks of a distributed system. This type of assignment is typically done by considering the nature of the different transactional tasks. For this type of assignment one may resort to different approaches such as the assignment of priorities proportionally to the period and the deadline of applications (called deadline-monotonic assignment or rate-monotonic assignment in the literature [42]). These assignments are known to be suboptimal for general sets of transactional tasks.
   The main drawback of the approach is the stiffness of the computational model that requires a global priority to be defined for each task and that potentially may lead to suboptimal configurations.
2. A priority for each task in each node:
   To increase flexibility of the task set, another approach is to define an execution priority for each task in each node. This type of assignment may produce better performance results because it generalizes the global priority scheme to a more flexible scenario under the following constraint:

$$P_i^z = P_i^j \text{ with } \mathrm{seg}(i) \in \mathrm{node}(z) \qquad \forall j \in [1, \mathrm{seg}_i] \text{ and } z \in [1, m] \tag{7}$$

One important issue in both transactional models is that after a priority assignment, the resulting set of tasks may be analyzed using common-off-the-shelf techniques like the different response time analyses: [43–45] to check if application meet their deadlines.

4.1.2 Remote invocation as transactional segments

This section extends the computational model of a remote invocation with the information required to be modeled with the transactional model defined in Sect. 4.1.1. The proposed parametrization requires a maximum execution cost for each step and a scheduling relationship with the underlying CPU (i.e., a priority) to be defined in each step of the remote invocation:

- For each step described in Fig. 3, the model assumes that each step has a priority (Pi). In total, there are seven priorities: $\langle P1\rangle, \ldots, \langle P7\rangle$. Each priority corresponds to one of the steps (i.e. first step has $\langle P1\rangle$) and they are different for each different client.
- In addition, the middleware process has to define a worst-case execution time Ci for each step of the communication: $\langle C1\rangle, \ldots \langle C7\rangle$. Each Ci refers to the worst-case execution cost of a different step in a communication.

Using this model, the node may carry out remote invocations using seven segments (four in the client and five in the server) that are charged to the remote invocation. This remote invocation model is a particular case of the previous transactional model where the client and server interact in the client–server abstraction:

$$\begin{aligned}\mathrm{TRNS}_{ri}(\mathrm{cli}, \mathrm{serv}) = (& [C_{ri}^1, C_{ri}^2, C_{ri}^3, C_{ri}^4, C_{ri}^5, C_{ri}^6, C_{ri}^7], \\ & [P_{ri}^1, P_{ri}^2, P_{ri}^3, P_{ri}^4, P_{ri}^5, P_{ri}^6, P_{ri}^7], \\ & [\pi_{\mathrm{cli}}^1, \pi_{\mathrm{cli}}^2, \pi_{\mathrm{serv}}^3, \pi_{\mathrm{serv}}^4, \pi_{\mathrm{serv}}^5, \pi_{\mathrm{cli}}^6, \pi_{\mathrm{cli}}^7]) \qquad (8)\\ & \text{with cli and serv} \in [1, m]\end{aligned}$$

This model is compatible with the computational model defined in Sect. 4.1.1 because the invocation is described as a set of steps with maximum costs: C, priorities: P; and a running node: π.

Likewise to reduce the number of priorities, one may resort to the one-global priority strategy or to a per-node strategy. In the context of a remote invocation, the following strategies are available:

1. Client policies These policies use a single priority (that includes the priority of the client) for the seven steps of a remote invocation:

$$\begin{aligned}\langle P1\rangle = \langle P2\rangle = \langle P3\rangle = \langle P4\rangle \\ = \langle P5\rangle = \langle P6\rangle = \langle P7\rangle \qquad (9)\end{aligned}$$

   This type of policy is typically combined with the global priority approach described in Eq. 4.
2. Server policies
   The policies define a priority for the client side of the invocation and another for the server side (assuming that they are different nodes):

$$\begin{aligned}\langle P1\rangle = \langle P2\rangle = \langle P6\rangle = \langle P7\rangle \\ \text{and} \qquad (10) \\ \langle P3\rangle = \langle P4\rangle = \langle P5\rangle\end{aligned}$$

   This type of policy fits better with the idea of introducing local bounds on the resources used in a certain computational node.

The next section shows how this simple transactional model described in the context of remote invocations is useful for the messages exchanged by the distributed garbage collector, assuming that the distributed garbage collector uses remote invocations in communications.

4.2 A computational model for the RT-DGC service

RT-DGC requires taking additional actions into account before a remote reference to a remote object is to leave a node (to avoid race conditions); and after the reference to the remote object is destroyed (to allow remote object destruction). In addition, the RT-DGC has to renew its remote references periodically to deal with the possibility of having failures. This section shows how all these actions may be modeled using the transactional model previously described for remote invocations assuming that the distributed garbage collector algorithm is a remote object.

4.2.1 RT-DGC: remote reference creation

Before any remote reference (see Fig. 4) exits the local node, the middleware communicates the remote node in which the remote object is allocated. The goal of this communication is to avoid a prior-to-time destruction of a remote object by creating a local reference in the owner remote node.

Figure 4 shows this mechanism. The example assumes that data sent from client to server have a reference to a remote object. This remote object is allocated in a remote node named owner node. Before transferring the remote reference (step 2), the middleware communicates the owner node that a remote reference is to be transferred. This notification is done via a remote communication (described in Sect. 2) to the DGC service of the remote owner reference: i.e., $(1 \rightarrow \cdots \rightarrow 7)$ in Fig. 4.

The main difference between this communication and the remote communication is that the DGC communication is an internal communication between two remote nodes. In DGC messages, the user cannot write application code. In its most basic form (step 4), the remote service creates a local reference to the local remote object (see Fig. 2). The goal of this local reference is to avoid the remote object destruction. The reference is removed later, when the reference is destroyed in the server node.

4.2.2 Real-time transactional characterization

As in general remote communication, the middleware defines seven priorities for the DGC communication: $\langle$Pdgc1$\rangle$,..., $\langle$Pdgc7$\rangle$. In addition, a worst-case execution cost (i.e., $\langle$Cdgc1$\rangle$,...,$\langle$Cdgc7$\rangle$) has to be provided by the implementation.

Therefore, the `reference` mechanism may be characterized as the following transaction, following the model described in Sect. 4.1.1:

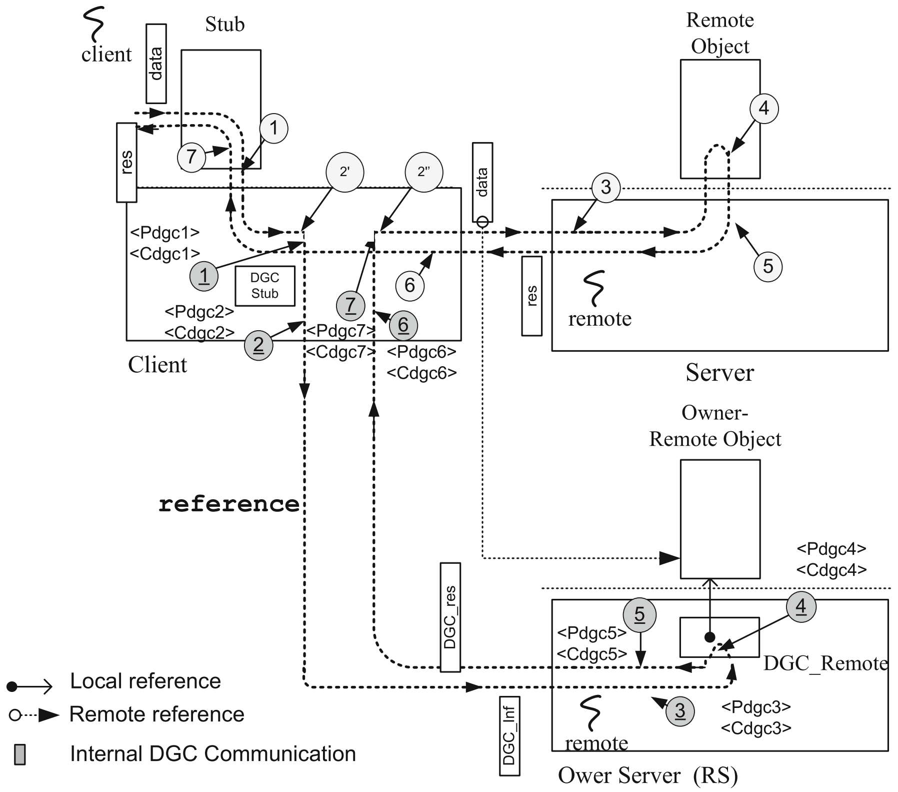

Fig. 4 RT-DGC model when a remote reference exits the local node. The client node is transferring a remote reference (to a remote object allocated in the owner node) to the server node

$$\begin{aligned}&\mathrm{TRNS}_{\mathrm{ref}_i}(\mathrm{cli},\mathrm{owner})\\&\quad=([\mathrm{Cdgc1}_{\mathrm{ref}_i},\mathrm{Cdgc2}_{\mathrm{ref}_i},\mathrm{Cdgc3}_{\mathrm{ref}_i},\mathrm{Cdgc4}_{\mathrm{ref}_i},\mathrm{Cdgc5}_{\mathrm{ref}_i},\mathrm{Cdgc6}_{\mathrm{ref}_i},\mathrm{Cdgc7}_{\mathrm{ref}_i}],\\&\qquad[\mathrm{Pdgc1}_{\mathrm{ref}_i},\mathrm{Pdgc2}_{\mathrm{ref}_i},\mathrm{Pdgc3}_{\mathrm{ref}_i},\mathrm{Pdgc4}_{\mathrm{ref}_i},\mathrm{Pdgc5}_{\mathrm{ref}_i},\mathrm{Pdgc6}_{\mathrm{ref}_i},\mathrm{Pdgc7}_{\mathrm{ref}_i}],\\&\qquad[\pi^1_{\mathrm{cli}},\pi^2_{\mathrm{cli}},\pi^3_{\mathrm{owner}},\pi^4_{\mathrm{owner}},\pi^5_{\mathrm{owner}},\pi^6_{\mathrm{cli}},\pi^7_{\mathrm{cli}}])\\&\qquad\text{with cli and owner} \in [1,m]\end{aligned} \tag{11}$$

The main difference between this model and the previous one is that the remote node refers to the node hosting the remote object, which is called owner.

The concept of client-propagated and server-defined priority is applied as in the previous case but with the owner node. The client-propagated policy enforces the following constraints:

$$\begin{aligned}\langle\mathrm{Pdgc1}\rangle&=\langle\mathrm{Pdgc2}\rangle=\langle\mathrm{Pdgc3}\rangle\\&=\langle\mathrm{Pdgc4}\rangle=\langle\mathrm{Pdgc5}\rangle=\langle\mathrm{Pdgc6}\rangle=\langle\mathrm{Pdgc7}\rangle\end{aligned} \tag{12}$$

In addition, the server-propagated policy on a DGC communication enforces the following constraint:

$$\begin{aligned}&\langle \text{Pdgc1}\rangle = \langle \text{Pdgc2}\rangle = \langle \text{Pdgc6}\rangle = \langle \text{Pdgc7}\rangle \\ &\qquad\qquad\qquad\qquad\qquad \text{and} \\ &\langle \text{Pdgc3}\rangle = \langle \text{Pdgc4}\rangle = \langle \text{Pdgc5}\rangle \end{aligned} \tag{13}$$

In both cases the goal of this mechanism is to be able to reduce the number of priorities involved in a remote invocation by means of inheriting or defining a priority. One simple approach may opt for using a global priority for each transactional task that includes also the interaction with the garbage collector. Other refined techniques may modify this default assignment by defining a global priority for the garbage collection.

4.2.3 RT-DGC: remote reference destruction

After a remote reference is destroyed, the node communicates to the owner of the remote object that the associated reference is no longer valid (see Fig. 5). As in the previous case, the communication with the owner node is done via a remote communication with a remote node with seven steps: $(1 \to \cdots \to 7)$. However, in the forth step of the invocation (i.e., step 4 in Fig. 5), the DGC algorithm removes the reference created to the remote object from its internal table.

4.2.4 Real-time transactional characterization

As in the case shown in Fig. 4, the priority model and the cost in each step are valid when notifying the destruction of a remote reference. In both cases, the implementation should provide a priority and a maximum cost for the seven steps of the model.

Therefore the `unreference` mechanism is characterized as the following transaction, which is compatible with the transactional tasks described in Sect. 4.1.1:

$$\begin{aligned}&\text{TRNS}_{\text{uref}_i}(\text{serv}, \text{owner}) \\ &\quad = ([\text{Cdgc1}_{\text{uref}_i}, \text{Cdgc2}_{\text{uref}_i}, \text{Cdgc3}_{\text{uref}_i}, \text{Cdgc4}_{\text{uref}_i}, \text{Cdgc5}_{\text{uref}_i}, \\ &\qquad \text{Cdgc6}_{\text{uref}_i}, \text{Cdgc7}_{\text{uref}_i}], \\ &\qquad [\text{Pdgc1}_{\text{uref}_i}, \text{Pdgc2}_{\text{uref}_i}, \text{Pdgc3}_{\text{uref}_i}, \text{Pdgc4}_{\text{uref}_i}, \text{Pdgc5}_{\text{uref}_i}, \\ &\qquad \text{Pdgc6}_{\text{uref}_i}, \text{Pdgc7}_{\text{uref}_i}], \\ &\qquad [\pi^1_{\text{serv}}, \pi^2_{\text{serv}}, \pi^3_{\text{owner}}, \pi^4_{\text{owner}}, \pi^5_{\text{owner}}, \pi^6_{\text{serv}}, \pi^7_{\text{serv}}]) \\ &\qquad \text{with serv and owner} \in [1, m] \end{aligned} \tag{14}$$

As in the previous case, the model may opt for client defined or server declared priorities. By default, an initial policy may opt for using the same priority in the local node and in the remote node (called owner) that hosts the unreferenced remote object.

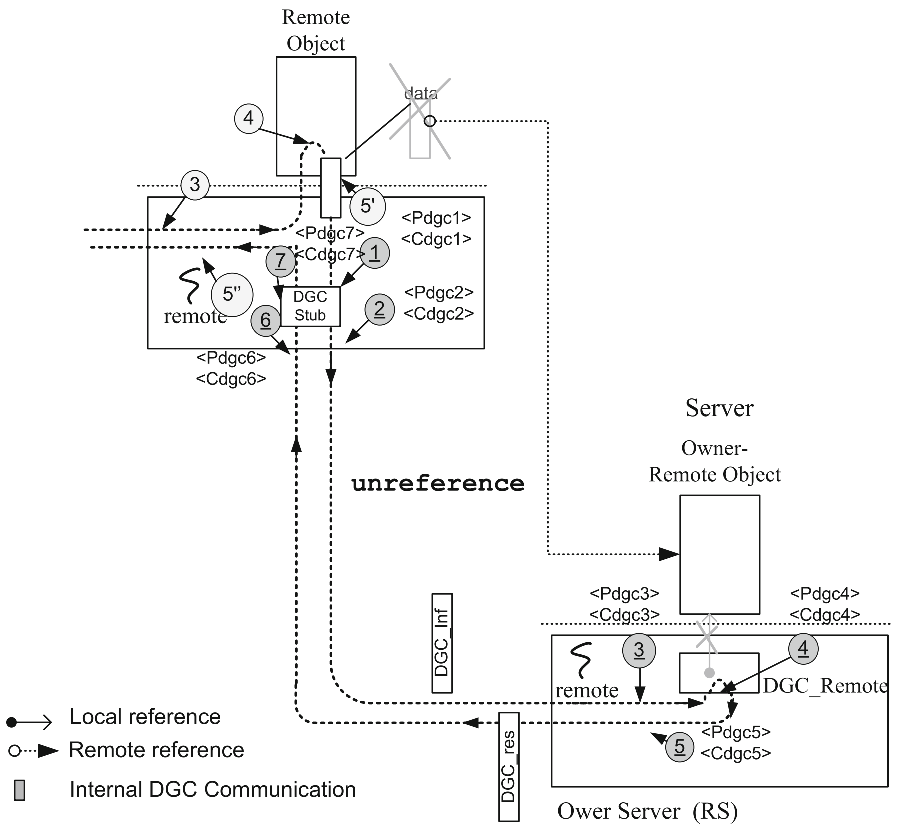

Fig. 5 Destruction of a remote reference (or unreference) in a client and corresponding notification to the owner server

4.3 RT-DGC: leasing model for remote references

To detect machine and remote object failures each node notifies periodically to the owner of a reference. This action is supported using a renewal mechanism, which renews a reference from a remote node. If a reference is not renewed periodically, then the RT-DGC algorithm may remove its related local reference from the set of remote references. This strategy helps to detect and remove references to non existing remote objects (see Fig. 6). The proposed model defines a periodic activity for the renewal process launched from the node that holds the remote reference and a periodic timer for the owner of the remote object (see Fig. 4).

For each reference, the local renews the lease via a $(1 \rightarrow \cdots \rightarrow 7)$ remote communication action. In the forth step of this communication, the remote node stores information about the time in which the last renewal arrived.

The owner node also requires a timeout associated to each reference stored in the local table of the DGC algorithm. Its goal is to detect (by waiting for a periodic renewal expiration) if a remote reference is still valid or not. If this reference is invalid, then the middleware removes the corresponding reference from the local table of references.

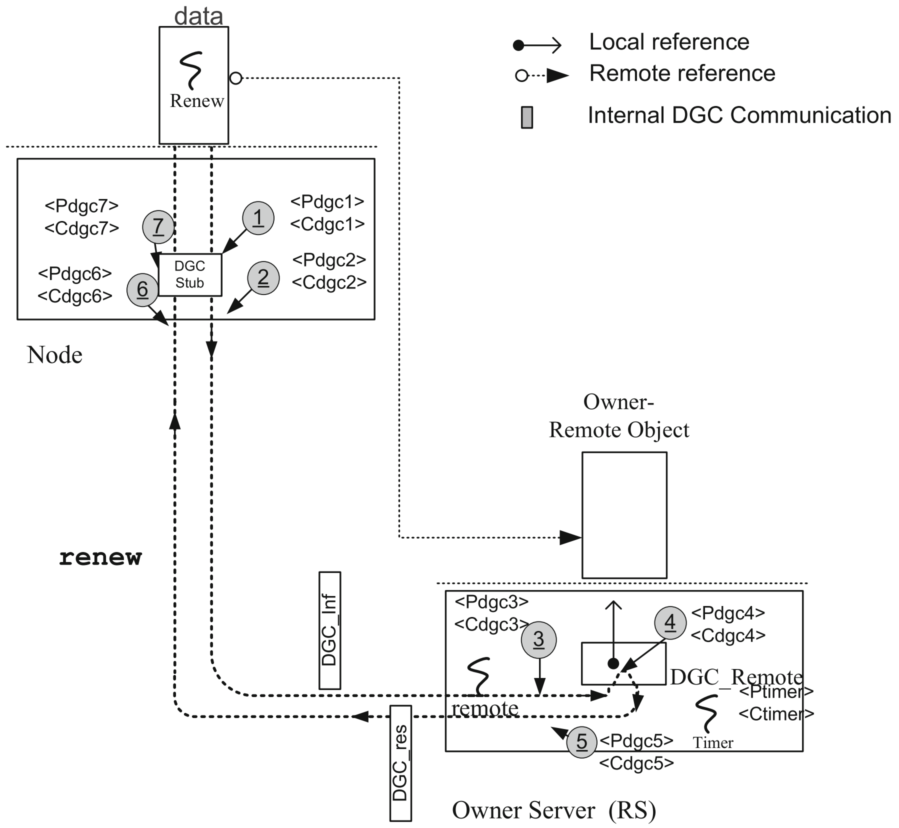

Fig. 6 Renewal message (from each remote reference to the owner)

4.3.1 Real-time transactional characterization

Two different activities have to be characterized: the leasing activity and the timer used at the owner.

First, it is the leasing mechanism, which requires defining a priority for the seven steps: ⟨Pdgc1⟩, . . . ,⟨Pdgc7⟩. In addition worst-case execution times have to be defined for ⟨Cdgc1⟩, . . .,⟨Cdgc7⟩.

In addition to costs and priorities, the designed algorithm has to deal with the leasing time. The leasing time is the time required in an owner node to decide that without a renewal, the remote reference is invalid (Fig. 7).

To transform the leasing time to a period (Tleasing) and a deadline (Dleasing) one approach is to define to use the following constraints:

$$\text{Tleasing} = \text{Dleasing} = \text{Leasing}/2 \tag{15}$$

As a result of this choice, the DGC leasing mechanism is characterized as the following transactional task:

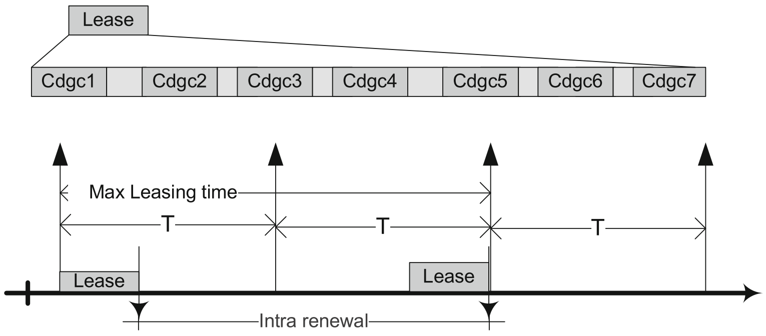

Fig. 7 Leasing process characterization proposed for the garbage collector

$$\begin{aligned}
&\tau_{\mathrm{ref}_i}(\text{hold, owner}) = (\text{Leasing}/2, \text{Leasing}/2,\\
&[\text{Cdgc1}_{\mathrm{ref}_i}, \text{Cdgc2}_{\mathrm{ref}_i}, \text{Cdgc3}_{\mathrm{ref}_i}, \text{Cdgc4}_{\mathrm{ref}_i}, \text{Cdgc5}_{\mathrm{ref}_i}, \text{Cdgc6}_{\mathrm{ref}_i}, \text{Cdgc7}_{\mathrm{ref}_i}],\\
&[\text{Pdgc1}_{\mathrm{ref}_i}, \text{Pdgc2}_{\mathrm{ref}_i}, \text{Pdgc3}_{\mathrm{ref}_i}, \text{Pdgc4}_{\mathrm{ref}_i}, \text{Pdgc5}_{\mathrm{ref}_i}, \text{Pdgc6}_{\mathrm{ref}_i}, \text{Pdgc7}_{\mathrm{ref}_i}],\\
&[\pi^1_{\text{hold}}, \pi^2_{\text{hold}}, \pi^3_{\text{owner}}, \pi^4_{\text{owner}}, \pi^5_{\text{owner}}, \pi^6_{\text{hold}}, \pi^7_{\text{holder}}])\\
&\text{with holder and owner} \in [1, m] \qquad (16)
\end{aligned}$$

Lastly, the characterization refers to the timer process (Fig. 4 at owner node) carried out to release the reference. In the approach, the timer is characterized with a maximum cost (⟨Ctimer⟩) and a running priority (⟨Ptimer⟩).

Selecting a proper leasing time requires dealing with a trade-off between accuracy in fault detection and computational overhead. The shorter the leasing time, the better the failure detection provided by the leasing mechanism and the higher the computational overhead due to the renewal messages of the distributed garbage collector.

5 Integration on a real-time Java platform infrastructure

The internals of a real-time Java middleware were modified to support the RT-DGC service described in Sect. 4.

In this infrastructure, the DGC service is in charge of removing remote objects when they are not in use; no manual removal is allowed. The DGC service extends the garbage collector hosted in each node to remote objects. It avoids memory leaks due to remote objects that cannot be referenced from remote machines.

Figure 8 contains the basic remote interface of the RT-DGC service described in the previous section. It contains three methods: one to create a remote reference from a remote node, another to unreference it, and a third for the renewal of the `reference/unreference/renew` methods to a certain locally hosted remote object. The `reference` remote method and `unreference` are both synchronous operations, called before a remote reference leaves a node and each time a remote reference is removed. The `renew` mechanism is invoked asynchronously; i.e., the application does not block for a renewal.

```
package es.uc3m.it.drequiem.rtrmi.server.dgc;
import Java.rmi.server.dgc.*;
public interface RTDGCInterface extends Java.rmi.Remote{
     public void reference(Java.rmi.server.ObjID[] objids)
             throws Java.rmi.RemoteException;
     public void unreference(Java.rmi.server.ObjID[] objids)
             throws Java.rmi.RemoteException;
     public void renew(Java.rmi.server.ObjID[] objids)
             throws Java.rmi.RemoteException;
}
```

Fig. 8 Garbage Collectors API. An inter-virtual machine interface for a real-time DGC service

The period of the renewal actions, the maximum execution costs, and the priorities are introduced as global parameters. The following new symbols are defined for Java's RMI to be parametrised:

- `rtdgc.LeasingTime`
- `rtdgc.LocalStubCost`
- `rtdgc.LocalSkelCost`
- `rtdgc.LocalStubPriority`
- `rtdgc.LocalPriority`

In each node, they control the main parameters of the garbage collector. The goal of the local cost parameters is to bound local interference generated from the distributed garbage collector inside other tasks as follows:

$$\text{LocalStubCost} = \text{Cdgc1} + \text{Cdgc2} + \text{Cdgc6} + \text{Cdgc7} \tag{17}$$

$$\text{LocalSkelCost} = \text{Cdgc3} + \text{Cdgc4} + \text{Cdgc5} \tag{18}$$

Lastly, the model allows defining a per-node policy (stub and server priorities) for priority server-defined policies. They relate to the model as follows:

$$\text{LocalStubPrio} = \text{Pdgc1} = \text{Pdgc2} = \text{Pdgc6} = \text{Pdgc7} \tag{19}$$

$$\text{LocalSkelPrio} = \text{Pdgc3} = \text{Pdgc4} = \text{Pdgc5} \tag{20}$$

6 Empirical evaluation

This section focuses on the evaluation of the proposed RT-DGC algorithm, described in Sect. 4. It quantifies its impact on an infrastructure described in Sect. 5. All results refer to a modified version of real-time Java, running on 1 Ghz MHz machines interconnected with a 100 Mbps Switched-Ethernet stack. Two goals are pursued in the evaluation:

– to establish empirical evidence on the intrinsic behavior of real-time garbage collectors.
– to offer performance results on the overhead introduced by the proposed RT-DGC mechanism.

The overhead was measured in several benchmarks. The results offer clues on the overhead this type of support in distributed real-time Java. Optimized implementations should outperform results included in this section.

6.1 Real-time DGC vs. plain DGC

6.1.1 Priority inversion due to the `renew` operations

The first evaluation scenario shows a simple scenario (see Fig. 9) where two clients try to access a remote object (`RO`) in a remote node server from two independent machines (called Client_1 Node and Client_2 Node in Fig. 9):

– The first node hosts a periodic activity (period = 25 ms) that invokes a local stub (`S1`) that transfers the information to the server. In addition, the server carries out a periodic local renewal from a DGC service to another in order to avoid prior-to-time removals (with a leasing time of 1 s and taking a 1 ms in each node to send the renewal message). This task takes 14 ms in the client and 10 ms in the server to complete its execution.
– The second client node hosts another activity (period = deadline = 50 ms) that invokes on the (`RO`) remote object. This task consumes 2 ms in the client node (i.e., node_2) and 8 ms in the server to complete its execution.

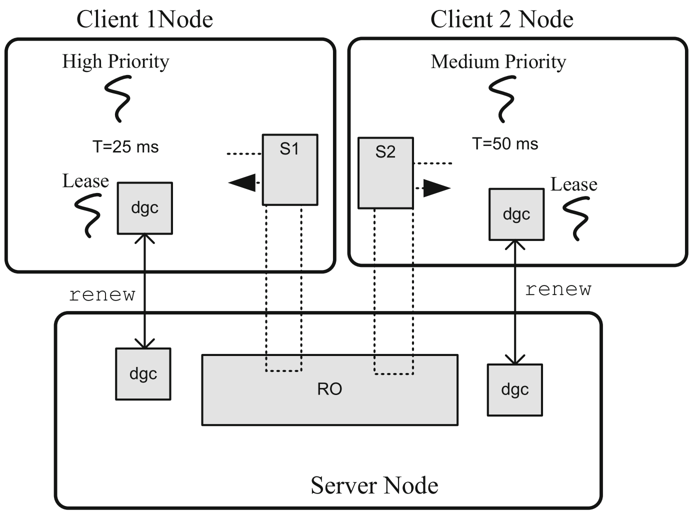

Fig. 9 Two clients accessing to the same server. All absolute times (in ms) refer to 1 Ghz machines connected with a Switched-Ethernet 100 Mbps network

Table 1 Results corresponding to the worst-case response times of tasks included in Fig. 9

Activity	Period	Deadline	Cost_server	Cost_Client_1	Cost_Client_2
Trans_D_25 ms	25	25	10	14	–
Trans_D_50 ms	50	50	8	–	2
Renew@Node1	1,000	1,000	1	1	1
Renew@Node2	1,000	1,000	1	1	1
Activity	**U_Cli_1**	**U_Cli_2**	**U_Serv**	**WCRT(RT_DGC)**	**WCRT(DGC)**
Trans_D_25 ms	0.56	–	0.4	24	**27**
Trans_D_50 ms	–	0.04	0.16	20	23
Renew@Node1	0.001	–	0.01	20	20
Renew@Node2	–	0.001	0.01	20	20

Results refer to two 1 Ghz machines connected with a Switched Ethernet 100 Mbps network

– Assuming a deadline-monotonic approach, the priority of the 25 ms task is higher than the 50 ms task. In addition, when the real-time garbage collector renewal has a priority defined, then this priority is lower than the priority of the other tasks (because its leasing time is 2 s, which corresponds to a 1 s period) whenever it is possible (Note: When there is no real-time garbage collection, this priority is unknown).

Under these constraints, the response time of the four transactions started on the clients (two in each client) are bounded (see Table 1). This table contains the four tasks characterization (periods, deadlines, and local costs in each node), the utilization generated by each task in a node (client 1, client 2, and server) and the resulting response time considering (WCRT_RT_DGC) (i) that there is a garbage collector, or (WCRT_DGC), (ii) that there is a non real-time garbage collector.

The following are remarkable outputs of the experiment:

– The indeterminism of the plain garbage collector generates pessimism in the server (that has to consider that it generates interference in other tasks).
– As a result of the previous situation, this indeterminism increases the response time of the 25 ms transactional task from 24 to 27 ms, missing its deadline (which was 25 ms).

6.1.2 Priority inversion on `reference` *operations*

Another set of experiments was conducted to show the type of real-time application that benefits from having a real-time garbage collector as part of its core. To show the benefits stemmed from a RT-DGC, a simple application was developed (Fig. 10).

It consists of three nodes. The first is a client node data that have the highest priority ($\langle 100 \rangle$). The client node communicates a remote object (R1) to obtain a remote reference (S2) to another remote object (R2). The second node (Node 1) is the node that holds R1. Lastly, the third node contains remote object R2 and a real-time task running at a $\langle 50 \rangle$ priority.

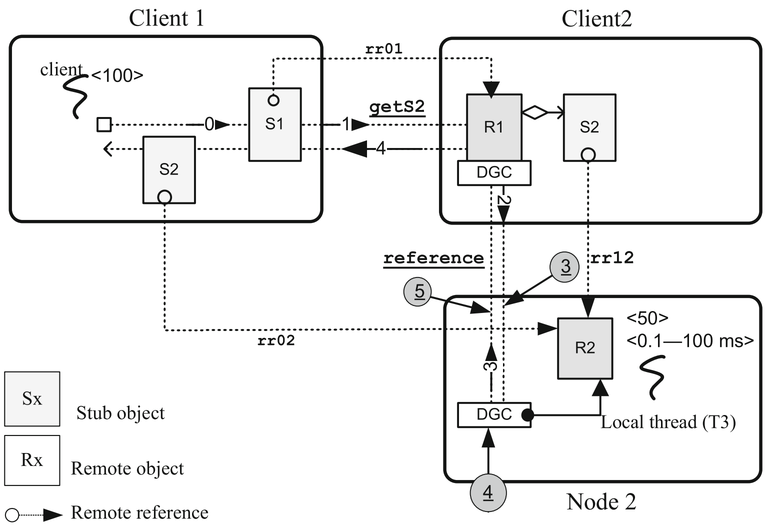

Fig. 10 Scenario for DGC and RT-DGC comparison

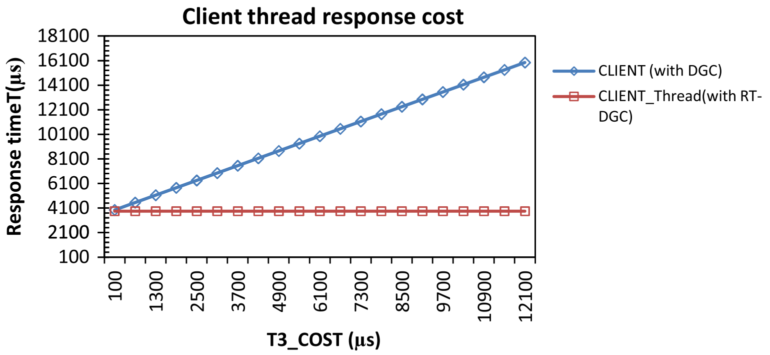

Fig. 11 Response time for the client with a plain DGC algorithm and with a RT-DGC

In this system there are two real-time activities with real-time constraints:

- The end-to-end activity (Node 0 → Node1) which retrieves a remote object reference stored in Node 2.
- A local thread activity (T3) in Node 2.

If there is not a DGC algorithm deployed, the two tasks should not interfere. However, the use of a garbage collector causes the subtle priority inversion in Node 2: T3 blocks the whole end-to-end activity at (3 → 4 → 5).

In a plain DGC, that issue is problematic because the execution priority of reference is not known a priori whereas in a RT-DGC enabled node it allows defining that priority. Empirically, this effect is shown in Fig. 11.

The figure shows that with a plain DGC the worst-case response time of the end-to-end communication receives interference from `T3`. In the worst-case analysis, it receives all the worst-case execution time from T3, which ranges in the application

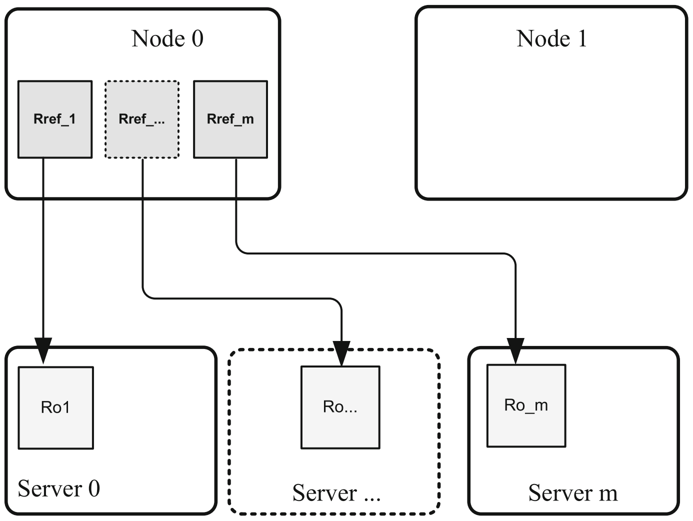

Fig. 12 Configurable scenario with multiple remote servers. *m* 1 GHz servers connected with a 100 Mbps Switched-Ethernet LAN acting as server nodes

from 100 to 12,100 μs. For the same interaction, the RT-DGC (with a client-propagated policy) offers a plain worst-case response time, because it is able to avoid the interference from the local thread `T3`.

Lastly, notice that the worst-case response time of `T3` does not depend on whether there is a real-time garbage collector or not. In both cases, `T3` has potential interference from the end-to-end transaction. In the case of the RT-DGC it is because it runs with a higher priority (i.e., ⟨100⟩). With plain DGC, the worst-case scenario is to assume that `T3` receives interference from the end-to-end transaction (since there is no information on its running priority).

6.2 Overhead introduced by the distributed real-time garbage collector

The second set of experiments refers to the overhead introduced by each remote reference. The results provide useful information (the overhead introduced by a real-time remote reference) to practitioners using similar settings (in terms of CPU and network). This section illustrates performance patterns while the next section refers to similar results obtained in an industrial application benchmark.

To show the performance patterns, a parametric scenario was developed (see Fig. 12). It consists of *m* different servers, each one of them is hosting a remote object that may be accessed remotely via a Switched-Ethernet LAN.

6.2.1 Overhead in operations

Figure 13 introduces the cost of managing remote references. This cost is divided into three types of operations: (i) the creation of a remote reference; (ii) its deallocations;

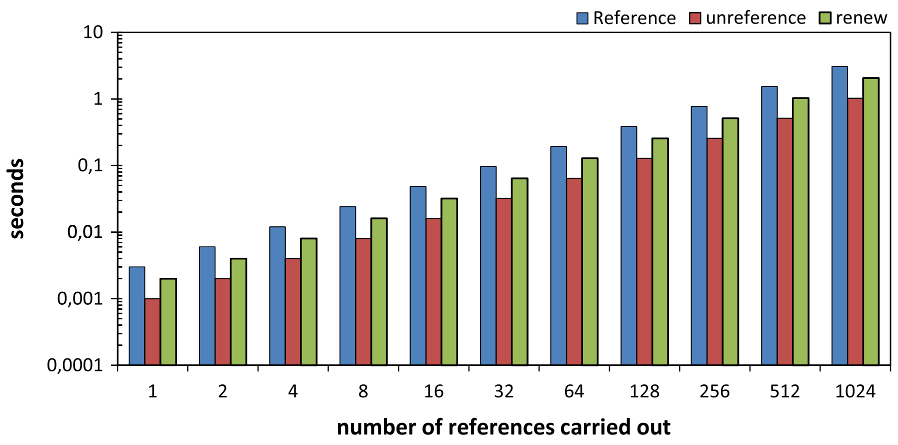

Fig. 13 Main costs in the proposed RT_DGC algorithm in a scenario with *m* servers each one then holding a reference. The cost refers to *m* consecutive invocations to `reference`, `unreference`, and `renew`

and (iii) renewal. Results show that the most inefficient mechanism is the `reference` method that has to allocate new structures within the middleware, the most efficient mechanism is the `unreference` execution, and the `renew` process is in between.

From the perspective of the scalability, the `reference` is the mechanism that challenges scalability on the system. The DGC algorithm mandates that a reference that is going to be sent out to another is to be communicated to the remote node before it disappears. So that, this cost cannot be deferred, as in the case of an `unreference` invocation or a `renew` command.

The influence of the `reference` command on the cost of the remote invocation was evaluated on the scenario described in Fig. 12 with a variable number of remote references (1, 2, 5, 10, 100, and 1,000) in applications with deadlines ranging from 10 ms to 10 s. Results (see Fig. 14) show that applications with a moderate number of remote references in a remote invocation may be satisfied with low overhead on the deadline of the application.

It also show how an increase on the number of transferred remote references means a high overhead due to the cost of communicating remote nodes that have a real-time garbage collector installed.

6.2.2 Leasing time performance

The rest of the section evaluates the impact of the renew mechanism on the cost of the remote invocation. The evaluation scenario used is the configurable scenario described in Fig. 12. For this scenario, the evaluation considers (i) how the number of remote references and (ii) the leasing period impact on the overhead introduced in the remote invocation.

Each reference introduces a leasing task with blocking in the local node and consumes resources in the remote reference holder, which notifies the owner about a new reference, and the owner, who has to process the renewal. The client experiences a

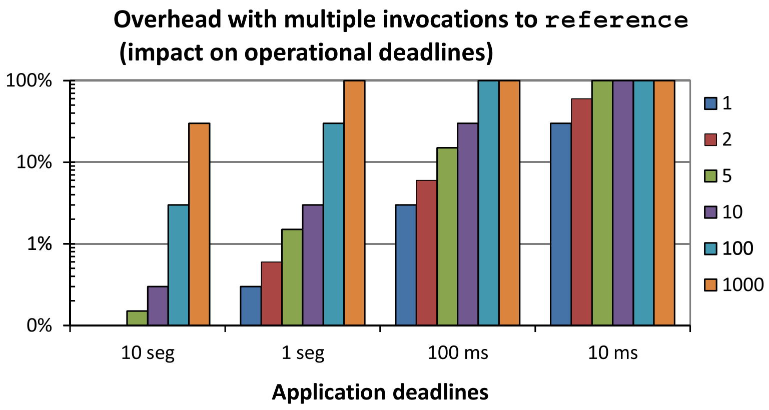

Fig. 14 Overhead of the `reference` mechanism on the cost of the remote invocation (1,2, 3, 10, 100 and 1,000 remote references). Each reference is hosted in a unique remote server. Application deadlines ranging from 10 ms to 10 s

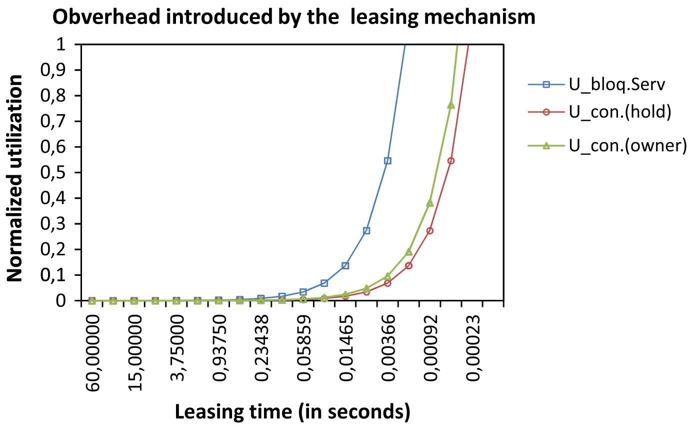

Fig. 15 Overhead introduced by the leasing mechanism of the RT-DGC algorithm

blocking (U_bloq_Serv) and consumes certain resources periodically which are modeled as a percentage in the utilization bound in the client (U_con_(hold)) and the remote object host (U_con(owner)).

Figure 15 summarizes the main results obtained after the experiment. It shows the evolution of the three types of cost with the leasing time; the shorter the leasing time is, the higher is the amount of CPU required per period. Notice that the following inequations are true:

$$\begin{gathered}\text{U\_bloq\_Serv} > \text{U\_con\_(hold)} \\ \text{and} \\ \text{U\_bloq\_Serv} > \text{U\_con\_(owner)}\end{gathered} \tag{21}$$

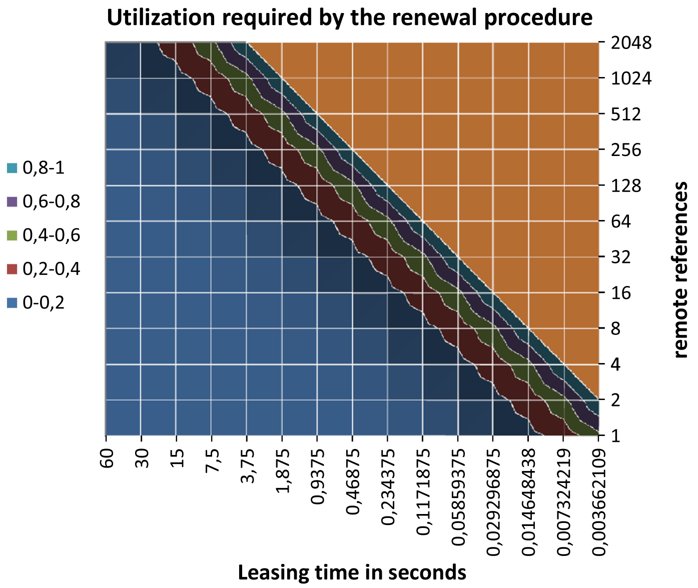

Fig. 16 Evolution of the utilization required by the garbage collection as a variable of the number of references in the system and the leasing time. M servers, each one holding a remote reference. 1 Ghz machines with a 100 Mbps Switched-Ethernet Network

because the blocking introduced at the holder includes resources consumed at server and owner nodes. In addition, notice that a relationship between U_con_(hold) and U_con_(owner) cannot be established.

The last part of the experiment evaluated U_bloq_serv only. It shows this cost for a bounded number of remote object references and leasing times. The obtained results show that for a leasing time of 60 s, one node is able to manage 1,024 references without much overhead (less than 10 %) (see Fig. 16). The available CPU time increases as the number of references increases and/or its leasing time decreases.

6.3 Evaluation on an application benchmark

The last set of results refers to an evaluation carried out on a benchmark derived from the Automotive Open System Architecture (AUTOSAR) [46] previously used to evaluate real-time Java performance in networked applications [36,47,48]. The former evaluation benchmark was extended to evaluate the overhead of the real-time distributed garbage collector.

The benchmark takes four operational frequencies (83.3 Hz, 0.10, 0.20 and 0.25 kHz). On these frequencies the benchmark considers the blocking time due to one remote reference (Low), two (Medium), and four (High) references in each periodic activation. The allocation costs refer to the cost of establishing a remote communication with the node that hosts the remote object.

The following are remarkable outputs from the benchmark (see Fig. 17):

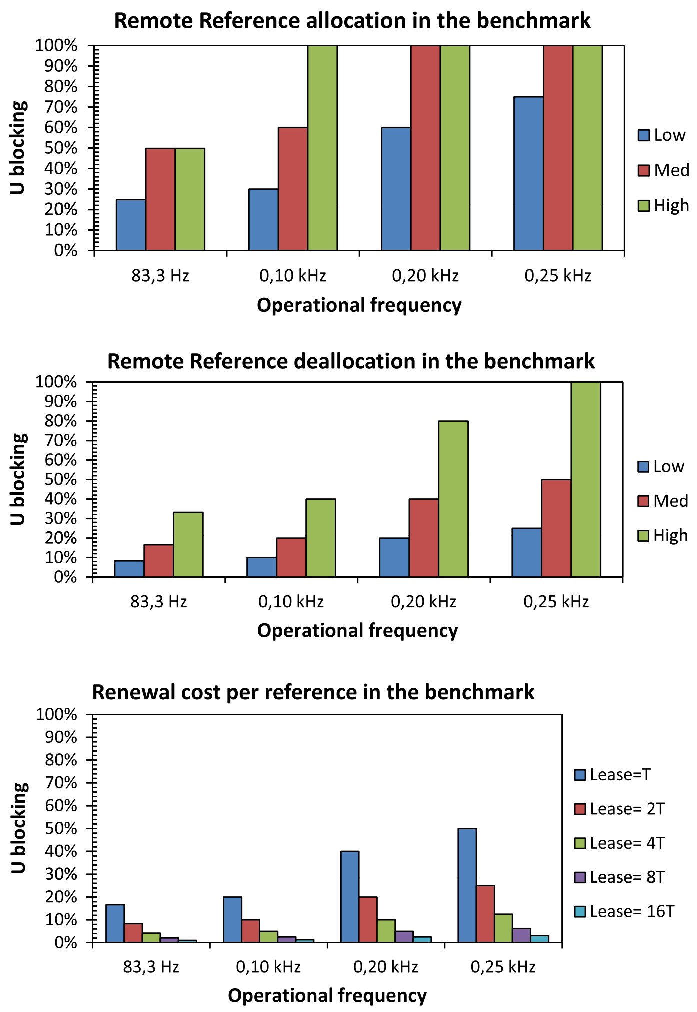

Fig. 17 Summary of results obtained in the benchmark: Two 1 Ghz machines connected with a 100 Mbps Switched Ethernet network. Leasing time multiple of the frequency of the application ($1/T$)

- Results are related to short frequency applications (83 Hz) and `reference` operations. In this case, the overhead of the reference operations is medium to high to be considered as a part of the total cost in the distributed application (minimum costs by 22 % of the available CPU time). This cost increases with frequency, consuming an important amount of CPU (100 % of the available) when the number of allocated references increases from a medium number to a high number.
- Results related to `unreference` operations performance. In general, `unreference` operations take less time than their corresponding reference equivalents. This is the reason why the application computes more unreference commands than reference commands. Reference logic tends to be more complex.
- Distributed garbage collection introduces a moderate overhead. This overhead is minimum when the leasing period is relatively high (i.e. 16 times the period of

the applications) and keeps reduced for shorter periods. For these low frequency periods, the overhead is less than 5 % of the utilization available in the system. In those cases with higher operational frequency (i.e., 0.25 kHz), the utilization takes half the amount of the resources of the application. These results show the importance of selecting a proper leasing period that minimize the overhead in distributed applications.

In general, results showed the importance of selecting in which state of the application a remote reference is allocated. A high number of allocations consume an important amount of CPU. The empirical results also recommend choosing a proper leasing time that does not take a huge amount of time to renew all remote references.

7 Conclusions and future work

Next generation real-time systems demand high-level programming abstractions that reduce maintenance and development costs. As a piece of work that contributes to the improvement of current infrastructures, this article dealt with the problem of having a distributed garbage collector in a real-time Java infrastructure. The work identified the problems in currently implemented approaches and proposed mechanisms to leave out DGC indeterminism, defining extensions to the API and specifying a real-time distributed garbage collector for real-time Java infrastructures.

The simple mechanism is based on two direct methods that allow remote reference allocation and deallocation, and includes also a simple leasing mechanism. All these methods are parametrised as real-time activities. The empirical evaluation showed the penalty introduced by the RT-DGC service in the end-to-end communication paths. Results highlight how choosing a proper leasing time improves performance dramatically.

Our most directed related work refers to connect the predictability of the proposed work with the local predictability offered by local real-time garbage collectors and propose other alternative models for the distributed garbage collector included in RMI.

We also plan to extend the evaluation of the algorithm to other infrastructures. The first piece of work addresses Service Oriented Architectures (SOA), following the strategies described in [8,49,50]. The second set of ongoing work considers the integration of the proposed RT-DGC as building block within the JES framework [50] to improve its flexibility, and specific tools like RESANA [51,52] that could be extended with specific support for RT-DGC.

Acknowledgments The authors have to thank the anonymous reviewers of this article their efforts in improving the quality of this work.

References

1. Rajkumar R, Lee I, Sha L, Stankovic J (2010) Cyber-physical systems: the next computing revolution. In: 47th ACM/IEEE design automation conference (DAC), p 731
2. Kyoungho A, Shekhar S, Caglar F, Gokhale A, Sastry S (2014) A cloud middleware for assuring performance and high availability of soft real-time applications. J Syst Archit. doi:10.1016/j.sysarc.2014.01.009

3. Fisher A, Jacobson C, Lee E, Murray R, Sangiovanni-Vincentelli A, Scholte E (2014) Industrial cyber-physical systems-iCyPhy. In: Complex systems design and management, pp 21–34
4. Liu B, Chen Y, Blasch E, Pham k, Shen D, Chen G (2014) A holistic cloud-enabled robotics system for real-time video tracking application. In: Future information technology, pp 455–468
5. Silvestre-Blanes J, Almeida L, Marau R, Pedreiras P (2011) Online QoS management for multimedia real-time transmission in industrial networks. IEEE Trans Ind Electron 58(3):1061–1071
6. Balani R, Wanner L, Srivastava M (2014) Distributed programming framework for fast iterative optimization in networked cyber-physical systems. ACM Trans Embed Comput Syst 13(2):66
7. Park S, Kim J, Fox G (2014) Effective real-time scheduling algorithm for cyber-physical systems society. Future Gener Comput Syst 32:253–259
8. Garcia-Valls M, Basanta-Val P (2014) Comparative analysis of two different middleware approaches for reconfiguration of distributed real-time systems. J Syst Archit 60 (2):221–233
9. NIST (1999) Requirements for the real-time extensions for the Java platform. http://www.nist.gov/itl/div897/ctg/real-time/rtj-final-draft. Accessed July 2014
10. RTEG (2001) The real-time specification for Java. http://www.rtsj.org/. Accessed July 2014
11. Jensen E (2001) The distributed real-time specification for Java: an initial proposal. Comput Syst Sci Eng 16(2):65–70
12. Locke D, Andersen B, Brosgol B, Fulton M, Henties T, Hunt J, Nilsen K, Schoeberl M, Tokar J, Vitek J, Wellings A (2011) Safety-critical Java technology specification. https://jcp.org/en/jsr/detail?id=302. Accessed July 2014
13. Dibble P, Wellings A (2009) JSR-282 status report. JTRES 2009:179–182
14. Wellings A, Clark R, Jensen E, Wells D (2002) The distributed real-time specification for Java: a status report. In: Embedded systems conference, pp 13–22
15. Anderson J, Jensen E (2006) Distributed real-time specification for Java: a status report digest. In: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, JTRES '06, Paris, pp 3–9
16. Borg A, Wellings A J (2003) A real-time RMI framework for the RTSJ. In: Proceedings of 15th euromicro conference on real-time systems, pp 238–246
17. Basanta-Val P, Anderson J (2012) Using real-time Java in distributed systems: problems and solutions. In: Toledano TH, Wellings AJ (eds) Distributed and embedded real-time Java systems. Springer, New York, pp 23–45
18. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I (2009) Simple asynchronous remote invocations for distributed real-time Java. IEEE Trans Ind Inform 5(3):289–298
19. Oracle (2004) Java remote method invocation. http://Java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0. Accessed July 2014
20. Raman K, Yue Z, Panahi M, Colmenares JA, Klefstad R, Harmon T (2005) RTZen: highly predictable, real-time Java middleware for distributed and embedded systems. In: Middleware, pp 225–248
21. Tejera D, Alonso A, de Miguel MA (2007) RMI-HRT: remote method invocation—hard real time. In: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, Vienna, Austria, pp 113–120
22. Jones R, Lins E (1996) Garbage collection—algorithms for automatic dynamic memory management. Wiley, New York
23. Jones R, Hosking A, Moss E (2011) The garbage collection handbook. The art of automatic memory management. Chapman & Hall CRC, London
24. Birrell A, Nelson G, Owicki S, Wobber E (1995) Network objects. Softw Pract Exp 25(4):87–130
25. Wollrath A, Riggs R, Waldo J (1996) A distributed object model for the Java system. In: 2nd Conference on object-oriented technologies & systems (COOTS),UNESIX Association, pp 219–232
26. Waldo J, Wyant G, Wollrath A, Kendall S (1996) A note on distributed computing. Mob Object Syst 1996:49–64
27. Bacon D, Cheng P, Rajan V (2003) The metronome: a simpler approach to garbage collection in real-time systems. In: OTM workshops, pp 466–478
28. siebert F (2012) Parallel real-time garbage collection. In: Toledano T, Wellings A (eds) Distributed and embedded real-time Java systems. Springer, New York, pp 79–100
29. Robertz S, Henriksson R (2003) Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. LCTES 2003:93–102
30. Baker H (2009) The treadmill: real-time garbage collection without motion sickness. ACM Sigplan Not 27(3):66–70

31. Nilsen K, Schmidt W (1996) US Patent No. 5,560,003. U.S. Patent and Trademark Office, Washington, DC
32. Pizlo F, Petrank E, Steensgaard B (2008) A study of concurrent real-time garbage collectors. SIGPLAN Not 43(6):33–44
33. Kalibera T, Jones R (2011) Handles revisited: optimising performance and memory costs in a real-time collector. ISMM 2011:89–98
34. Masmano M, Ripoll I, Balbastre P, Crespo A (2007) A constant-time dynamic storage allocator for real-time systems. Real Time Syst 40(2):149–179
35. Anderson J, Ravindran B, Jensen E (2007) Consensus-driven distributable thread scheduling in networked embedded systems. In: Proceedings of the 2007 international conference on embedded and ubiquitous computing, EUC'07, Taipei, Taiwan, pp 247–260
36. Basanta-Val P, Garcia-Valls M (2014) A distributed real-time Java-centric architecture for industrial systems. IEEE Trans Ind Inform 10(1):27–34
37. Moreau L, Dickman P, Jones R (2005) Birrell's distributed reference listing revisited. ACM Trans Program Lang Syst (TOPLAS) 2005:1344–2395
38. Plainfosse D, Shapiro M (2005) Survey of distributed garbage collection techniques. Mem Manag 2005:211–249
39. Mohan V (2014) Birrel's distributed reference listing algorithm formalization and implementation. http://www.vikrammohan.com/uploads/GC. Accessed July 2014
40. Klein M (1993) A practitioner's handbook for real-time analysis: guide to rate monotonic analysis for real-time systems. The Kluwer, Kluwer international series in engineering and computer science, New York
41. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T (2009) The worst-case execution-time problem overview of methods and survey of tools. ACM Trans Embed Comput Syst (TECS) 7(3):36
42. Sha L, Abdelzaher T, Årzén K, Cervin A, Baker T, Burns A, Buttazzo G, Caccamo M, Lehoczky J, Mok A (2004) Real time scheduling theory: a historical perspective. Real Time Syst 28 2(3):101–155
43. Tindell K, Clark J (1994) Holistic schedulability analysis for distributed hard real-time systems. Microproces Microprogram 40(2–3):117–134
44. González-Harbour Palencia-Gutirrez J (1998) Schedulability analysis for tasks with static and dynamic offsets. RTSS 1998:26–37
45. Sun J, Gardner M, Liu J (1997) Bounding completion times of jobs with arbitrary release times, variable execution times and resource sharing. IEEE Trans Softw Eng 23(10):603–615
46. Autosar (2012) Release 4.0 overview and revision history. http://www.autosar.org. Accessed July 2014
47. Garcia-Valls M, Basanta-Val P (2013) A real-time perspective of service composition: key concepts and some contributions. J Syst Archit (Part D) 59(10):1414–1423
48. Basanta-Val P, Garcia-Valls M, Baza-Cunado M (2014) A simple data mulling protocol. IEEE Trans Ind Inform 10(2):895–902
49. Garcia-Valls M, Rodriguez-Lopez I, Fernandez-Villar L (2013) iLAND: an enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. IEEE Trans Ind Inform 9(1):228–236
50. Holgado-Terriza J, Viudez-Aivar J (2009) A flexible Java framework for embedded systems. In: Proceedings of the 7th international workshop on Java technologies for real-time and embedded systems, Madrid, Spain, pp 21–30
51. Korsholm S, Sndergaard H, Ravn A (2013) A real-time Java tool chain for resource constrained platforms. Concurr Comput. doi:10.1002/cpe.3164
52. Kersten R, van Gastel B, Shkaravska O, Montenegro M, Eekelen M (2013) ResAna: a resource analysis toolset for (real-time) Java. Concurr Comput. doi:10.1002/cpe.3154