Comparative analysis of two different middleware approaches for reconfiguration of distributed real-time systems

Marisol García Valls and Pablo Basanta Val
Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
Leganés, Madrid, Spain
{mvalls, pbasanta}@it.uc3m.es

Abstract— Software-based reconfiguration of distributed real-time systems is a complex problem with many sides to it ranging from system-wide concerns down to the intrinsic non-robust nature of the specific middleware layer and the used programming techniques. In a completely open distributed system, mixing reconfiguration and real-time is not possible; the set of possible target states can be very large threatening the temporal predictability of the reconfiguration process. Over the last years, middleware solutions have appeared mainly for general purpose systems where efficient state transitions are sought for, but real-time properties are not considered. One of the few contributions to run-time software reconfiguration in distributed real-time environments has been the iLAND middleware, where the germ of a solution with high potential has been conceived and delivered in practice1. The key idea has been the fact that a set of bounds and limitations to the structure of systems and to their open nature needs to be imposed in order to come up with practical solutions. In this paper, the authors present the different sides of the problem of software reconfiguration from two complementary middleware perspectives comparing two strategies built on distribution middleware. We highlight the lessons learned in the iLAND project aimed at service-based reconfiguration and compare it to our experience in the development of distributed Real-Time Java reconfiguration based on distributed tasks rescheduling. Authors also provide a language view of both solutions. Lastly, empirical results are shown that validate these solutions and compare them on the basis of different programming language realizations.

Keywords—reconfiguration; distributed systems; middleware; real-time

I. INTRODUCTION

Information and communication technologies are rapidly driving the road to an enhanced distributed computing paradigm where public, private, individual, and group's resources can all be put in common to provide enhanced processing power in a highly connected environment. In this way, the resources needed and utilized by application users are, in part, hosted remotely and provided from the outside. This is one of the principles of cloud computing that allows to make use of huge amounts of computational and storage resources [1] by means of reduced computing power devices as personal smart phones [2], tablets, etc. Applications and data are run and stored on virtualized resources that are potentially shared with large numbers of other users. At the same time, this brings in interesting properties for companies and society in general, such as energy efficiency and environment-friendly.

The appearance of efficient communications middleware technologies and flexible software paradigms as service oriented architectures (SOA) has been of paramount importance for enabling open and versatile environments. It provides a fertile soil for developing new applications with a decoupled and flexible structure. Still, support for real-time in such domains has not progressed so fast. There are a number of reasons for this. Although real-time research has provided numerous results in scheduling theory, the practical implementation of real-time software systems is still the bottleneck to timing predictability. Specially, the communications middleware level presents numerous challenges that are conceptually simplified in the theory. Middleware is often taken as a black box with a number of needed properties and assumptions that fit well with the real-time abstractions and models. However, in practice, middleware technologies have not completely eliminated the uncertainty brought in by the underlying techniques such as serialization, address resolution, transport and Internet level details, etc. In fact, as they introduce extra logic, they also increase the number of sources of unpredictability.

Providing real-time guarantees in distributed computing environments is a hard problem, and it becomes non tractable if these are considered to be open systems with dynamic behavior. The question is that openness and dynamicity are characteristics of the emerging applications; systems undergo changes that may require a modification of their structure to adequately process and react to events from the environment. In this work, we refer to such changes as reconfigurations, i.e., the process of transitioning from the current system structure (or configuration) to the target one.

For open distributed systems, merging reconfiguration with real-time is not solvable with the available techniques. A set of bounds and limitations to the structure of systems needs to be imposed in order to reduce complexity by means of limiting the size of the space of solutions, i.e., the number of possible target configurations. This is one of the first lessons learned in our previous work in the iLAND project

1 http://sourceforge.net/projects/iland-project/
that resulted in the identification of a set of phases for the complete time-bounded reconfiguration process. We further elaborated on some of the reconfiguration problems in [5] providing a high-level view of these, and showing how to reduce the space of solutions by embedding extra logic in the reference implementation of iLAND [3][6].

A. The Contribution

Different languages such as Ada [27], Java (Real-Time Specification for Java or RTSJ) [28], or C/C++ have been instrumental for implementing distributed real-time systems. The actual distribution middleware for real-time systems such as RT-CORBA [29], Distributed RTSJ [22], DSA[30], or DDS [31] are actually silent about run-time reconfiguration aspects. Even the modeling languages such as MARTE UML[32] do not integrate run-time real-time reconfiguration capacities. They rather support the specification and usage of the basic building blocks that may be reused in other high-level abstractions for specific applications. As a result, these specifications do not provide templates or entities that enable control on dynamic functionality replacement during system operation. Providing practical approaches to real-time reconfiguration is one of the next challenges in real-time systems; in fact, this is a fundamental property of CPS (Cyber Physical Systems).

Only some initial steps have been taken to achieve real-time reconfiguration guided by specific research projects in different domains such as [25] for real-time systems and [34] for mainstream service provisioning. Their efforts in defining simple and reusable patterns and techniques for either real-time reconfiguration or considering a restricted view of timing parameters contribute to a general corpus of techniques that may be used to provide real-time system practitioners with a valuable glossary.

Our contribution subsumes results from two different approaches (initially presented in [5] and [33]) that provide practical real-time reconfiguration solutions embedded in the core of distribution middleware. These are: (1) iLAND, that is an DDS (Data Distribution Service for real-time systems) based middleware enhanced with reconfiguration logic and real-time resource management; and (2) DREQUIEMI, that is a middleware based on DRTSJ – Distributed Real-Time Java). The first presented reconfiguration approach is based on the empirical experience gained in the iLAND project [5] that focused on providing real-time reconfiguration capabilities for service-based real-time applications. The DRTSJ reconfiguration approach presents a Java-based middleware [33] to reschedule distributed task sets by means of Java language templates.

In a previous work ([5] and [33]), we discussed separately the individual problems and solutions to software reconfiguration on each of the two above-mentioned middleware. In this paper, we present an integrated view of them that provides a common corpus of reusable techniques that may be employed in different distributed real-time applications. We present a new comparative view, elaborating a common benchmark that draws interesting conclusions on both reconfiguration approaches and provides useful hints on their behavior and suitability under different conditions.

The rest of the paper describes the two previously introduced strategies under a unified and revised perspective. Section 2 introduces the problems of software reconfiguration. Section 3 presents the different approaches to software reconfiguration from the point of view of the distribution middleware. Section 4 describes system and application models for each of the middleware approaches: based on services or tasks. Also, it presents an overview of the API that shows the middleware realization of the reconfiguration approaches. Section 5 presents empirical results about the reconfiguration models of both middleware approaches and contributes a common benchmark for reconfiguration. Section 6 is the related work, and section 7 draws some conclusions about the work.

II. EXPOSING THE PROBLEMS OF SOFTWARE RECONFIGURATION

Run-time reconfiguration challenges temporal predictability at the different architectural layers. We have investigated different sides of software reconfiguration in distributed systems that have real-time requirements. A general description of the challenges and approaches to reconfiguration is provided in this section.

A. Some reconfiguration challenges

Real-time applications may have different degrees of tolerance to deadline misses. In soft real-time applications, deadline misses result only in a degraded but acceptable operation, e.g., real-time video processing in intelligent cooperative nodes. We target at emerging distributed applications such as CPS where some parts have soft real-time requirements; they must be designed in order to react to the occurrence of events that may require some adaptation (or reconfiguration) of the system in terms of: (i) functional structure where some software parts need to be removed, replaced, or newly added, or (ii) internal processing configuration where the same functionality may continue to be provided but adaptation of the processing parameters is performed. An example of a reconfiguration event is one generated by an unexpected movement in some high-security perimeter as a sign of a possible intrusion detected by a sensor; it may require that a different camera set be activated instantaneously and higher resolution images are captured and sent at the same time.

Reconfiguration events can be triggered either internally, i.e., due to the self monitoring process, or externally, i.e., requested by an external entity such as a user. Moreover, reconfiguration triggers may arrive:

- **Synchronously**: at a specified time when the system is able to process them.
- **Asynchronously**: with an unknown arrival pattern. They can be modeled as a periodic task as well.

To adequately process reconfiguration events, real-time systems require that the transition, either to a new functional structure or a new internal processing configuration, is time bounded. However, there are different important threats to preserving temporal predictability in the presence of...
dynamic behavior. A number of considerations to be made are:

- **Dynamic versus static entity membership.** Systems can have a fixed size or dynamic size. The former does not allow any modification of the software execution units at run-time, whereas the latter does allow the spontaneous dis/appearance of active software units at any time.

- **Size of the space of solutions.** Upon a reconfiguration event, the reconfiguration protocol logic decides the target system configuration to make the transition to. There is a number of possible target configurations, named space of solutions, where the final target configuration is selected from. For a system of dynamic size, the space of solutions can be unlimited; so, there is no feasible solution to derive a time-bounded reconfiguration strategy.

- **Network effects and schedulability.** In a distributed system, the distributed communication and coordination of the entities may incur in extra delays. There are some techniques that confine this problem as the real-time traffic scheduling and the synchronous communications [7]. Some approaches, as the worm whole [8], allot space for both synchronous and asynchronous traffic. The existence of central entities for coordination also limits the unpredictability of the distributed real-time transition.

- **Virtual platforms.** Emerging applications may execute on virtualized environments where hardware is not directly accessible; some parts of the application do not have direct access to the hardware. Also, general purpose hypervisors do not provide basic tools for applications to guarantee temporal predictability and avoid interference among applications and/or virtual machines. In such environments, only performance evaluation can be carried out to assess the level of timing guarantees provided by the specific virtualizer (e.g. [35]). Virtualized real-time environments can be modeled and abstracted properly by means of virtual resources and virtual processors utilizing hierarchical scheduling among partitions, i.e., virtual machines. However, their implementation in practice presents many challenges that are being solved incrementally by some real-time hypervisors. Still, real-time hypervisors are an open area of research to provide efficient implementations of the mechanisms that enable time deterministic execution.

### B. Approaches to reconfiguration

Researchers tackle the problem of reconfigurations from different points of view and backgrounds. Mainly, we find:

- **Software engineering versus performance-based schemes.** The first approach mainly considers the specific programming-level details as the software units (e.g. objects, components, etc.) and their connectors and bindings to make a transition to a new state. On the other hand, performance-based approaches take a more abstract view presenting a formalized system model and the algorithms for selecting the target configurations that meet the given system-level properties, e.g., timeliness, quality values, or interface functionality.

- **Non real-time versus real-time schemes.** Non real-time schemes design efficient transition models mostly disregarding the temporal properties of systems. In real-time reconfiguration schemes, the system must be temporally predictable not only during normal operation but also during reconfiguration; timeliness is central in the system design.

- **Centralized versus distributed schemes.** A centralized perspective does not consider the network effects in the coordination of the reconfiguration; whereas the distributed reconfiguration must account for the schedulability model of the network, the effect of possible communication delays, and the distributed coordination problems.

- **Closed versus open schemes.** In closed schemes, all possible system states and transitions are known a priori; therefore the reconfiguration is performed at a given time or synchronization point, typically when the system is in a safe state. This is the main approach taken by mode changes in hard real-time systems. In open reconfiguration schemes, all possible system states are not known; therefore, extra logic is needed to select the target system configuration or state.

- **Execution entity level versus system-level.** Execution entity level deals with the low-level details of the run-time execution and schedulability by checking the compatibility of the execution model of task sets and deciding the required adjustments to their parameters (e.g., change of activation period, priority, execution time refinements, etc.). System level strategies deal with the selection of the target set of execution units by applying optimization techniques or graph theory and search techniques and come up with target states that preserve the system run-time requirements.

- **Hardware versus software reconfiguration.** Hardware reconfiguration approaches rely on the usage of programmable hardware for providing different computation platforms without the cost of building a new design and development of a hardware setting. In this way, programmable hardware (i.e., mainly FPGAs) transforms into specific computation platforms. Software reconfiguration consists of managing the change of structure of the different possible running software units.

### III. Middleware-based Reconfiguration Approaches

The physical realization of the different approaches to software reconfiguration requires the presence of additional modules that carry out/coordinate the actual transition of the system to a new state. Distribution middleware containing reconfiguration logic is needed to coordinate the reconfiguration in a distributed environment. The middleware must also be very efficient, and its internal structure has to be drawn based on real-time tasks as part of the overall schedulability model of the system.
It must be mentioned that the space of solutions of the presented techniques for real-time systems is bounded, and it is analyzed a priori. Automatic or ad-hoc techniques for this can be employed. Profiling tools are used in the specific case of iLAND (see [6]) which are publicly available.

The two middleware approaches that we present have a number of differences that relate to the presence of: (1) support for dynamic execution, i.e., the system can change at run-time in bounded-time; (2) self detection and autonomous handling of reconfiguration triggers (3) an algorithm for the run-time selection of the new system state; (4) remote deployment of tasks to nodes.

Based on these considerations, the two approaches to reconfiguration are classified as follows:

- **Reconfiguration arbiter and transition coordinator.**
  The reconfiguration protocol is executed at run-time, and it arbitrates the whole transition process, e.g., re-scheduling of execution units such as tasks or services, determining the new state of the system, rebinding the connections or dependencies between tasks (typically, this sets the execution order), and execution of mode change. Execution units are typically assigned to a node at system start up and different versions/implementations are available. Execution units have an interface that contains both functional and non-functional properties; such information is used for determining the new system state. Therefore, this middleware contains the above-mentioned characteristics (1) through (3).

- **Distributed task scheduler.** This is a lower abstraction level approach supporting off-line reconfiguration in the sense that tasks are rescheduled and remotely deployed prior to the system start up. It handles execution units which are typically tasks with non-functional parameters related to timing properties. The schedulability analysis resides on an entity that is centralized in a particular node with the knowledge of all tasks'properties. It can be linked to an enhanced logic/unit that automatically deploys the schedulable tasks in the distributed nodes. This middleware contains characteristic (4). (1) through (3) can be executed off-line.

Following, both approaches are presented.

A. **iLAND middleware: communications middleware and reconfiguration arbiter**

In the software engineering domain, middleware solutions supporting software reconfiguration refer to the capacity of the middleware itself to modify its structure (typically based on components) to suit the needs of a specific deployment with respect to the resource limitations, needs of the running software stacks, component compatibility requirements, etc. This is the case of some approaches as [9]. On the other hand, other middleware approaches support the reconfiguration of the application-level software entities; this is typically applied in the recent years to SOA-based applications although existing solutions are silent about the temporal constraints of the reconfiguration. The first middleware to merge real-time execution and application reconfiguration for real-time service-based applications is iLAND. Depending on the needs of the specific domain, the flexible iLAND architecture (see figure 1) can be implemented to provide a range of temporal guarantees, from QoS-based execution to hard real-time guarantees.

iLAND reference implementation is based on a DDS middleware backbone, that provides QoS communication parameters for setting the temporal values of the distributed interaction. Also, iLAND provides a hard real-time network stack for distributed communications that make use of time triggered communication.

![Fig. 1. iLAND middleware Architecture](image)

The two main functionalities of iLAND are:

- **Real-time communication.** This is managed by the Communication Backbone and Resource Management Layer (CBL). This layer contains the component Comm. Middleware that provides a bridge for using existing or off-the-shelf middleware that adjusts to the specified component interfaces as principal communication backbone. Solutions as DDS (Data Distribution Service for Real-time systems) can be used to provide QoS-level guarantees in the communication. The CBL layer contains the component named Custom Protocol Stack for customized real-time network scheduling for strict temporal guarantees; in iLAND, time triggered networks can be used, controlling the media access and avoiding packet collisions to achieve temporal isolation.

- **Real-time reconfiguration management.** The logic that supports the reconfiguration protocol is embedded in the Core Functionality Layer (CFL). The two basic components in this layer are the Control Manager and the Composition Logic that performs active coordination of the reconfiguration protocol arbitrating the different steps to be taken in bounded-time.
B. DRTSJ: communication middleware and distributed task scheduler

The proposed distributed task scheduler is described for DREQUIEMI [36]: a framework for distributed real-time Java including an API [37][38]. The extensions and techniques described here may be included without major changes into DRTSJ specification, which is still an open issue in the real-time Java community.

DREQUIEMI (see figure 2) divides the system distribution infrastructure into a set of layers that contribute to hide distribution details. The model identifies three main resources that can be managed, namely: memory, CPU, and the network.

These resources are accessed through an infrastructure middleware supported via a real-time Java virtual machine. On top of the infrastructure middleware, two new layers (distribution and common services) are in charge of providing standard services for distribution. The list includes a stub/skeleton in charge of real-time invocations ([37][38]); a DGC for distributed garbage collection; a naming service for a white page service, among others.

Globally, three different managers control resource allocation via: (i) a distributed memory manager; (ii) a distributed processor manager; and (iii) a distributed connection manager. Distributed memory managers, distributed processor manager and distributed connection manager allow configuring and managing the resources involved in a remote invocation to a remote object. Distributed managers are pluggable entities that offer different policies for each application domain. Typically, the algorithms included in distributed managers refer to: (1) memory used for the remote object up-calls that could be regions or real-time garbage collectors; (2) the priorities used in the up-call to the remote object that could be propagated from the client and defined by the server; and (3) a distributed connection policy that could allow connection pre-allocation strategies and other QoS network management policies.

On top of common services, there is an application subsystem that divides applications into a set of reusable components.

In this real-time Java architecture, there are three natural reconfiguration points:

- The first is placed at the infrastructure level, as a means to control memory, processor, and network locally.
- The second extends this control to a distributed level scheme and it is placed at the distributed manager that controls distributed applications performance.
- The third level of control is at component level and may perform adaptations taking into account the components of an application.

The distributed reconfiguration service analyzed here is at distribution level (highlighted in figure 2).

![Fig. 2. Reconfiguration in DREQUIEMI middleware](image-url)

Adaptation is addressed at distribution level by controlling resources in the remote nodes. Internally, it uses the resource manager included in each node. It also offers support to the component manager defined at the application level.

IV. TWO REAL-TIME RECONFIGURATION MODELS AND APIs

This section explains the computational model and constraints of the iLAND reconfiguration arbiter and the distributed task scheduler of DREQUIEMI. It also describes their APIs for real-time reconfiguration. In the case of iLAND this API is not directly used by the application, whereas in DREQUIEMI it may be a fundamental part of the application that uses this template to offer new functionality.

A. iLAND: Service-based reconfiguration

iLAND views applications as extended functionality made by the aggregation of a set of services that are the reconfiguration units. Services are self-contained software functionality pieces that are realized by concrete and specific service implementations. A service implementation is an active software entity that provides the functionality of the specific service. Service implementations communicate among themselves only via message exchanges.

1) System model

An application \( a_i \) has different possible realizations, \( a_{i,j} \), \( \{ a_{i,j} : j = 1,2,... \} \) or internal structure possibilities. In the same way, a service \( s_i \) can have different realizations \( \{ s_{i,k} : k = 1,2,... \} \). Also, \( a_i \) has a specific set of services \( S_i = \{ s_j \} \).
As a result, a reconfiguration is the replacement, removal, or addition of one or more service implementations. **Reconfiguration** refers to a change in the structure of the active software units that are part of an application. In general, a service can contain a number of threads or tasks; in our model, we consider single-task services. A reconfigurable system or application \(a_i\) is, therefore, a superset of \(n\) service implementations which are, in the end, executed by tasks \(\tau\), i.e., \(a_i = \{\tau_1, \tau_2, \ldots, \tau_n\}\).

A reconfiguration occurs when the system or application configuration \(a_{i,j}\) is replaced by another one \(a_{i,k}\), such that the target service implementation set is not equal to the original \(S_{k\text{init}} \Rightarrow S_{k\text{target}}\). Here, we use the terms system and application interchangeably. In a real-time environment, the time \(t'\) taken to complete the transition from \(S_{k\text{init}}\) to \(S_{k\text{target}}\) has to be bounded.

In the specific case of a real-time distributed system, the effect of the operating system and middleware threads should not be neglected. In our model, we account for such a structure. A system state is made of a set of threads that are scheduled by the underlying infrastructure. Depending on the type of environment, the underlying infrastructure may refer to the kernel and/or the JVM, the latter in the case of Java applications. The threads of the initial state are:

- \(as_i = \{\tau_x: x=1,\ldots,n\}\); where \(n\) is the number of service implementations that are running. This set is the “payload” of the system; these tasks are functionality providers for users.
- \(os_i = \{\tau_i: i=1,\ldots,m\}\); where \(m\) is the number of operating system tasks that are providing the basic execution services and environment. This is the actual run-time services.
- \(mw_i = \{\tau_z: z=1,\ldots,p\}\); where \(p\) is the number of middleware tasks that are actively providing the communications logic and the extended functionality that is typically contained in it (e.g., thread pools).

A service implementation \(s_{i,k}\) is specified by its functionality \(f_i\) (all \(s_{i,k}\) have a common \(f_i\)), execution time \(C_{i,k}\), release period \(T_i\) in our real-time model (see [10]), execution priority \(P_{i,k}\), response deadline \(D_{i,k}\), the quality value it offers \(Q_{i,k}\) (an application-related utility value), and the \(A\) is the dependency list with respect to other service implementations.

Services are also considered to be activated by periodic messages, and they can be scheduled according to a utilization based model [13] that also accounts for the operating system and middleware tasks. All tasks in our model are contemplated in the schedulability analysis of the system. Therefore, the whole task set to be considered in the analysis is \(\{as_i, os_i, mw_i\}\) with different priority level values that determine the pre-emption and execution mode privileges. Priority assignment also plays a major role; we use a simple priority assignment technique based on priority bands [14].

2) **Reducing the complexity**

An additional consideration on complexity reduction is required in the specific case of iLAND. It is impossible to achieve real-time behavior in the reconfiguration of a completely open distributed real-time. In iLAND, we require that systems are checked a priori to determine the size of the solution space and schedulability of their solutions. This is done by means of a priori system profiling tools [3][4] for the pre-visualization of the different possible system states and transitions.

Figure 3 illustrates the high-level view of the reconfiguration logic aiming at reducing the complexity of the solution space.
Following, figure 5 shows a synthesized API for the reconfiguration service of iLAND.

![Fig. 4. iLAND components that encapsulates the middleware enhanced logic](image)

Figure 5 shows an overview of the API containing the main functions that are direct part of the reconfiguration process. These functions are contained in the two main components for the reconfiguration logic: composition logic and control manager. iLAND is coded in C language and, for the sake of clarity, each component starts with their designated initials that are cl and cm, respectively. The naming structure of components is inspired in HOLA-QoS [17].

```plaintext
01: //Control manager
02: 03: int cm_set_reconf_policy(int reconf_code);
04: int cm_execute_reconf(rec_param_type reconf_params);
05: int cm_reconfigure(id_type appid, int no_nodes, int no_slides);
06: int cm_exec_mode_change(id_type app_id, graph_type dst_graph, graph_type src_graph); 07: 08: //Composition logic
09: 10: int cl_set_compos_alg(int compos_code);
11: int cl_select_compos_alg(int compos_code,
compos_param_type compos_params);
12: int cl_exec_compos(graph_type expanded_graph);
```

Fig. 5. Overview of the reconfiguration API of iLAND

The middleware allows users to provide different reconfiguration policies (e.g., centralized, distributed, or hybrid). Function cm_set_reconf_policy sets the specific policy for next reconfiguration; then, cm_execute_reconf will launch the coordination of the reconfiguration process. Eventually, a transition to the target state selected by the composition algorithm will take place; this is carried out by invoking cm_exec_mode_change. The composition logic is in charge of selecting the target state system from the graph of possibilities. iLAND bounds the temporal cost of this process with an off-line application profiling study. Different composition algorithms can be used as explained in [4]; therefore, function cl_select_compos_alg allows to select them. If a specific single composition algorithm is enforced in the system, function cl_set_compos_alg is used. During the reconfiguration process the Control Manager component will execute the composition algorithm using function cl_exec_compos.

B. DRTSJ: Task-based rescheduling

1) System model

The default reconfiguration strategy proposed for distributed real-time Java (see figure 6) is based on the end-to-end flow-shop model previously defined in Sun [40], Tindell [41], and Palencia [42]. In this model, an end-to-end transaction is as a sequence of tasks that execute in order (i.e. with precedence constraints).

Each end-to-end transaction $T^j$, where $j \in 1..M$, is defined as having: a global deadline ($D^j$), a global period ($P^j$), and a set of $n$ schedulable segments $\{S^j_i, \ldots S^j_n\}$. The schedulable segments have:

- A local execution priority ($P^j_n$) for each end-to-end transaction segment ($S^j_i$).
- A local worst-case execution time ($C^j_i$) for each segment ($S^j_i$).

In addition, each node runs a periodic enforcer [43] that allows analyzing each flux in a node as an independent task.

This constrained real-time scheduling theory has been integrated in a reconfiguration process that deploys the code on other real-time Java remote nodes.

![Fig. 6. Flow-shop model used in the default reconfiguration policy defined for distributed real-time Java](image)

This default reconfiguration process, shown in figure 7, consists of a set of steps. First, it checks the feasibility of the system by using an independent task analysis (step 1 in figure 7). Then, the algorithm selects the remote node in which the remote object is to be allocated by sorting all segments (according to their utilization $C^j / T^j$), to be assigned to different nodes using an assignment policy. The assignment policy takes care of being under the rate monotonic utilization bound in each node. After that, each segment is assigned a priority that depends on its period. The last step (5 in figure 7) in the process corresponds to the mechanism in charge of deploying tasks in remote nodes.
2) Software elements and API

Figure 8 shows the API changes introduced to accommodate the reconfiguration service in the DREQUIEMI ecosystem. These modifications include new APIs for centralized and distributed real-time Java.

In centralized Java, the main extension required is network support. Currently, RTSJ, the main approach in real-time Java, does not take into account the existence of different types of networks. In the proposed support for reconfiguration, two new elements are included to consider the network. The first is a characterization for networks similar to memory parameters and scheduling parameters currently included in RTSJ via a tagging interface (NetworkParameters). All network classes include a tagging interface that extends scheduling parameters list with network parameters. Network parameters are domain dependent (e.g., in [44] the authors used a non-preemptive model with network release times, and scheduling parameters to model a prioritized IP-based Switched-Ethernet). The possibility of additional networks is added via a new scheduler template named 3ResourcesScheduler.

Then, the distributed scheduler API extends the API scheduler interface of RTSJ to be accessible as a remote object via DRTSJ. This simple mechanism is enough to allow the allocation of schedulable entities from another remote node. In all this code, all parameters of the remote object have to be serializable in order to be able to be transferred via the network.

From the point of view of the definition of a reconfiguration service ReconfigurationService shown in figure 9, the approach analyzed in this article is simple. It is based on modeling the distributed scheduler as a schedulable object. This simple approach enables a mechanism to define generic real-time parameters for the reconfiguration service.

**Fig. 7. Reconfiguration process steps for DRTSJ**

**Fig. 8. Class hierarchy related to the real-time reconfiguration in distributed real-time Java**

```java
01: public interface ReconfigurationService extends
02   DistributedScheduler, Schedulable{
03:  MemoryParameters ... RemoteException;
28:  void setSchedulingParameters(
29: SchedulingParameters sp) 
30: throws RemoteException;
31:  }
```

**Fig. 9. API interface for the reconfiguration service in real-time Java**

```java
01: interface FlowShopSchedulables extends Schedulable{
02:  Vector getAllShedulable(); //It returns all sch . 
03: }
```

**Fig. 10. API for a simple reconfigurable flow-shop strategy in real-time Java**

By using this simple approach that models a scheduler as a real-time Java task, the API of the reconfiguration can be
bounded-time. The resulting reconfiguration service offers mechanisms to limit the amount of CPU, memory, and bandwidth in the network required for the reconfiguration process. Furthermore, the reconfiguration process could be globally modeled as a periodic or sporadic process which may be scheduled with the remaining tasks in each remote node.

Lastly, it should be noticed that, by default, there is not a dominant reconfiguration algorithm or technique. This is a distinctive feature of the scheduler of RTSJ. In both cases, the idea is that different algorithms may extend the basic tasking template to include system requirements. A developer may also extend the basic reconfiguration policy to take into account different application domains.

The previous flow-shop reconfiguration strategy would require the definition of two new classes: FlowShopSchedulables to map the flow-shop task model and PriorityBasedReconfigurationService for the reconfiguration service as can be seen in figure 10.

V. PERFORMANCE EVALUATION

A crucial factor in real-time reconfiguration is the time taken to perform the transition from the initial state to the target one. This time depends on different software stacks (i.e., the middleware used in the evaluation) and the functionality offered by the reconfiguration logic. In the algorithms previously described for iLAND and DREQUIEMI, the reconfiguration time depends on the number of schedulable entities that have to be reconfigured and the steps of the reconfiguration process itself.

This empirical section provides evidence on the performance trends of the two proposed reconfiguration strategies. The strategy offered by iLAND is more powerful and can be executed at run-time, whereas DREQUIEMI offers a simple off-line reconfiguration based on rescheduling of tasks. We provide a benchmark for real-time reconfiguration that may be used to compare the performance of different strategies in different scenarios that suit the needs of different application domains.

A. Reconfiguration costs in iLAND-RI

iLAND-RI [6] is the reference implementation of iLAND that is available as an open-source project. It has a quasi-linear dependency on elements involved in the reconfiguration process and the cost of the reconfiguration (see figure 11). In total, each reconfiguration action consumes less than 2.57 ms in 1 Ghz machines connected via a 100 Mbps Ethernet local network.

This time value includes the following tributaries:

- The time required by the selection process that decides among the different options by using a simple heuristic.
- The time required for notifying the reconfiguration trigger to the remote nodes.
- The time required to perform the mode-change at the remote nodes.

In addition, this reconfiguration time does not include cost of the schedulability tests that are carried out by an off-line tool.

![Fig. 11. Cost of the reconfiguration in iLAND-RI and DREQUIEMI. Dependency with the number of reconfigurable entities (1 Ghz machines – 100 Mbits Ethernet)](image)

B. Reconfiguration cost in DRTSJ

A similar experiment was carried out in DREQUIEMI on a 1 Ghz stack on a 100 Mbits Ethernet network. The results (see figure 11) show that DREQUIEMI may offer a slightly better performance than iLAND-RI. In the analyzed cases, the reconfiguration of each remote reconfigurable element may take less than 2.17 ms.

This time includes the following actions (shown in diagram of figure 7):

- The time required to test if the distributed system is schedulable or not and to select a proper remote node in which to deploy the schedulable entity.
- The time required to deploy a schedulable entity in a remote node.
- The time required to setup the schedulable entity in the remote node.

A detailed analysis of the different tributaries in the case of real-time Java was carried out. The tributaries refer to the cost of the feasibility algorithm, and the remote allocation of remote entities.

The first analyzed cost is the feasibility cost. This cost refers to the following three combined actions:

- The cost of deciding if the reconfiguration is feasible or not,
- The time required to sort the tasks,
- The time required to select the physical nodes (i.e., on which JVMs) where tasks will be allocated.

The results show that the system is able to decide about the feasibility of the task set in less than 1.2 seconds for a 4096 schedulable entities to be allocated to 16 different Java virtual machines (see figure 12). They also show that the feasibility cost increases with the number of tasks (in the 64 tasks to 4096 tasks range) and the number of virtual machines in the system (that ranges from 2 to 16).
Fig. 12. Cost of executing the feasibility algorithm with the number of remote nodes (JVMs) and tasks. (1 Ghz machines with DREQUIEMI)

The second experiment refers to the cost of deploying remotely by using the Java remote method invocation model. Results, shown in figure 13, reflect:

- A linear cost behavior with respect to the number of reconfigurable entities that are allocated and the total cost of the reconfiguration.
- A dependency in cost with respect to the type of real-time Java schedulable entity that it is allocated (i.e. an event-handler, a remote object, or a real-time Java thread).

Fig. 13. Remote allocation of entities in a remote node with real-time Java. (1.0 Ghz machines with DREQUIEMI)

C. Application Benchmark

The last part of the empirical evaluation contributes a benchmark designed with the goal of providing a general framework to compare the performance of different reconfiguration techniques. The resulting benchmark is inspired in the operational deadlines defined within the iLAND project [3].

The benchmark (see Table I) defines a maximum and minimum number of transactions that may be reconfigured (Trans). Each transaction has a global period (T) and a number of tasks (also named segments) that may be executed with strict precedence sequence. Each task of each transaction takes from 5% to 30% of the local utilization. The benchmark includes from 1 to 64 different execution nodes (running the reconfigurable middleware). Lastly, the reconfiguration process is modeled as a periodic task with a minimum period ranging from 10 ms to 16 min.

Table I: Reconfiguration benchmark. Main parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Reconfigurable Transactions (Trans)</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>Tasks (or segments) per transaction ($)</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Cost per task (or segment) (%)</td>
<td>5%</td>
<td>30%</td>
</tr>
<tr>
<td>CPU units</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>Reconfiguration Transactions</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Reconfiguration Periods (Tr)</td>
<td>10 ms</td>
<td>16 min</td>
</tr>
</tbody>
</table>

The benchmark defines three different reconfiguration zones depending on the specific time required for the whole system reconfiguration:

- **Short reconfiguration deadlines** where the system has a reconfiguration period of 10 ms.
- **Average reconfiguration deadlines** that have operational reconfiguration periods in the range of 10 s to 10 ms.
- **Large reconfiguration deadlines**, in which the reconfiguration time is the order of minutes (>10 s) and typically corresponds to reconfiguration times that may be easily handled by the middleware.

In addition to the three different reconfiguration zones, the benchmark defines a reduced, moderated and high number of reconfigurations:

- The **reduced number** of reconfigurations refers to a scenario with a maximum number of up to five reconfigurable entities. The type of application that may be modeled following this approach is simple video surveillance applications.
- A **moderated** scenario refers to a system that has to manage up to 50 reconfigurable nodes.
- The last scenario (**complex**) refers to scenario with a high number of nodes that have to be reconfigured at the same time. The typical number of nodes is 500 and is intended to represent medium-size reconfigurable sensor networks.

1) Evaluation of the two different reconfiguration strategies in the real-time reconfiguration benchmark

The aforementioned benchmark was used to test the coverage given the two proposed reconfiguration strategies in a common set of scenarios. The results obtained after the evaluation are summarized in figure 14.
Fig. 14. Benchmark results for iLAND-RI and DREQUIEMI (1 Ghz and 100 Mbits Ethernet). Reconfiguration deadlines range from micro (<10 ms) to medium (10 s) and long (2 minutes) reconfigurations. The number of elements that have to be reconfigured ranges from 5 to 500 reconfigurable elements.

They show the following performance patterns:

i) DREQUIEMI and iLAND-RI offer a similar performance (DREQUIEMI outperforms iLAND-RI slightly) in terms of amount of time required to perform the reconfiguration.

ii) In heavy scenarios (high number of reconfigurations and short reconfiguration times), none of the real-time reconfiguration techniques is able to meet the deadlines of the whole reconfiguration process.

iii) On large reconfiguration processes (deadline >2 minutes) the reconfiguration process is very light; it requires no more than 5% of the total time to perform a system reconfiguration.

This last type of pattern is ideal from the point of view of real-time reconfiguration, because the system may offer real-time reconfiguration at a low computational price. This scenario also encourages the real-time system engineering to look for other reconfiguration policies and algorithms that may offer similar performance.

As summary, more simple reconfiguration scheme (as the one provided by DREQUIEMI) provides slightly better performance. Nevertheless, iLAND provides full run-time reconfiguration in a distributed system that outperforms the simple functionality provided by DREQUIEMI at a slightly higher cost due to a thorough architectural work. As a result, we have given a comparative analysis to derive the best strategy to be used according to the needs of the target application.

VI. RELATED WORK

During the past three decades, dynamic software reconfiguration has been studied by different communities, mainly software engineering, distributed systems, and later real-time systems. Software engineering approaches such as [15] studied the whole process of supporting software updates by providing languages and supporting tools, such as CONIC, for dynamic linkage of code modules in distributed systems. This is nowadays a very limited approach since current paradigms like SOA are not compatible with such tightly coupled solutions where code modules where restructured based on interface compatibility. Later contributions tried to overcome this restriction by using components with QoS execution properties such as [16], and recently others looked into the adaptation mechanisms required to preserve the system service values [34]. However, they were silent about the run-time reconfiguration process itself and not concerned about the time-bounded property of the transition.

Over the years, reconfiguration policies and mechanisms have been developed for real-time domains initially for centralized embedded systems. Some of these solutions aimed at achieving cost-effective hardware utilization developing resource management techniques for adapting the resource assignments to applications tasks according to their instant computation needs; these are the cases of HOLA-QoS [17], specific resource management for ambient intelligence real-time applications [18], or dynamic resource management based on efficient mode changes [11] and priority assignment schemes [14]. The system, in these cases, detects environmental or self monitored conditions adapting its structure accordingly. These concepts introduced by HOLA-QoS for multimedia embedded consumer electronics (i.e., centralized systems) were latter re-worked by some research projects such as Frescor [49] and Ocera [50].

In a general sense, two types of reconfiguration approaches are typically faced from the distributed real-time systems arena: network-level providing traffic scheduling integrated with task scheduling at the nodes or real-time middleware approaches that integrates the real-time traffic communications in a layer that provides higher abstraction
primitives for communication and configuration of interaction parameters in the remote nodes.

In distributed systems, the local and remote tasks as well as the messages over the network must be properly scheduled and synchronized to meet the deadlines of the application. This is a challenging problem since the future arrivals of tasks and requests are typically not known in a complex open system. Some approaches only consider synchronous communication real-time networks, as [7], which eases the schedulability analysis and confines the temporal cost. Dynamic scheduling was also addressed by some contributions that provide guarantees to the distributed processes communicating via synchronous primitives, combining off-line and on-line scheduling [19].

In distributed environments, the popularization of communication middleware, such as CORBA [20] attracted a number of efforts to support dynamic configuration. The subject of adaptation was, however, the middleware itself; later, some additional contributions, e.g., CIAO, allowed some level of changes in the system using specific QoS policies. Still time-bounded transitions were out of the picture.

Recently, researchers working in RT-CORBA have described a distributed real-time manager [23][25] that connects different ORBs with a task manager in charge of allocating components to ORBs and performing load balancing and admission control. The described real-time reconfiguration service is closer to their model, i.e. both carry out admission control dynamically.

Despite all efforts, middleware is still perceived by the real-time community as a source of unpredictability, containing yet another extra layer of code that may decrease the robustness of the system. Often middleware is seen as a black box over which only performance tests are carried out. However, the appearance of middleware technologies such as DDS, ICE, or (previously) RT-CORBA has provided a powerful, flexible, and versatile development push. One of the most interesting characteristics is the abstraction of the network details to the business-level logic. However, when providing real-time guarantees, programmers must specify the required application properties in the pertinent middleware hooks that must be traceable across the software layers. In domains such as grid, this has been tackled by some contributions integrating QoS in the service level agreements [24], but only from a design point of view without actual real-time scheduling support. Other approaches carry out performance evaluations to further abstract these distribution technologies, e.g., DDS, to virtualized environments [35].

The problem of combining distribution middleware and real-time is even harder if we try to support dynamic reconfiguration. Some work is already available although it typically targets mainstream distributed systems. In [48], an algorithm for dynamic reconfiguration of applications is provided that keeps the structural integrity and system states consistency, but it is silent about timeliness. Other approaches, mainly focus on the reconfiguration of the software components at the level of binding and re-binding of the connectors such as [26].

More recently, the iLAND middleware [3][5] has appeared as a framework to support time-bounded dynamic functional reconfiguration of distributed real-time applications. It stands out the work carried out in real-time service composition, [4], and QoS models for supporting composition, [39], which deals with the problem of dynamic allocation of services with multiple available implementations and in dynamic component-based reconfiguration [47]. The authors addressed the problem of finding a suitable system configuration with composition algorithms, which contain heuristics and figures of merit. The use of these techniques allows reducing the overhead of executing these algorithms several orders in magnitude. The proposed real-time reconfiguration service could be extended with these parameters. However, other approaches to reconfiguration that provide a per-request scheduling of distributed tasks are also needed in real-time systems. This is the case of a light extension provided for DRTSJ based on the DREQUIEMI approach [33][38]. Off-line reconfiguration provides flexibility at a reduced cost.

In this paper, authors have presented their experience after several years of working on real-time reconfiguration in the context of the iLAND project by combining scheduling and resource management with middleware technology design. As a result, authors have provided real-time communication middleware enhanced for arbitration of reconfigurations and dynamic task re-schedulers. The comprehensive reconfiguration framework of iLAND that includes a powerful coordination mechanism has been compared against a simplified reconfiguration based on task rescheduling provided by DREQUIEMI [21] for off-line deployment. The problems identified during this time have been presented as well as their major achievement in dynamic reconfiguration. iLAND middleware that is efficient in reconfiguration providing reduction of the solution space. It is compared to the distributed scheduling services over the real-time Java-based DREQUIEMI middleware.

VII. CONCLUDING REMARKS

Designing and implementing real-time systems with some enhanced techniques that run-time reconfiguration is one of the challenges ahead in real-time systems. These techniques may offer a tactical advantage to the industry of applications interested in providing next generation real-time reconfigurable applications. Currently, this technological challenge is still not well understood and there is a general lack of techniques that provide generic support for real-time reconfiguration in all hardware-software levels.

The specific contributions carried out in this paper are two different strategies for the reconfiguration of two different distribution middlewares. The first technique is focused on the context of real-time SOAs reconfigurations, and the second in the context of distributed real-time Java task scheduling. Each one of them presents its own task model and specific reconfiguration support to offer predictable reconfiguration.

Globally, the outcomes have shown that the reconfiguration challenge involves dealing with compromise
solutions that provide attractive strategies for reconfiguration that may be integrated in a real-time ecosystem. Also, they have shown that real-time reconfiguration means new models that allow some type of constrained reconfiguration. In the empirical plane, they have illustrated the deadline reconfiguration cost; in several cases real-time reconfiguration may be offered with a reduced overhead. Lastly, the evaluation has shown that the performance of both techniques may offer response-times lower than 3 ms per reconfigurable entity in a local area network.

Much work is still pending to achieve a mature technology for real-time reconfiguration. Currently, the authors explore different approaches to enhance their basic reconfiguration strategies with new characteristics and features. The authors are currently working on introducing hierarchical scheduling techniques as defined in [45] within their basic reconfiguration strategies. Also, they are working on the concepts of federation, fault tolerance, and cooperative scheduling [46] within a real-time reconfigurable ecosystem. Their collective results will help improve current state-of-the-art technology with a corpus of new techniques for real-time reconfiguration.

ACKNOWLEDGEMENTS

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.ieland-temis.org), ARTISTDesign NoE (IST-2007-214373) of the EU 7th Framework Programme, and by the Spanish national project REM4VSS (TIN 2011-28339).

REFERENCES


