SPECIAL ISSUE PAPER

A DDS-based middleware for quality-of-service and high-performance networked robotics

Jesús Martínez Cruz*,†, Adrián Romero-Garcés, Juan Pedro Bandera Rubio, Rebeca Marfil Robles and Antonio Bandera Rubio

Dpto. Lenguajes y Ciencias de la Computación, Dpto. Tecnología Electrónica, University of Málaga, Spain

SUMMARY

Social robots must adapt to dynamic environments, human interaction partners and challenging new stringent tasks. Their inner software is usually distributed and should be designed and deployed carefully because slight changes in the robot’s requirements can have an important impact not only on the existing source code but also on the resulting performance at run-time. This paper proposes an implementation of this inner software using a new lightweight middleware for networked robotics called Nerve. The principal novelty this middleware has with respect to other state-of-the-art approaches is that it guarantees both scalability and QoS, which are key requirements for real-time robotics software. The benefits of Nerve have been proved through its use in two key components of the cognitive system of a social robot: (i) the visual attention mechanism, used to extract relevant data from perceived images; and (ii) a robot learning by imitation control architecture that allows the social robot to be taught by people using natural demonstrations (i.e. using the same communication channels that would be used to teach people). Nerve makes use of existing patterns for networked applications together with the recent Data Distribution Service specification, which is a publish/subscribe standard for real-time and distributed systems that provides a wide set of QoS policies. In this paper, these different QoS policies have been applied carefully to achieve the best performance of the target robot. Copyright © 2012 John Wiley & Sons, Ltd.

Received 24 September 2011; Revised 5 December 2011; Accepted 14 December 2011

KEY WORDS: robotics; middleware; data distribution service; performance; QoS

1. INTRODUCTION

Social robots are designed to cooperate with people in solving tasks in daily life. This implies that these robots have to work in challenging, uncontrolled environments. Thus, even social robots that are restricted to indoor scenarios have to deal with light variations, structural changes (e.g. moving objects), unknown people, variable interaction distances and partial perception (i.e. occlusions). Social robots should extract useful information from these dynamic environments, while they provide people with natural and intuitive interaction channels and adapt to different tasks and situations [1]. These robots have to use complex cognitive systems to meet these stringent requirements [1]. Two main parts of these cognitive systems are the perception component and the learning component.

Perception components are needed by the robot to perform autonomous tasks such as navigation, mapping or grasping specific objects. Developing systems that allow the robot to focus attention on the same phenomena (i.e. objects, people, etc.) as its human partners is also a critical step in designing social robots that cooperate with people, and that learn from natural human instruction.

*Correspondence to: Jesús Martínez Cruz, Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain.
†E-mail: jmcruz@lcc.uma.es

Copyright © 2012 John Wiley & Sons, Ltd.
Whereas other sensory inputs are also used, the main sensor these perceptual systems rely on is vision, as this is the main perceptual channel people use. Thus, in the last few years, the emphasis has been increasingly on the development of robot vision systems according to the model of natural vision, because of its robustness and adaptability. These systems are inspired by the concept of visual attention, which is the process that filters out irrelevant information and limits processing to items that are relevant to the task in hand. These models are particularly useful for social robots, as they need to filter relevant data from the huge amount of information provided by a sequence of images [2]. Thus, visual attention mechanisms for social robots aim to provide efficient human-like perception that can be used to facilitate sharing attention between the robot and its human companions at run-time [3].

Increased perceptual ability is one of the main requirements for social robots. The other is learning. Social robots cannot predict all the possible situations they will face during their working life. Thus, they have to be provided with mechanisms to adapt to dynamic environments, different people and a huge variety of tasks. Learning allows them to achieve adaptation and thus it becomes a key topic for social robots. There are different options to achieve learning. Individual learning mechanisms (e.g. trial-and-error, imprinting, classical conditioning,...) can be useful for autonomous agents. However, their application to a social robot may lead it to learn incorrect, disturbing or even dangerous behaviours; thus, they should be restricted to specific scenarios and tasks (e.g. games based on controlled stigmergy) [1, 2]. Social learning mechanisms are a different option that avoids most issues of individual learning, as these mechanisms allow the human teacher to supervise the learning process. Whereas there are different social learning strategies, learning by imitation appears as one of the most intuitive and powerful ones. Thus, in the last decade, much learning by imitation architectures for social robots or Robot Learning by Imitation (RLbI) architectures have been proposed to provide social robots with the ability to acquire knowledge by observing human demonstrations [4].

Cognitive systems for social robots integrate attention mechanisms and RLbI architectures with other modules related to decision layers, reflexes or autonomous action generation. The design of such systems has to consider how these modules are going to share resources and information, and how they are going to be synchronized. Different cognitive systems for social robots have recently been proposed [4, 5], which fall into the category of real-time embedded (DRE) systems. Thus, they need to meet stringent QoS requirements such as to provide responses at human interaction rates (performance), react to unpredictable events efficiently (robustness) and limit their execution to the resources that can be mounted on an autonomous platform (memory, number of cores and CPU speed, etc.). Nowadays, the implementation of robotics systems is moving towards using modern component-based frameworks, which focus on reusability and software evolution [6]. Therefore, the robot software can be composed of interacting binary modules (the components) that communicate using middleware. However, the use of existing component-based frameworks and middleware for robotics usually comes at the expense of a steep learning curve for the adaptation and porting of existing legacy code, which also introduces new challenges to satisfy high-performance and other QoS requirements [7]. Unfortunately, current state of the art in the robotics domain makes it difficult to design robotics components, which can meet these requirements.

This paper analyses the effects of porting two main modules of the cognitive system of a social robot (the attention mechanism and the RLbI architecture) to a fully scalable DRE system with QoS features. The starting point was the original C++ multi-threaded code that was running as a single application within one of the social robot’s (loosely-coupled) embedded computers. The overall performance of the two modules was not entirely satisfactory, which led us to migrate the perceptual part of the system that involves the complete attention mechanism and some modules of the RLbI architecture to a different computer. In order to guarantee that this redesign would meet our QoS requirements, we opted to implement a lightweight middleware in C++ to ease this translation. This middleware, which we called Nerve, guarantees the scalability and QoS requirements for real-time scenarios, but it also allows developers to reuse most of the existing code and provides them with lightweight mechanisms to deploy parts of it into networked tasks. Nerve makes extensive use of design patterns for networked and multi-platform applications available in the Adaptive Communication Environment (ACE) toolkit [8], along with the new Data Distribution Service (DDS)
specification [9] for critical distributed applications with real-time features. The use of DDS and its data-centric publish/subscribe model with QoS policies is what distinguishes Nerve from other existing middleware for robotics. The paper also details how these different QoS policies have been mapped into Nerve functions and their direct application to improve the overall performance of the redesigned system.

The paper is organized as follows. Section 2 gives a short overview of existing middleware for robotics along with their main advantages and drawbacks. Section 3 focuses on the design and implementation of Nerve. Section 4 shows its proposed application to the attention mechanism and the RLbI architecture, respectively, and discusses the main results obtained. Finally, we give our conclusions and future work.

2. MIDDLEWARE FOR EMBEDDED ROBOTICS

Robotics software developers can select from many existing frameworks and libraries for the design and implementation of their applications. Most of these approaches offer solutions (middleware) for the problem of distributed code [10–16]. Where some of them prefer the use of their own ad hoc client/server-based communication model such as Player [10], Carmen [11] or Robot Operating System (ROS) [12], other frameworks make use of platform and language-independent standard middleware that follow the distributed object computing paradigm, such as the Common Object Request Broker Architecture (CORBA) [17] standard by the Object Management Group or the Internet Communications Engine (Ice) by ZeroC [18]. Both CORBA and Ice make use of an Interface Definition Language to define communication interfaces for distributed objects, which are made available through the so-called Object Request Broker. This is the approach followed by Orocos [13] and Miro [14] (CORBA-based), or by Orca [15] and RoboComp [16] (Ice-based).

Orocos and Miro include mechanisms to ensure predictability and deterministic behaviour, which make them suitable for building real-time systems. Orocos makes use of the so-called Real-Time Toolkit, a set of C++ primitives to implement (lock-free) data exchanges and event-driven services in hard real-time. Miro relies on The ACE Object Request Broker (TAO) [19], an open source implementation of the real-time CORBA standard [20] in C++, which supports real-time concurrency and real-time event services for CORBA.

Regarding their communication models, all of the earlier mentioned frameworks provide developers with at least a one-to-one mechanism to implement a basic request/response service (for remote procedure calls). However, only Carmen, Orca and ROS include a publish/subscribe model for one-to-many communications, which usually improves the overall performance of data exchanges and also decouples the way in which networked modules discover and communicate themselves. Nevertheless, these frameworks with publish/subscribe capabilities are not specific for real-time communications and fault tolerance requirements (because of their architectures or of their underlying transport protocols [7]).

None of the earlier mentioned C++ frameworks cover all the features needed by a critical networked system for robotics: simultaneous high-performance and QoS communication and concurrency models. Therefore, Nerve comes to fill this gap between modularity and high-performance QoS-based communications, as a stand-alone middleware or integrated partially (i.e. its DDS features) with existing proposals, such as RoboComp [21].

3. NERVE: A LIGHTWEIGHT QOS MIDDLEWARE FOR DRE ROBOTICS

There are significant challenges in creating DRE software for robotics. Resource-intensive tasks are usually executed on their own threads or processes, which are deployed at different network nodes that deal with specific hardware (sensors and actuators). This decision is delayed until deployment time, and it determines the kind of middleware to be used. However, usual distribution middleware introduces problems to guarantee end-to-end QoS and bounded latency and jitter [8]. A complementary alternative is a host-based middleware, which is a middleware based on inter-process communication mechanisms and threads to guarantee QoS requirements. For instance, vision-related tasks exchanging image data should be deployed as threads, where possible, to reduce penalties.
associated to process’ context-switches and data marshalling. Therefore, we have designed *Nerve* as a lightweight C++ middleware with QoS to cover all these features:

- The encapsulation of critical tasks as platform-independent services, which can be executed as threads or processes at deployment time.
- The adoption of a reactive execution model, in which services react to events available from their own event queue.
- The internal selection of the fastest mechanism for communications: zero-copy buffering for threads, shared memory for processes running within a node and reliable multicast for networked processes. Users will be unaware of the selected mechanism.
- The adoption of a standard with QoS management support in the distributed case by adopting the Object Management Group’s Data Distributed Service recommendation (v1.2).

*Nerve* makes use of several frameworks provided with ACE, which is a platform-independent object-oriented toolkit aimed at implementing host-based middleware for real-time and embedded systems [8]. These frameworks rely on an operating system adaptation layer together with different C++ wrapper façades, which encapsulate the core network programming mechanisms available in common platforms such as interprocess communications, event loops, timers, threads or message queues, among others. The frameworks used in *Nerve* are the Service Executive framework, which supports the configuration of applications the services of which may be assembled dynamically at installation time and/or run-time; and the Streams framework, which simplifies the development of communication software services that exchange data asynchronously using threads, message queues and zero-copy buffering policies [22]. Therefore, *Nerve* allows developers to implement and deploy services, which can exchange and dispatch data asynchronously as threads, but also to publish and subscribe to data topics in distributed mode (using DDS). Figure 1 shows the most important classes in *Nerve*. The Nerve_Service abstract class is a wrapper to implement communicating tasks. It inherits from the ACE_Task_Base class, which is an elaborated ACE wrapper for a thread function. Therefore, developers will implement their service algorithms in a specific method, where they also have available two factories to manage data readers and writers (see ReaderFactory and WriterFactory in Figure 1). These factories provide access to different objects that share a common interface, which hides users from internal communication and buffering strategies (IReader and IWriter for reading and writing data samples, respectively).

The Execution_Manager class is responsible for loading services and executing them in the required deployment plan (available from a configuration file). When services are running as threads, the Execution_Manager takes care of the (local) inter-thread publish/subscribe mechanisms. However, when services are executing as processes *Nerve* uses the mechanisms provided with DDS. For instance, a typical *Nerve* service usually obtains a reference to a ServiceReader object that manages internally a LocalReader object (for inter-thread data receptions) and a SpliceReader (for receptions from DDS). Figure 2 shows the two ways of obtaining new data in both synchronous and asynchronous scenarios (left and right hand side of the figure, respectively). It is worth noting that the Execution_Manager is responsible for executing callbacks associated to inter-thread data receptions in asynchronous mode (using the Reactor design pattern [22]), thus offering the same behaviour of the DDS API (and with a unified interface: the ReaderListener class).

### 3.1. Data Distribution Service in Nerve

The Data Distribution Service for real-time systems standard is a recent specification, which focuses on describing a middleware based on the publish/subscribe model for distributing data with high-performance in real-time environments, where systems must be predictable and deterministic. The standard is composed of the Data-Centric Publish and Subscribe layer and by the DDS Interoperability Wire Protocol (DDSI v2.1). The former defines the DDS architecture, participants and standard API, along with some profiles, which enhance its use [9]. The latter defines a new protocol, which ensures interoperability across DDS implementations from different vendors.

The publish/subscribe model implemented in DDS does not use any central broker. Publishers and subscribers access to the so-called global space data to exchange information, which avoids
Figure 1. Main classes in Nerve (excerpt).

Figure 2. Two methods for receiving data: synchronously (left) and asynchronously (right).

A DDS-BASED MIDDLEWARE FOR QOS AND HIGH-PERFORMANCE NETWORKED ROBOTICS

a single point of failure. Data in DDS are also defined as topics with an associated QoS. These QoS policies in DDS specify resource limits for data queues, liveliness or reliability, among other features. Moreover, publishers/writers and subscribers/readers can also have QoS attributes, which must be compatible before a communication takes place. The DDS in C++ used within Nerve allows our middleware to reuse the same topic type definitions (as C data structures) for both inter-process communications (using shared memory or reliable multicast) and inter-thread data exchanges (using message queues).

Nerve relies on the API available with OpenSplice DDS [23]. This is an open source implementation of the DDS with no CORBA dependence and a very simple configuration mode, in contrast with some other open approaches such as OpenDDS [24], which currently uses CORBA/TAO and does not support inter-process communication with shared memory.

3.2. Managing QoS

Data Distribution Service provides a wide set of real-time QoS policies to control many properties of the communicating entities. However, these QoS policies must be compatible [9] between entities before they can exchange any data. Therefore, the selection and configuration of the most appropriate QoS policies constitute a challenge for any non-expert DDS developer. For instance, different QoS attributes can be associated with different participants in the DDS architecture model, such as topics, readers, writers, publishers, subscribers... (see [9] for more details). Nerve hides these complexities from developers by allowing a single QoS-per-topic configuration policy that is inherited by readers and writers implicitly (although it can be reconfigured as needed). When a new topic
is defined, it owns a QoS object with some default attribute values. Developers can override these values using methods for configuring specific and valid QoS policy groups at the same time, as is discussed next.

3.2.1. Reliable data delivery. Reliability (ReliabilityQoS) is one of the properties that can be adjusted in DDS, regardless of the data transport protocol used. This QoS specifies whether a data exchange must be fully reliable, reliable until resource exhaustion or unreliable, where duplicates, data loss and reordering are possible. However, this QoS must be in-sync with other policies that could prevent the intended behaviour, such as LifespanQoS, HistoryQoS, ResourceLimitsQoS and DestinationOrderQoS (see Table I).

The LifespanQoS configures the expiration time for topic samples. After this happens, data are considered outdated and will not be delivered to service tasks. The HistoryQoS policy is a very powerful QoS because it defines the data queue length for readers and writers in DDS. For instance, a value of \( n \) indicates that queues can only store at most \( m \) data samples, which implies that new incoming samples will replace oldest ones when queues are full. This is the so-called KEEP_LAST history mode. However, the KEEP_ALL mode (infinite depth) indicates DDS that it must keep all new incoming samples in their corresponding queues.

Similarly, the ResourceLimitsQoS policy controls the maximum amount of resources (data samples) that DDS can use. We are considering two main cases in Nerve. The first one comprises an infinite history depth and an infinite or limited number of samples, which allows transport protocols to use flow control when queues are full in a fully reliable scenario. The second case considers a limited history depth and a number of samples that are adjusted automatically to the same value, but with a configurable lifespan value. Finally, the DestinationOrderQoS uses a timestamp to determine the order in which readers will enqueue incoming data samples. By default, Nerve specifies that the data source timestamp (writer) will decide the storage of incoming samples, which could also imply having to replace older ones.

Table I shows the configuration methods available for these different types of reliable and unreliable data delivery. For instance, a fully reliable data exchange could use the QoS::reliable(INFINITE) method, which also sets appropriate values for all the related policies. It is also possible to configure a non-fully reliable behaviour using the QoS::reliableUntil\((n, t)\) method. Finally, the unreliable data delivery is supported as well. This method does not guarantee data delivery when requirements (its arguments) are not satisfied.

3.2.2. Throughput. Some combinations of DDS QoS policies can also improve the throughput between readers and writers (as shown in Table II). For instance, the LatencyBudgetQoS and DeadlineQoS must be minimized to zero, which means that latency and deadline of enqueued data will be optimized internally in the DDS implementation to improve the overall communication performance. TransportPriorityQoS can also help to improve the throughput by assigning different priorities to specific data exchanges (the default and lowest priority in Nerve is zero).

<table>
<thead>
<tr>
<th>RealiabilityQoS</th>
<th>HistoryQoS ((\text{samples}))</th>
<th>LifespanQoS ((\text{nanoseconds}))</th>
<th>ResourceLimitsQoS ((\text{max. samples/instance}))</th>
<th>DestinationOrderQoS</th>
<th>QoS class method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>INFINITE</td>
<td>INFINITE</td>
<td>( r )</td>
<td>by source timestamp</td>
<td>QoS::reliable((r))</td>
</tr>
<tr>
<td>Yes</td>
<td>( n )</td>
<td>( \text{INFINITE} )</td>
<td>( (\text{internal})\ ( r == n )</td>
<td>by source timestamp</td>
<td>QoS::reliableUntil((n, t))</td>
</tr>
<tr>
<td>No</td>
<td>( n )</td>
<td>( 1 )</td>
<td>( (\text{internal})\ ( r == n )</td>
<td>by source timestamp</td>
<td>QoS::unreliable((n, t))</td>
</tr>
</tbody>
</table>

Table II. Nerve QoS configuration values for throughput.

<table>
<thead>
<tr>
<th>LatencyBudgetQoS ((\text{nanoseconds}))</th>
<th>DeadlineQoS ((\text{nanoseconds}))</th>
<th>TransportPriorityQoS ((\text{priority}))</th>
<th>QoS class method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINIMUM</td>
<td>MINIMUM</td>
<td>( p &gt;= 0 )</td>
<td>QoS::throughput((p))</td>
</tr>
</tbody>
</table>
3.2.3. Data persistence. Data persistence is associated with the durability of data samples regarding a writer’s lifecycle. The aim of this property is to guarantee that data samples will be available to readers that start their execution after the writer has been closed. As Table III shows, this feature is related to DurabilityQoS, LifespanQoS, DurabilityServiceQoS and WriterDataLifecycleQoS policies. For instance, data samples will not be delivered to new subscribers when lifespan expires, regardless of the DurabilityQoS strategy (volatile, transient local, transient, persistent) and other values. The complexity of these QoS policies and their relationships are out of the scope of this paper, because they have not been used for the redesign of our robotics cognitive system. Nevertheless, data persistence in Nerve is being used to emulate remote procedure calls with DDS.

4. APPLYING NERVE TO A COGNITIVE SYSTEM FOR A SOCIAL ROBOT

Different cognitive systems for social robots have been proposed in the last decade [2, 5]. They differ in their objectives, components and implementation. All of them, however, include mechanisms for focusing attention on the relevant perceived data and for learning from experience. Vision is the main sensory input social robots employ [4], and some of the main requirements for these robots are being natural and intuitive in interaction with people; therefore, this is why human-like, visual attention mechanisms and RLbI architectures are used as common solutions for perception and learning. In this section, a specific implementation of these two parts of the cognitive system has been migrated from a monolithic design to Nerve to improve their performance and scalability. First of all, we introduce the modules of each part and their functionality. Finally, we describe the porting process, together with the QoS applied and the results obtained.

4.1. The attention mechanism

Methods to model attention are classified as space-based and object-based. Space-based methods deploy attention at space locations, thus they find difficulties in dealing with objects that overlap, are partially occluded or share some characteristics. Object-based approaches use a pre-attentive stage to segment the image into objects, and then the attention is allocated to these objects. They provide a more efficient search than space-based methods, and ease posterior recognition and action processes [3]. Besides, recent psychological contributions show that, in natural vision, the pre-attentive process does not divide a visual input into well-defined objects, but into raw primitive objects, commonly referred to as proto-objects [3]. Proto-objects are image entities, which do not necessarily correspond with a recognizable object, although they possess some of the characteristics of objects.

Figure 3(a) depicts the attention mechanism described in this paper that applies three sequential stages to extract proto-objects from input images. The first stage captures stereo images and represents them using hierarchical structures. Then, the segmentation module is used to group image pixels into proto-objects. This grouping process starts in the first levels of the hierarchical representation, where a pre-segmentation step uses a colour-based distance to group pixels into homogeneous blobs. After this step, blobs are grouped into proto-objects, through the successive levels of the representation, using a distance, which integrates edge and region descriptors.

Once the set of proto-objects has been obtained, the saliency of each of them is computed in the last module depicted in Figure 3(a). Saliency is computed using four features for each proto-object: colour contrast, intensity contrast, disparity and skin colour. From these four features, attractivity maps are computed, containing high values for interesting proto-objects and lower values for other regions. Finally, the saliency map is computed by combining the feature maps into a single representation [3].

Table III. Nerve QoS configuration values for persistence.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>t</td>
<td>h, r</td>
<td>QoS::persistence(d,t,h,r)</td>
<td></td>
</tr>
</tbody>
</table>

4.2. The RLbI architecture

Figure 3(b) shows the vision-based RLbI architecture that has been used as a case study. This architecture focuses on learning social gestures through imitation, using only visual perception. The architecture is divided into six main components: input, perception, knowledge, learning, motion generation and output. As detailed in [4], these components are present in any RLbI architecture. The differences between architectures lie in the modules each component contains and in the relationships between them. The particular characteristics of the components of the architecture used for this case study are briefly detailed next (see [2] for a more detailed explanation).

As depicted in Figure 3(b), the only module used in the input component is related to visual perception. More precisely, in this system, a pair of stereo cameras is used to provide the social robot with information about colour and disparity. The perception component filters the huge amount of information provided by these stereo cameras. The first step in achieving this filtering process is to extract relevant features from input images. The feature detection element focuses on the perception of the human performer to recognize and learn human social gestures. So, it firstly searches a close human face in the perceived images. Then, the person’s silhouette is obtained from a disparity map, and the hands are located as skin colour regions in certain parts of the silhouette. Face and hands regions are tracked using a fast hierarchical algorithm [25]. Finally, the last stage in the perception component uses a model-based approach to infer upper-body torso pose from the silhouette information and tracked regions. Before describing the rest of the components of the architecture, it is important to mention that these first two components (input and perception) represent a very important percentage of the computational load of the gesture recognition and learning process. This should be carefully considered when executing the integration of this RLbI architecture with Nerve.

The output of the perception component is a sequence of three-dimensional trajectories followed by different body parts that are fed to the knowledge component. Whereas these trajectories do not represent a large flow of data per frame, a gesture can be composed by several hundreds of frames. Social robots have to achieve online response, thus the knowledge component firstly reduces the dimensionality of perceived data by encoding them in an efficient representation, based on local and global features [2]. Then, perceived gesture is compared against the gestures stored in the repertoire of the social robot. This comparison is based on simple analytical relations for global features. Local features are compared using more complex dynamic alignment techniques [2]. Whereas gesture comparison may be a time-consuming process, it may also benefit from being executed in parallel with the perception processes (i.e. the recognition process can be executed while the social robot begins perceiving a new gesture). In any case, the results of this comparison are used by the
learning component to modify the contents of the gesture repertoire. This learning process requires a degree of human supervision, as detailed in [2].

The representation, recognition and learning processes are executed in the human motion space. Thus, these processes are the same irrespective of the particular social robot being used. The retargeting component, however, considers the characteristics of the robot to make it correctly imitate human gestures (i.e. map human gestures into the robot body). Following ideas taken from computer graphics [26], a combined retargeting strategy is used in this module. The resulting robot motion is fed to the motion generation component. This component checks the validity of the robot poses before sending these poses to the motors of the robot (i.e. the output component). An important characteristic of the motion generation and output components of the used RLbI architecture is that they do not need to be executed unless physical imitation is required of the robot. The same can be applied to the retargeting component.

4.3. From a monolithic cognitive system to Nerve

The software of both the attention mechanism and the RLbI architecture was originally developed in C++ as single but complex applications. The social robot uses the attention mechanism when it is moving around autonomously, or the RLbI system when a human requires it to learn new social gestures. Figure 4 shows the simplified communication diagrams of the objects involved in both systems.

The attention mechanism is composed by three stages that are executed sequentially. Thus, when new image data are available, hierarchical representations are created, then they are segmented and the saliency of the resulting proto-objects is computed.

The simplicity of the attention mechanism architecture contrasts with the RLbI one. This relies on a main thread, depicted as UpdateModel that executes an infinite loop to query and update RLbI-related information. Two more threads are responsible for motion capture and gesture processing, respectively. First of all, the UpdateModel main loop gets new positions of parts of the tracked person (and their interpolated values) from the MovementCapture object. Secondly, the main thread updates data in the PersonModel object. Then, it notifies the GestureProcess thread, which is waiting for new gesture positions at specific frames. After this notification, the main procedure gets the current positions of the RobotModel and then gives the order to the Retargeting object to update the RobotModel (a decision based on PersonModel current parameters). Finally, the Learner object stores the perceived trajectories and also executes the gesture recognition and learning algorithms when required.

The attention mechanism has a simple structure in which the segmentation process represents the critical stage in terms of computational complexity. The first approach to migrate this system using Nerve could be based on three different services, one for each stage in the attention process. However, saliency computation is a process tightly coupled with the segmentation process. Thus, the final implementation of the attention mechanism uses two Nerve services, as Figure 5 depicts. These two services, ImageCapture and Segmentation_Saliency are executed in the node that is connected to the robot cameras, thus it is not necessary to transmit images between different network nodes.

The RLbI architecture represents a more complex application in which performance is clearly constrained by its design and the features of its execution platform. Fortunately, the migration of the existing architecture to a DRE system has been quite straightforward using Nerve. First of all, we

![Figure 4. Interaction diagrams of the original attention mechanism (left), and RLbI architecture (right).](image-url)
have grouped the existing classes into four Nerve services, as depicted in Figure 5. As mentioned before, these services are fully decoupled and will communicate by publishing and subscribing to data topics. The MovementCapture service will now work as a request/response service, namely it will subscribe to a request topic and, after its arrival, it will publish a response topic, which contains new movement parameters; the PersonModel service, equivalent to its counterpart in the original version, is the one that requests data from the MovementCapture service. In the redesign, we have also created a GestureProcess service (a clear candidate because of its original threaded design, which now executes the learner procedure). Finally, we have merged the Retargeting and RobotModel processes into a single service. This makes sense because the retargeting strongly depends on the kind of robot model being used, and the robot model is not used elsewhere.

It is worth noting that in the resulting distributed version, there is no need for a main loop to govern the whole execution of the cognitive system. With our approach, it is now possible to completely decouple the processes involved and to migrate the most critical parts to another dedicated network node. In this case, as shown in Figure 5, the services that are related to perception are now deployed in a node that also controls the cameras. These services, ImageCapture, Segmentation_Saliency and MovementCapture, correspond to the attention mechanism and to the input and perception blocks in the conceptual RLbI schema (Figure 3). The social robot sends the information provided by the cameras to the ImageCapture service when it is navigating around or to the MovementCapture service when it is learning new gestures. The rest of the services of the RLbI architecture are deployed in a second node and are executed as threads.

With this strategy, we have overtaken the performance limitations of the original cognitive system in general, and RLbI architecture, in particular. Tables IV and V show the results of the experiments using the original system and the new one using Nerve. Tests have been conducted in real indoor

<table>
<thead>
<tr>
<th>Measured service</th>
<th>Orig. RLbI</th>
<th>Nerve-RLbI (local)</th>
<th>Nerve-RLbI (distributed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MovementCapture</td>
<td>8.41</td>
<td>10.16</td>
<td>14.98</td>
</tr>
<tr>
<td>PersonModel</td>
<td>26.80</td>
<td>27.50</td>
<td>40.21</td>
</tr>
<tr>
<td>RobotModel</td>
<td>26.80</td>
<td>27.50</td>
<td>40.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average time per frame</th>
<th>Orig. attention mechanism</th>
<th>Nerve-attention mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>559.7</td>
<td>248.4</td>
<td></td>
</tr>
<tr>
<td>12.7</td>
<td>6.6</td>
<td></td>
</tr>
</tbody>
</table>
Table VI. Main data topics exchanged in the system.

<table>
<thead>
<tr>
<th>System topic</th>
<th>Publisher</th>
<th>Subscriber</th>
<th>QoS configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_motion_capture</td>
<td>PersonModel</td>
<td>MovementCapture</td>
<td>reliableUntil(1)</td>
</tr>
<tr>
<td>motion_capture_request</td>
<td>PersonModel</td>
<td>MovementCapture</td>
<td>reliableUntil(1)</td>
</tr>
<tr>
<td>motion_capture_response</td>
<td>MovementCapture</td>
<td>PersonModel</td>
<td>throughput(1)</td>
</tr>
<tr>
<td>gesture_data</td>
<td>PersonModel</td>
<td>GestureProcess</td>
<td>reliable(INFINITE)</td>
</tr>
<tr>
<td>retargeting_data</td>
<td>PersonModel</td>
<td>RetargetingRobot</td>
<td>reliableUntil(15)</td>
</tr>
<tr>
<td>image_data</td>
<td>ImageCapture</td>
<td>Segmentation_Saliency</td>
<td>unreliable(1)</td>
</tr>
</tbody>
</table>

Scenarios. For the RLbI application, different human performers execute gestures that have to be recognized by the social robot (see [2] for a detailed description of the used experimental setup).

The performance of the attention mechanism has been obtained by measuring the complete processing time. Both the average time per frame and the standard deviation are provided in Table V. It can be seen that the implementation using Nerve services represents an important improvement that reduces processing times to less than a half with respect to the previous monolithic system.

Table IV shows the performance measurements related to the RLbI architecture, in two different deployments: one with all the services running locally as threads, and the final configuration with two nodes (as depicted in Figure 5). In the three cases, the performance of the most critical tasks in frames per second have been measured. Higher values in frames per second ease human–robot interactions, improve the perceptual capabilities of the robot and produce more natural imitated motions. The MovementCapture measurements represent the number of human poses perceived per second. This is usually the natural bottleneck for any vision-based RLbI architecture, where the de facto standard is 25 fps to show a fluid perception. The PersonModel measurements indicate the speed to update the imitated movement. Finally, the RobotModel measurements show the number of positions per second that are sent to the motors of the robot. After inspecting the results in the table, it is worth noting that we already obtain an interesting performance boost after the application of Nerve in the local scenario with one node (all services are threads). However, the best results are reached in the distributed scenario, where the most important improvement can be found in the MovementCapture service, which performs almost 80 per cent better than in its original version. The improvements are also notable in the rest of the measured services, which validates our middleware as a way to develop scalable DRE applications in the robotics domain.

Table VI shows the main topics used in the cognitive system and their different QoS configurations. Some of them have been defined as reliable (with or without history depth, as described in the previous section) and others have a higher priority to optimize their throughput (such as the critical motion_capture_request, motion_capture_response). Experiments showed us that the RetargetingRobot module could enqueue the latest 15 data samples, which were identified as an acceptable delay between this service and the PersonModel one.

5. CONCLUSIONS

This paper has presented a new middleware called Nerve. Its principal novelty and therefore what differentiates it from other existing approaches, is its main focus on the fine tuning and modelling of the communication and concurrency dimensions. This allows Nerve to guarantee stringent QoS requirements such as real-time and high-performance. It has been validated in the domain of social robotics by implementing two key parts of the cognitive system of a social robot.

Developers using QoS-aware communication modes in Nerve can think naturally on software data buses to publish their distributed information, in the same manner that they use and understand existing hardware-based buses. In spite of its novelty, the DDS standard has already demonstrated that it is a powerful approach for the development of DRE systems and that it is now developed enough to be used in the field of robotics. Nerve developers can also decouple (and maintain) their...
modules more easily than using a stand-alone Remote Procedure Call/Remote Method Invocation approach such as CORBA.

Our future work will focus on incorporating our middleware as part of a full software development cycle for robots, where implementation details are delayed until the design of services and their deployment plan are specified visually. We are now exploring the possibilities of generating Nerve-based code from visual specifications in SmartSoft [27], which should be also extended to include the definition of QoS properties for tasks together with the mechanisms to verify them.

ACKNOWLEDGEMENTS

This work has been partially granted by the Spanish Ministerio de Ciencia e Innovación (MICINN) and FEDER funds, and by the Junta de Andalucía, under projects no. TIN2008-05932, TIN2008-06196 and P07-TIC-03106, respectively.

REFERENCES

