This is the pre-peer reviewed version of the following article: Cruz, J.M., Romero-Garcés, A., Rubio, J.P.B., Robles, R.M. and Rubio, A.B. (2012), A DDS-based middleware for quality-of-service and high-performance networked robotics. Concurrency Computat.: Pract. Exper., 24: 1940-1952. https://doi.org/10.1002/cpe.2816, which has been published in final form at https://doi.org/10.1002/cpe.2816. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **00**:1–12
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

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, 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 describes our experiences in the design and implementation of a lightweight middleware for networked robotics called *Nerve*, which guarantees the scalability and quality-of-service (QoS) requirements for this kind of real-time software. Its benefits have been proved through its use in two key components of the cognitive system of a social robot: a visual attention mechanism and a robot learning by imitation control architecture. *Nerve* makes use of existing patterns for networked applications along with the recent Data Distribution Service specification (DDS), where different QoS have been applied carefully to achieve the best performance of the target robot. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: robotics; middleware; data distribution service; performance; quality-of-service

1. INTRODUCTION

Social robots are designed to work in daily life scenarios, and to cooperate with people in solving daily life tasks. In order to achieve these objectives, these robots have to use complex cognitive systems that allow them to extract useful information from uncontrolled environments, to provide people with natural and intuitive interaction channels, and to adapt to different tasks and environmental conditions [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. On the other hand, developing systems that allow the robot to focus attention on the same cues as their human partners is a critical step in designing social robots that cooperate with people, and that learn from natural human instruction. While 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. In fact, in the last few years the emphasis has increased in the development of robot vision systems according to the model of natural vision, due to 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

*Correspondence to: Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain. jmcruz@lcc.uma.es
†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.

Copyright © 2011 John Wiley & Sons, Ltd.

to the present task. Thus, visual attention mechanisms for social robots aim to provide efficient human-like perception, that can be used to facilitate sharing perceptual cues between the robot and its human companions at run time [2].

Increased perceptual ability is one of the key differences between industrial and social robots. The other main difference is learning. While for an industrial robot it is usually possible to preprogram all possible situations it may face, social robots have to be provided with mechanisms to adapt to dynamic environments, different people and a huge variety of tasks. Learning becomes, then, a key topic for social robots. There are different options to achieve learning. Individual learning (e.g. trial-and-error, imprinting, classical conditioning,...) may lead the robot to learn incorrect, disturbing or even dangerous behaviours [3]. Social learning mechanisms, on the other hand, avoid these issues as they allow the human teacher to supervise the learning process. While there are different social learning strategies, learning by imitation appears as one of the most intuitive and powerful ones. Thus, in the last decade, many 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 components related to higher level decision layers, reflexes, or autonomous action generation. The design of such systems has to be carefully considered. While different cognitive systems for social robots have been recently proposed, the way in which these systems are implemented and deployed in a real robot platform introduces many open issues. Traditionally, robotic tasks are decomposed and mapped into software modules with stringent requirements in terms of speed and robustness. These constraints have influenced the methodologies adopted by robotics engineers, which are usually more focused on performance than reusability and software evolution. Nevertheless, recent approaches propose a more modern component-based development process [5]. Although this approach has some important advantages regarding reusability and scalability, it is worth noting that existing legacy code may be difficult to adapt to this paradigm, which also implies a steep learning curve. In fact, the selection of the most appropriate frameworks and middleware along with their specific configurations and programming modes introduces new challenges for developers that influence the way in which previous requirements on performance can still be satisfied [6].

This paper introduces our experience in porting two main components of the cognitive system of a social robot to a fully scalable distributed real-time embedded (DRE) system. These two components are the attention mechanism and the RLbI architecture. The starting point for both of them 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. Unfortunately, the overall performance of these two components 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. However, instead of redesigning the application from scratch using a component-based approach, we decided to reuse most of the existing code and provide developers with lightweight mechanisms to distribute parts of it into networked tasks. The result, and therefore, the main contribution of the paper is the design and development of a lightweight middleware in C++ called *Nerve*, which guarantees the scalability and quality-of-service (QoS) requirements for real-time scenarios. *Nerve* makes extensive use of design patterns for networked and multi-platform applications available in the ACE toolkit [7], along with the new Data Distribution Service (DDS) specification [8], a new standard for critical distributed applications with real-time features. The use of DDS and its associated protocols is what distinguishes *Nerve* from other existing middleware for robotics, although it is worth noting that its QoS-based design principles can also serve as the right complement to improve other approaches.

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 discuss the main results obtained. Finally, we give our conclusions and future work.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2. MIDDLEWARE FOR EMBEDDED ROBOTICS

Robotics software developers can select from many existing frameworks and APIs for the design and implementation of their applications. Most of these approaches offer solutions (middleware) for the problem of distributed code [9, 10, 11, 12, 13, 14, 15]. Where some of them prefer the use of their own ad-hoc client/server-based communication model such as Player [9], Carmen [10] or ROS [11], other frameworks make use of platform and language-independent standard middleware that follow the distributed object computing paradigm, such as the CORBA [16] standard by the Object Management Group (OMG) or the Internet Communications Engine (Ice) by ZeroC [17]. Both CORBA and Ice make use of an Interface Definition Language (IDL) to define communication interfaces for distributed objects, which are made available through the so-called Object Request Broker (ORB). This is the approach followed by Orocos [12] and Miro [13] (CORBA-based), or by Orca [14] and RoboComp [15] (Ice-based).

Orocos and Miro include mechanisms to ensure predictability and deterministic behavior, 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 TAO [18], an open source implementation of the real-time CORBA standard [19] in C++, which supports real-time concurrency and real-time event services for CORBA.

Regarding their communication models, all of the above-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 (due to their architectures or to their underlying transport protocols [6]).

None of the above-mentioned C++ frameworks cover all the features needed by a critical networked system for robotics: simultaneous high-performance and quality-of-service 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 [20].

3. NERVE: A LIGHTWEIGHT QOS MIDDLEWARE FOR DRE ROBOTICS

There are significant challenges in creating DRE software for robotics. First of all, 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). The way in which these distributed tasks communicate is critical for high-performance. For instance, inter-thread communication will be faster than inter-process communication. Moreover, communicating processes within the same node will obtain better data transfer rates than their equivalent networked case. Middleware should hide developers from these low-level communication details, i.e., which protocol and inter-process communication technique (shared memory, message passing, sockets) is the most suitable to use in order to satisfy the robot's performance requirements.

But, while taking into account the best way to exchange data, these tasks should also be unaware of their execution context. Developers should not have to be concerned whether their algorithms will be executed as threads or processes until deployment time, which means that our middleware should deal with this situation providing a way to wrap (service) tasks within an appropriate context to execute and manage them at run-time.

Therefore, we have designed *Nerve* as a lightweight C++ middleware to cover the features mentioned above along with quality-of-service network-based communications. Its main features are:

Copyright © 2011 John Wiley & Sons, Ltd.

Concurrency Computat.: Pract. Exper. (2011)
DOI: 10.1002/cpe

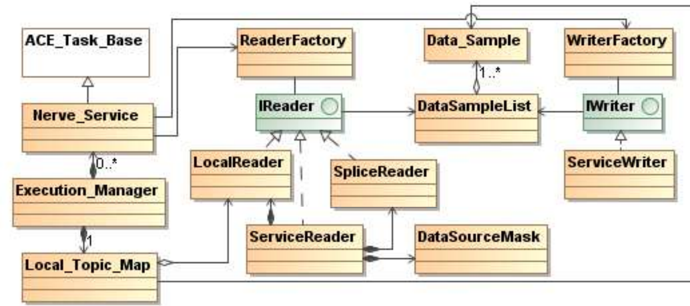

Figure 1. Main classes in Nerve (excerpt.)

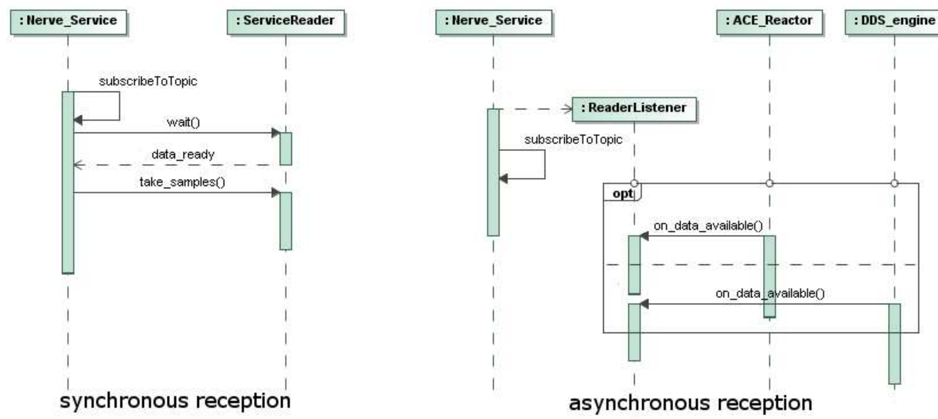

Figure 2. Two methods for receiving data: synchronously (left) and asynchronously (right)

- 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 in the distributed case by adopting the OMG's Data Distributed Service recommendation (v1.2).

Nerve makes use of several frameworks provided with ACE, which is a platform-independent object-oriented toolkit used in many types of real-time, and embedded systems. It consists of frameworks that implement design patterns for communications and concurrency [7]. They 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 C++ wrappers in ACE not only ensure platform independency but also reduce most of the development effort spent on lower-level network and concurrent programming details.

Nerve reuses part of the ACE Service Executive and the Streams frameworks, it being now possible not only to create services which can exchange and dispatch data asynchronously using message queues and zero-copy buffering policies, but also to publish and subscribe to data topics. Fig. 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

Copyright © 2011 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

in a specific method, where they also have available two factories to manage data readers and writers (see ReaderFactory and WriterFactory in fig. 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 [7]), thus offering the same behaviour that is available with the DDS API (and with a unified interface: the ReaderListener class).

3.1. DDS in Nerve

The Data Distribution Service for Real-time Systems standard is a recent specification which focus 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 (commonly referred to as DCPS) 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 [8]. 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 a single point of failure. Data in DDS are also defined as *topics* with an associated quality of service. 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 [21]. 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 [22], which currently uses CORBA/TAO and does not support inter-process communication with shared memory.

3.2. Managing QoS

DDS provides a wide set of real-time QoS policies to control many properties of the communicating entities. These QoS policies are matched based on a request vs. offered model: when QoS policies are incompatible between two entities they will not be able to exchange data.

Therefore, the selection and configuration of the most appropriate QoS policies constitutes a challenge for any non-expert DDS developer. First of all, different QoS attributes can be associated with different participants in the DDS architecture model, such as topics, readers, writers, publishers, subscribers... (see [8] 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 (for publication or subscription purposes), 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.

Copyright © 2011 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

RealiabilityQoS	HistoryQoS (samples)	LifespanQoS (nanoseconds)	ResourceLimitsQoS (max. samples/instance)	DestinationOrderQoS	QoS class method
Yes	INFINITE	INFINITE	r	by source timestamp	QoS::reliable(r)
Yes	n	t	(internal) r == n	by source timestamp	QoS::reliableUntil(n, t)
No	n	t	(internal) r == n	by source timestamp	QoS::unreliable(n, t)

Table I. Possible choices in QoS for reliability and their Nerve corresponding methods

LatencyBudgetQoS (nanoseconds)	DeadLineQoS (nanoseconds)	TransportPriorityQoS (priority)	QoS class method
MINIMUM	MINIMUM	p >= 0	QoS::throughput(p)

Table II. Nerve QoS configuration values for throughput

DurabilityQoS (enum. value)	LifespanQoS (nanoseconds)	DurabilityServiceQoS (history_depth, max.samples/instance)	WriterDataLifecycleQoS	QoS class method
d	t	h, r	(applied internally)	QoS::persistence(d,t,h,r)

Table III. Nerve QoS configuration values for persistence

3.2.1. Reliable data delivery Reliability (ReliabilityQoS) is one of the properties that can be adjusted in DDS (independently of the data transport protocol used). This QoS specifies whether duplicates, data loss and reordering are allowed. However, this QoS must be in-synch 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 x indicates that queues can only store at most x 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 is adjusted automatically to the same value, but with a configurable lifespan value. Finally, the DestinationOrderQoS determines the order in which readers will enqueue incoming data samples (using a timestamp). By default, *Nerve* specifies that the data source timestamp (writer) will decide the storage of incoming samples, which could also imply to replace oldest 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 sets also 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 also supported. This method does not guarantee data delivery when requirements (its parameters) 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).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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 with DurabilityQoS, LifespanQoS, DurabilityServiceQoS and WriterDataLifecycleQoS policies. For instance, data samples will not be delivered to new subscribers when lifespan expires, independently of the DurabilityQoS strategy (volatile, transient local, transient, persistent) and other values. The complexity of these QoS policies and their relationships is 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 [23, 3]. They differ in their objectives, components and implementation. All of them, however, include mechanisms to focus attention on the relevant perceived data, and to learn from experience. Being vision the main sensory input social robots employ [4], and being natural and intuitive interaction with people one of the main requirements for these robots, human-like, visual attention mechanisms, and RLbI architectures are common solutions to achieve perception and learning, respectively. In this section, a particular implementation of these two parts of the cognitive system has been migrated from a monolithic design to *Nerve*, to improve their robustness and performance.

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 ocluded or share some characteristics. Object-based approaches use a preattentive stage to segment 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 [2]. On the other hand, recent physcological contributions show that, in natural vision, the preattentive process does not divide a visual input into well-defined objects, but into raw primitive objects, commonly referred as proto-objects [2]. Proto-objects are image entities which do not necessarily correspond with a recognizable object, although they possess some of the characteristics of objects.

Fig. 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 sucessive 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 fig. 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 [2].

4.2. The RLbI architecture

Fig. 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

Copyright © 2011 John Wiley & Sons, Ltd.

Concurrency Computat.: Pract. Exper. (2011)
DOI: 10.1002/cpe

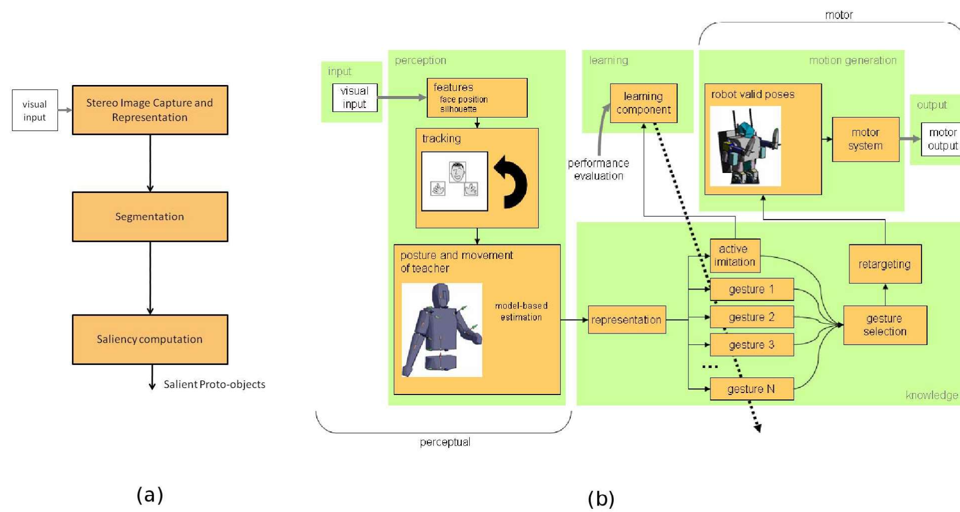

Figure 3. (a) Visual attention mechanism; and (b) Overview of the RLbI architecture

relationships between them. The particular characteristics of the components of the architecture used for this case study are briefly detailed below (see [3] for a more detailed explanation).

As depicted in fig. 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 color and disparity. The perception component filters the huge amount of information provided by these stereo cameras. The first step to achieve this filtering process is to extract relevant features from input images. In order to recognize and learn human social gestures, the feature detection element focuses on the perception of the human performer. Thus, 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 color regions in certain parts of the silhouette. Face and hands regions are tracked using a fast hierarchical algorithm [24]. 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 3D trajectories followed by different body parts, that are fed to the knowledge component. While these trajectories do not represent a large flow of data per frame, a gesture can be composed by several hundreds of frames. In order to achieve *on-line* response, the knowledge component firstly reduces the dimensionality of these data by encoding them in an efficient representation, based on local and global features [3]. 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 [3]. While gesture comparison may be a time consuming process, it may also benefit from being executed in parallel with respect to perception processes (i.e. the recognition process can be executed whilst 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 includes some degree of human supervision, as detailed in [3].

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, on the other hand, considers the characteristics of the robot to make it

Copyright © 2011 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

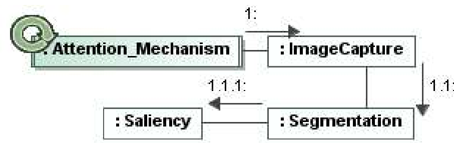

Figure 4. Interaction diagram of the original attention mechanism

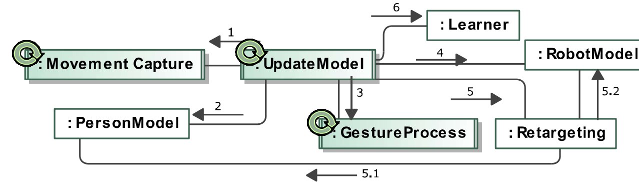

Figure 5. Interaction diagram of the original RLbI architecture

correctly imitate human gestures (i.e. map human gestures into the robot body). Following ideas taken from computer graphics [25], 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. Figs. 4 and 5 show the simplified communication diagram 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 fig. 6 depicts. These two services, `ImageCapture` and `Segmentation_Saliency` are executed in the node

Copyright © 2011 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

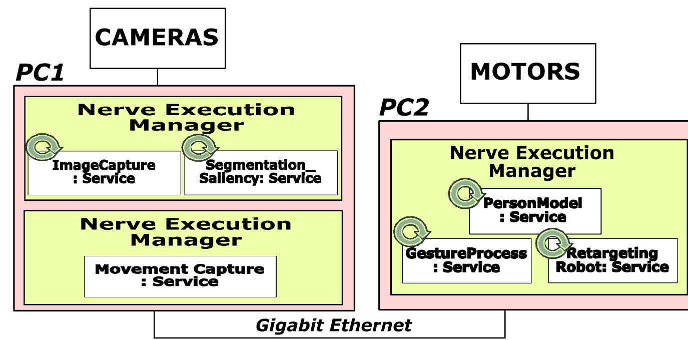

Figure 6. Resulting services in the cognitive system using Nerve

Measured Service	Orig. RLbI	Nerve-RLbI (local)	Nerve-RLbI (distributed)
MovementCapture	8.41	10.16	14.98
PersonModel	26.80	27.50	40.21
RobotModel	26.80	27.50	40.23

Table IV. Experiments and measurements for the RLbI subsystem in frames per second

	Orig. Attention Mechanism	Nerve-Attention Mechanism
Average time per frame	559.7	248.4
Standard deviation	12.7	6.6

Table V. Experiments and measurements for the attention mechanism, in milliseconds

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 have grouped the existing classes into four *Nerve* services, as depicted in fig. 6. 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 has a strong dependency on the kind of robot model that is 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 fig. 6, 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 (fig. 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.

Copyright © 2011 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

System topic	Publisher	Subscriber	QoS configuration
init_motion_capture	PersonModel	MovementCapture	reliableUntil(1)
motion_capture_request	PersonModel	MovementCapture	reliableUntil(1) throughput(1)
motion_capture_response	MovementCapture	PersonModel	reliableUntil(1) throughput(1)
gesture_data	PersonModel	GestureProcess	reliable(INFINITE)
retargeting_data	PersonModel	RetargetingRobot	reliableUntil(15)
image_data	ImageCapture	Segmentation_Saliency	unreliable(1)

Table VI. Main data topics exchanged in the system

With this strategy we have overtaken the performance limitations of the original cognitive system in general, and RLbI architecture in particular. Tables V and IV show the results of the experiments using the original system and the new one using *Nerve*. Tests have been conducted in real indoor scenarios. For the RLbI application, different human performers execute gestures that have to be recognized by the social robot (see [3] 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 fig. 6). 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 eighty per cent better than in its original version. The improvements are also notable in the rest of 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 fifteen data samples, which was identified as an acceptable delay between this service and the `PersonModel` one.

5. CONCLUSIONS

This paper has presented our experiences in developing software in the domain of social robotics. In contrast to other existing approaches and related work, our main focus has been on the fine tuning and modeling of the communication and concurrency dimensions in order to meet stringent requirements such as real-time and high-performance. Therefore, we have designed and validated the *Nerve* middleware, which can be also used as a novel complement for other well-established paradigms for developing robotics software that currently face a trade-off between modularity and quality-of-service agreement for embedded and critical software.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

In spite of its novelty, the DDS standard has already demonstrated to be a powerful approach for the development of DRE systems and that it is now mature to be used in the robotics field. Developers using DDS 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. They also decouple (and maintain) their modules more easily than using a stand-alone RPC/RMI 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 [26], which should be also extended to include new QoS properties along with mechanisms to verify them.

REFERENCES

1. Breazeal C. *Designing sociable robots*. Cambridge, MA, USA, 2002.
2. Palomino AJ, Marfil R, Bandera JP, Bandera A. A novel biologically inspired attention mechanism for a social robot. *EURASIP Journal on Advances in Signal Processing* 2011; **2011**, doi:doi:10.1155/2011/841078.
3. Bandera JP. Vision-based gesture recognition in a robot learning by imitation framework. Ph.d. dissertation, Available at http://www.grupoisis.uma.es. Dpto. Tecnología Electrónica, Universidad de Málaga, Spain 2010.
4. Bandera JP, Rodríguez JA, Molina-Tanco L, Bandera A. A survey of vision-based architectures for robot learning by imitation. *International Journal of Humanoid Robotics (to appear)* ; .
5. Brugali D, Prassler E. Software engineering for robotics [From the Guest Editors]. *Robotics & Automation Magazine, IEEE* March 2009; **16**(1):9–15, doi:10.1109/MRA.2009.932127.
6. Martínez J, Romero-Garcés A, Vázquez-Martín R, Bandera A. Recipes for designing high-performance and robust software for robots. *Proceedings of the International Conference on Robotics Automation and Mechatronics (RAM 2010)*, Singapore, Singapore, 2010; 250–255.
7. Schmidt DC, Huston S. *C++ Network Programming Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003.
8. Object Management Group. Data Distribution Service for Real-time Systems (DDS), version 1.2 2007.
9. Gerkey BP, Vaughan RT, Howard A. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003; 317–323.
10. Montemerlo M, Roy N, Thrun S. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit. *In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2003; 2436–2441.
11. ROS open source community. ROS: The meta-operating system for robots. Available at http://ros.org 2011.
12. Bruyninckx H, Soetens P, Koninckx B. The real-time motion control core of the Orocos project. *IEEE International Conference on Robotics and Automation*, 2003; 2766–2771.
13. Enderle S, Utz H, Sablatng S, Simon S, Kraetzschmar G, Palm G. Miro: Middleware for Autonomous Mobile Robots. *In Telematics Applications in Automation and Robotics*, 2001.
14. Brooks A, Kaupp T, Makarenko A, Williams S, Orebäck A. Orca: A component model and repository. *Software Engineering for Experimental Robotics*, Brugali D (ed.). Springer Tracts in Advanced Robotics, Springer - Verlag, 2007.
15. Bustos P, Bachiller P, Manso L. RoboComp Project. Available at http://sourceforge.net/apps/mediawiki/robocomp 2011.
16. Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP) 2008.
17. ZeroC. Internet Communications Engine. Available at http://www.zeroc.com/ 2011.
18. Schmidt DC, Gokhale A, Harrison TH, Levine D, Cleeland C. Tao: a high-performance endsystem architecture for real-time corba 1997.
19. Object Management Group. Real Time CORBA 2005.
20. Martínez J, Romero-Garcés A, Manso L, Bustos P. Improving a robotics framework with real-time and high-performance features. *Simulation, Modeling, and Programming for Autonomous Robots*, *Lecture Notes in Computer Science*, vol. 6472, Springer, 2010; 263–274.
21. PrismTech. OpenSplice DDS. Available at http://www.opensplice.com 2011.
22. Object Computing Inc. The open source C++ DDS implementation (OpenDDS). Available at http://www.opendds.org/ 2011.
23. Breazeal C, Brooks A, Gray J, Hoffman G, Kidd C, Lee H, Lieberman J, Lockerd A, Mulanda D. Humanoid robots as cooperative partners for people. *International Journal of Humanoid Robots* 2004; **1**(2):1–34.
24. Marfil R, Bandera A, Rodríguez J, Sandoval F. Real-time template-based tracking of non-rigid objects using bounded irregular pyramids. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems*, vol. 1, 2004; 301–306.
25. Shin HJ, Lee J, Shin SY, Gleicher M. Computer pupettry: An importance-based approach. *ACM Transactions on Graphics* 2001; **20**(2):67–94.
26. Schlegel C, Hassler T, Lotz A, Steck A. Robotic software systems: From code-driven to model-driven designs. *Advanced Robotics, 2009. ICAR 2009. International Conference on*, 2009; 1 –8.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe