Improving a Robotics Framework with Real-Time and High-Performance Features

Jesús Martínez*, Adrián Romero-Garcés, Luis Manso, and Pablo Bustos

Computer Science Department, University of Málaga, Spain
jmcruz@lcc.uma.es,
Dept. Tecnología de los Computadores y las Comunicaciones, University of Extremadura, Spain
adrigtl@unex.es,lmanso@unex.es,pbustos@unex.es

Abstract. Middleware has a key role in modern and object-oriented robotics frameworks, which aim at developing reusable, scalable and maintainable systems using different platforms and programming languages. However, complex robotics software falls into the category of distributed real-time systems with stringent requirements in terms of throughput, latency and jitter. This paper introduces and analyzes a methodology to improve an existing robotics framework with real-time and high-performance features using a recently adopted standard: the Data Distribution Service (DDS).

Keywords: robotics, middleware, data distribution service, performance, real-time

1 Introduction

Software Engineering for robotics is focused on designing and implementing robot control architectures to be reusable, scalable and maintainable, so that software modules and algorithms for robots can be adapted to new platforms and requirements, not only reducing the cost and time-to-market of a complete robotics system but also improving its overall performance. In this application domain, the complex tasks performed by a robot are divided into distributed processes, which communicate using a middleware. This key software layer connects networked applications in a platform and language-independent manner. The middleware also hides low-level communication details from developers.

One of the most important milestones in the robotics field has been the creation of libraries and object-oriented frameworks for building robotics software. Although some of these frameworks are now mature and widely used in the robotics community for research purposes, sometimes it is unclear whether they will allow developers to deploy complete systems in embedded platforms which

* This project has been partially supported by grants TSI-020301-2009-27 from the Spanish Government and the FEDER funds and by P07-TIC-003184 from the Junta de Andalucía funds

meet the stringent real-time requirements that a running robot needs. Therefore, rigorous analysis must be done to guarantee deterministic behaviors that do not miss any critical deadline, and it is not surprising that the use of third-party software, such as middlewares, may affect the predictability and performance of a system if they can not be adjusted properly.

Unfortunately, some middlewares are not best suited to implement distributed and real-time embedded (DRE) robotic systems. However, widely adopted frameworks may not replace middleware easily, unless they are redesigned from scratch. This paper presents our experiences to adapt a robotics framework to be DRE compliant. In our study we have selected the novel *Data Distribution Service for Real-Time Systems* (DDS) standard [10] from the Object Management Group (OMG), which addresses the anonymous, decoupled, and asynchronous communication among a data sender, called publisher, and its subscribers. This standard addresses the needs for real-time and quality of service (QoS) of distributed applications, and it is being adopted in mission- and business-critical applications, such as air traffic control, telemetry or financial trading systems.

The main contributions of the paper are *i)* the proposal and demonstration that a DDS-based middleware improves significantly a component-based robotics framework (RoboComp [15]) in terms of throughput, latency and jitter, and *ii)* a methodology to incorporate this middleware to the framework with minimum changes from a developer's perspective, which also ensures backward compatibility with previously deployed modules. Our study shows interesting results that help to conclude that the use of DDS will be important for the robotics community in the years to come.

The paper is organized as follows. Section 2 gives an overview of the state of the art in middleware for robotics. Section 3 introduces our methodology to include DDS in RoboComp, a component-based robotics framework. Finally, we give some concluding remarks.

2 State of the art in middleware for robotics

Design patterns [4] and frameworks have been used within the robotics community in many proposals [5, 7, 2, 3, 1, 15]. In general, these open source approaches provide a catalogue of services (from low-level to high-level tasks) which allow developers to reuse existing code in order to create complex software for a robot. They also rely on some kind of middleware to allow the deployment and communication of services within a distributed platform, where resource-intensive software modules are executed in different network nodes.

The middlewares proposed by Player [5] and Carmen [7] are the most basic ones, and require some management of low-level communication details from developers in order to implement new distributed services, for instance to define new message types in a platform-independent way using a specific C API (Carmen) or using the eXternal Data Representation (XDR) notation [16] and its C compiler (Player).

Other frameworks hide these complexities from developers by using platform and language-independent standard middlewares, which follow the distributed object computing paradigm, such as the CORBA [11] standard by the Object Management Group (OMG) or the Internet Communications Engine (Ice) by ZeroC [19] (an industrial-quality middleware used in many critical projects). The two middlewares make use of an Interface Definition Language (IDL) to define communication interfaces for distributed objects, which will be available through the so-called Object Request Broker (ORB). This is the approach followed by Orocos [2] and Miro [3] (CORBA-based), or by Orca [1] and RoboComp (Ice-based).

All of the above-mentioned frameworks may be configured to use a traditional client/server communication model (one-to-one) for remote procedure calls (RPC) or remote method invocations (RMI), although Carmen and Orca also use a publish/subscribe model (one-to-many). The latter is usually implemented using a central process (the broker) that delivers published messages to the processes that were previously subscribed to them. Although this mechanism decouples the way in which robotic services communicate, the use of a central broker has a direct impact in the overall performance of the system, reducing also its fault tolerance.

As software for robotics falls into the category of DRE systems, Orocos and Miro include mechanisms to ensure predictability and fully deterministic behavior. 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 [17], an open source implementation of the real-time CORBA standard [8] in C++. TAO provides predictable end-to-end quality of service in a modular and flexible design. It has an ORB that supports real-time concurrency and real-time event services [6] for CORBA.

Although CORBA (or Ice-based) applications are sometimes referred to as components, they are not exactly equivalent. Components are autonomous and loosely coupled pieces of software which allow developers to create more modular and extensible software. Therefore, the Object Management Group has standardized the CORBA Component Model (CCM) [9]. CCM aims at creating component-based distributed systems which communicate using well-defined CORBA interfaces. A component is described using the Component Implementation Definition Language (CIDL), an extended version of the CORBA IDL. CCM components are composed of attributes, facets (provided interfaces) receptacles (dependencies on external interfaces) and event sources and sinks. Components run within containers, which provide their runtime environment and are also responsible for local and remote communications (using an ORB). CCM is now a mature specification for component-based developments and it is already available in several open source middlewares such as the Component Integrated ACE ORB (CIAO) [18], which supports the TAO real-time ORB, making it possible to maintain predictable behavior. Nevertheless, The CORBA Component Model (and its associated specifications) are not widely used. In spite of that, the OMG aims at their future adoption within the robotics community by standardizing

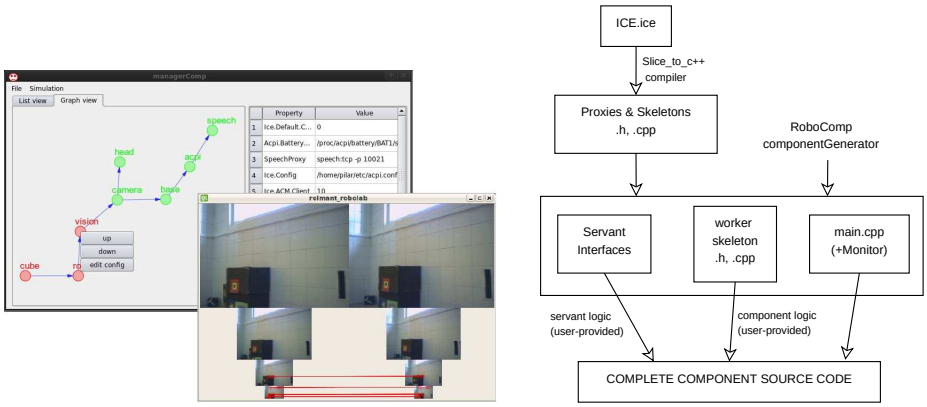

Fig. 1. RoboComp management tools (left) and its component generation process (right)

its novel Robotic Technology Component Specification [12], which focuses on the structural and behavioral features required by robotics software as a supplement to a general component model.

Meanwhile, some of the robotics frameworks described above have defined their own component model, such as Orocos and RoboComp. The components in Orocos are not language-independent and must be implemented directly in C++. However, RoboComp takes advantage of the Ice IDL capabilities to define and implement component interfaces for several programming languages. The following section discusses some of the main features of the RoboComp component model and its powerful management tools.

3 Improving robotics software with DDS

This section describes the improvements we have made to the RoboComp framework to be DRE-compliant. Therefore, we have recommended and justified the use of the DDS standard (and one of its open source implementations) as the most appropriate real-time middleware for RoboComp.

3.1 The RoboComp framework

RoboComp is a general purpose, open-source and component-based robotics framework. It provides ease of use and rapid development of robust software by providing a wide set of management tools that help robot software developers in most of their everyday tasks. RoboComp is also designed to be used as a hardware abstraction layer (HAL) for sensors and actuators (like Player) by standardizing some of its component interfaces.

As mentioned in the previous section, RoboComp relies on the Ice middleware, which has a wide language and platform support. Ice allows developers to

design distributed applications with a high-degree of fault-tolerance and security, using remote exceptions and secure connections, respectively. It supports RMI (in both synchronous and asynchronous mode) and publish/subscribe communications (using a centralized broker called IceStorm).

The point where RoboComp departs from Orca, or other Ice-based frameworks, is in its component model and in its configuration model. RoboComp components are composed of four modules. The first module is the main procedure, which acts as a ready-to-use runtime environment and as the component container. The second module includes the Servant classes that inherit from the skeletons generated by the Ice IDL compiler. They must implement the behavior associated with the interface operations. The third module is the *Monitor* class, a thread in charge of the initialization procedures which checks that the component is ready for a safe execution. The fourth module is the *Worker* class, which is the component main class. It encapsulates its behavior including the communication with other components using Proxies (the RMI stubs generated by the IDL compiler).

Fortunately, RoboComp provides developers with an automatic component generator tool (*componentGenerator*), a command line application which generates code from IDL files. Users only need to specify the interface names of the other components that will be accessed during the execution. The complete process is depicted in fig. 1 (right).

RoboComp also includes powerful management tools. The *component management tool* (fig. 1-left in background) is a visual application which displays the status of the whole system as a graph of interacting components. Each node in the graph is depicted with a color that represents the activity/inactivity of the component. Upon demand, it is also able to show the configuration properties of each component, and to start or to stop them while respecting their dependencies. The *monitoring tool* (fig. 1-left in foreground) allows users to write and perform unitary tests to RoboComp software components. This tool allows programmers either to include their own monitoring code or to use one of the templates available to test HAL components, making it possible to display the outputs of the tested component. In order to analyze the behavior of a component or a group of interacting components, users can benefit from the logging facilities of the *loggerComp tool*, which includes filters to display logs with different criteria, such as date, priority or sender. Finally, the output of RoboComp components can be recorded (and subsequently replayed) by the *replayComp tool*. Thereafter, components can connect to the replayComp application and obtain previously recorded data. This tool is really useful for off-line debugging, that is, when there is no robot hardware available. Besides, RoboComp provides support for some well-known simulators, such as Stage and Gazebo (from Player).

Despite the powerful component and configuration models provided by RoboComp, its wide adoption could be compromised if it does not fit well the stringent real-time and performance requirements of a DRE system. Fig. 2 shows some results that confirm this statement. They correspond to some experiments that

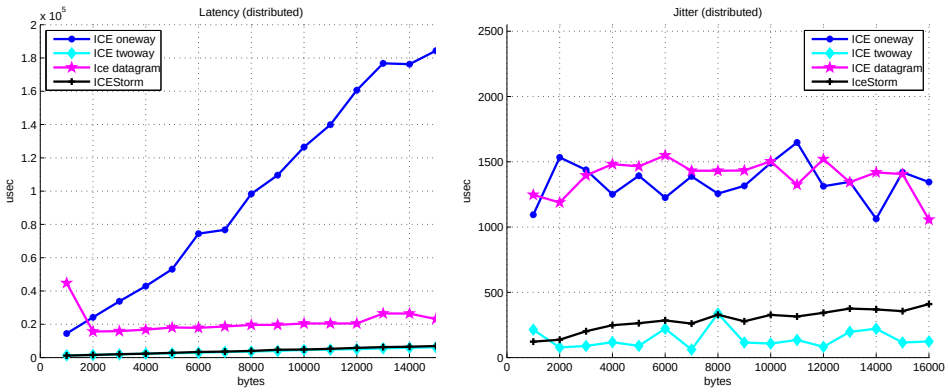

Fig. 2. latency and jitter results in RoboComp for different transport configurations

measure round-trip latency and jitter in a distributed scenario. This scenario consist of two nodes in a local area network (gigabit ethernet), which are executing two components: a sender and a receiver. The results are obtained for different configurations which use the underlying Ice communication protocols. So called *oneway* and *twoway* configurations use the TCP transport protocol, whereas the *datagram* configuration uses UDP. A comprehensive analysis on the advantages and disadvantages of these transport protocols is beyond the scope of this article, although it is surprising that the results are better for the *twoway* mode, and it is also worth noting that TCP is not the best choice to implement real-time distributed services (figs. 4 and 5 will give a closer look at these values).

The main conclusion is that the combination of the Ice proprietary RMI protocol with TCP gives results that are worse than what it should be allowed for very precise real-time configurations (especially for jitter whose values in the experiment range between 77 and 337 microseconds). Thus, it becomes clear that RoboComp needs an alternative middleware with real-time and high-performance features.

3.2 The Data Distribution Service

The OMG published the Data Distribution Service for Real-time Systems standard in 2004. This specification focus on describing a middleware based on the publish/subscribe model for distributing data with high-performance on real-time environments, where systems must be predictable and deterministic. Since its inception, the DDS has been used in defense and aerospace applications, radar processes, naval combat management or air traffic control systems, among others.

The DDS 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) [14]. The former defines the DDS architecture, partici-

pants and standard API, along with some profiles which enhance its use [10]. 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 a 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 defined as *topics*. Topics are described as type-safe data structures, which also contain a key (to identify different topic instances) and an associated quality of service. These QoS policies specify resource limits for delivery, liveliness or reliability, among other features. Moreover, publishers and subscribers can also have QoS associated, which must be compatible before a communication takes place. As in other OMG specifications, the definition of topics is done using an IDL (in this case, a subset of the CORBA IDL), which can be compiled to primitives and data structures for specific programming languages.

After the increasing success of the DDS standard, the OMG is trying to incorporate its main benefits to the Corba Component Model. Therefore, it is now defining the so-called DDS4CCM specification [13] (now in beta 2), where CCM will benefit from DDS features using special artifacts called *connectors*. The connectors will overcome the intrinsic limitations of the CORBA IDL interfaces, which originally were intended to work with a one-to-one communication model.

Regarding the availability of DDS implementations, there are a few commercial products but also some open source ones, such as OpenSplice DDS from Prismtech (LGPLv3 license) and OpenDDS from OCI (BSD-like license). The former middleware is the most complete one in its free version (referred to as *Community Edition*), which has support for the complete DCPS and the Interoperability Wire Protocol in C, C++, C# and Java. Therefore, we have selected OpenSplice DDS in C++ to improve the RoboComp framework.

3.3 Mapping RPC/RMI concepts to DDS

In order to hide the DDS API and its communication model from developers, we propose a methodology which maps existing RPC/RMI communication semantics to their equivalent publisher/subscriber operations. First of all, we have to emulate the RMI message exchange. A client request will be equivalent to the publication of a topic instance (with a specific random key identifier). The server (part of the component runtime environment) subscribes to this kind of topic and *i)* executes the reception asynchronously, *ii)* invokes the appropriate method of the component *Worker* instance with the input provided by the received data and *iii)* prepares a response topic whether needed (for instance, if the equivalent RMI operation returns a non-void type or if it contains output parameters) and then publishes it using also the same key that was found in the request. The client will wait for the response, which is perfectly identified by its matching key value.

Table 1 enumerates our proposed mappings. For the sake of clarity we have used a simplified notation that does not match exactly with the Ice and Open-

RMI interface	DDS request data type	DDS response data type

[return_type] operation();	data_operation_request {
	long id;
	};
		//only if return_type!=void
		data_operation_response {
		long id;
		return_type value;
		};

[return_type] operation(type param);	data_operation_request {
	long id;
	type param;
	};
		//only if return_type!=void
		data_operation_response {
		long id;
		return_type value;
		};

[return_type] operation(out type param);	data_operation_request {
	long id;
	};
		data_operation_response {
		long id;
		[return_type value;]
		type param;
		};

[return_type] operation(type iparam,
out type oparam);
	data_operation_request {
	long id;
	type iparam;
	};
		data_operation_response {
		long id;
		[return_type value;]
		type oparam;
		};

Table 1. Mapping RMI operations to DDS topics

Splice IDL grammar, and he symbols enclosed in brackets are optional. The first column represents the four main types of method signatures (called *operations*), which are part of the interfaces defined in the Ice IDL file. The second and third columns represent their equivalent DDS topics which implement the emulated request/response protocol. As mentioned above, the response is not always needed and, therefore, topics in the third column (and their associated publish/subscribe operations) will not be used.

In order to obtain the required DDS topics, we have designed a new Ice IDL to OpenSplice DDS IDL compiler. Fortunately, Ice has a modular compiler that implements the Visitor pattern [4] to navigate through the abstract syntax tree available with the IDL file. Our compiler makes extensive use of it. Until now, it supports only a subset of the Ice IDL language (interfaces without inheritance). This decision has been motivated by the usual interface descriptions found in the existing repository of RoboComp components. However, a more complete compiler to DDS is under development.

3.4 Implementation and discussion

Fig. 1 (right) has shown the development process of a RoboComp component. Our proposed extension now includes DDS capabilities and is depicted in fig. 3, where dark boxes represent the files and modules that are modified. First of all, we have extended the capabilities of the Ice IDL compiler in order to generate a DDS IDL file that follows the mapping procedures described in table 1. Therefore, we obtain the appropriate definition of DDS topics which will be the input to the OpenSplice IDL compiler to C++. The resulting files implement the DDS publishers, subscribers and other data structures and procedures that will work with our defined topics using the DDS API.

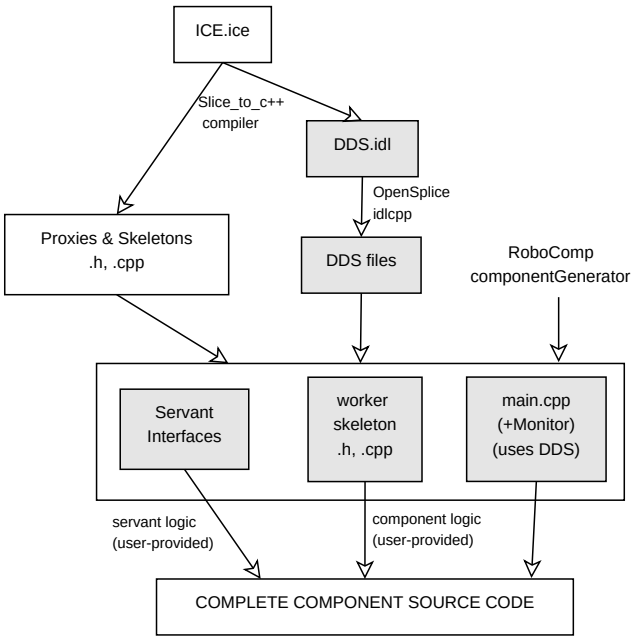

Fig. 3. The improved component generation process with DDS

Our next modification affects the RoboComp code generation script, which now needs to generate source code to implement the subscription and publication of the request/response topics using DDS. From a developer's point of view, the new code is still backward compatible with existing RoboComp components, that is, the Worker class can still use the traditional way of invoking remote operations as client using any Ice Proxy. However, these Proxies have been extended to include an extra method for every existing operation. These new methods share the signature of the original ones, although their names incorporate the suffix *_rt* (real-time), which means that they will use our DDS communication mechanism instead. Regarding the execution environment of the RoboComp module, the main procedure activates the servant objects, but also includes the automatic subscription to every request topic available with the DDS IDL.

Figs. 4, 5 and 6 show our experiment results after comparing communications using the original Ice-based RoboComp RMI mechanism and using the new DDS one for two scenarios: local (figures on the left) and distributed (figures on the right). The experiments consisted of a client component using the middleware facilities for sending messages (with different sizes) to a server component. This server implemented an echo service that sent every message back to the client. The middlewares were used with equivalent configurations: Ice with TCP and two-way mode, and DDS with reliable and ordered delivery.

It is worth noting that figures 4 and 5 show how latency and jitter are bounded and stable in the DDS case. In fact, latency is reduced to be less than

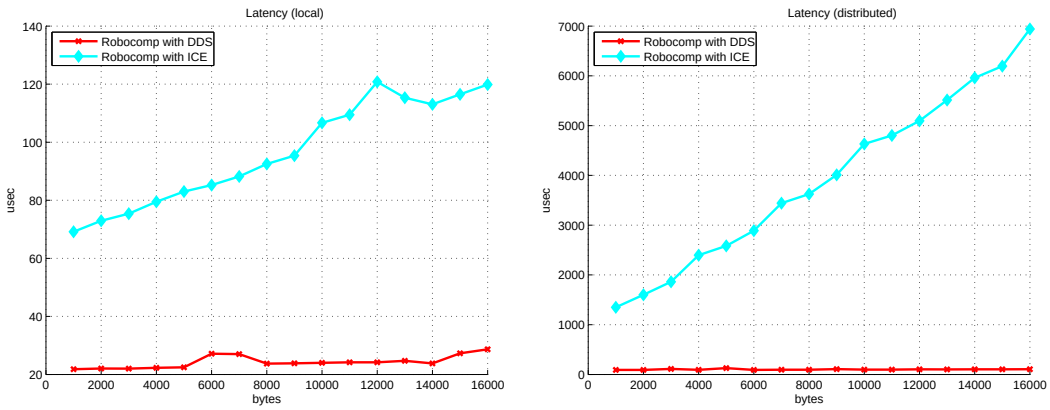

Fig. 4. Latency measurements for RMI and DDS communications in RoboComp

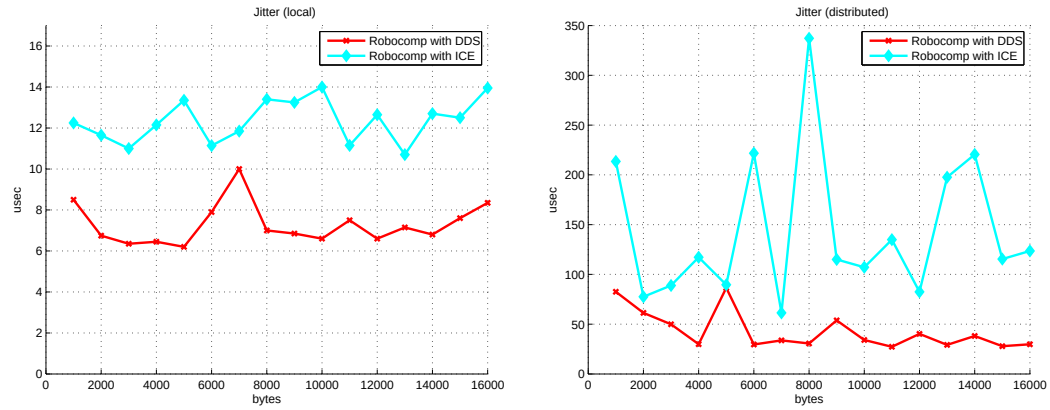

Fig. 5. Jitter measurements for RMI and DDS communications in RoboComp

29 microseconds for local communications and less than 128 in the distributed scenario (with a mean of 105 microseconds). These are impressive results with respect to their corresponding numbers for RMI. The results in the local scenario are also justified by the use of shared memory as the interprocess communication method in OpenSplice DDS. Therefore, we can conclude that every RoboComp component that uses DDS now can be DRE-compliant. The same good results are obtained for throughput in fig. 6, which also means that DDS uses the protocol with the lowest overhead in networked scenarios.

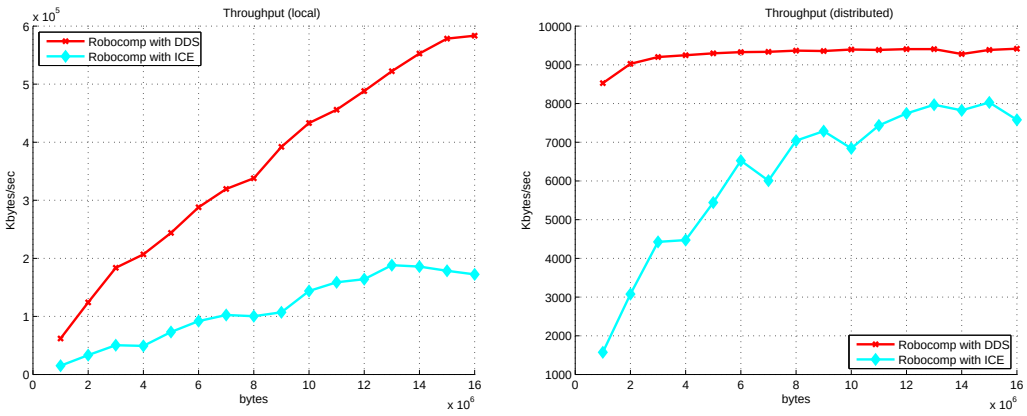

Fig. 6. Throughput measurements for RMI and DDS communications in RoboComp

4 Conclusions and Future Work

This paper has described our experiences on improving an existing robotics framework with real-time and high-performance features. Our main conclusions are that robotic software modules can benefit from the recent Data Distribution Service standard and from the emerging open source toolkits that implement it, such as OpenSplice DDS or OpenDDS. After the improvements, we have demonstrated how DDS middlewares allow the communication among robotic components with low latency and jitter, but also with a throughput that outperforms the one obtained with the previous protocol for RMI. We have developed a methodology that is little intrusive with respect to the existing IDL compilers and RoboComp code generators and that guarantees backward compatibility with previous deployed components. Besides, DDS now adds quality of service features to the framework. We plan to work intensively on exploring and adjusting these new DDS QoS features in order to understand which configurations are better suited in the context of the robotics software.

References

1. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Orebäck, A.: Orca: A component model and repository. In: Brugali, D. (ed.) Software Engineering for Experimental Robotics. Springer Tracts in Advanced Robotics, Springer - Verlag (April 2007)
2. Bruyninckx, H., Soetens, P., Koninckx, B.: The real-time motion control core of the Orocos project. In: IEEE International Conference on Robotics and Automation. pp. 2766–2771 (2003)
3. Enderle, S., Utz, H., Sablatng, S., Simon, S., Kraetzschmar, G., Palm, G.: Miro: Middleware for Autonomous Mobile Robots. In: In Telematics Applications in Automation and Robotics (2001)

4. Gamma, E., Helm, H., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Pub Co. (1995)
5. Gerkey, B.P., Vaughan, R.T., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In: In Proceedings of the 11th International Conference on Advanced Robotics. pp. 317–323 (2003)
6. Harrison, T., Levine, D.L., Schmidt, D.C.: The design and performance of a real-time corba event service. In: in Proceedings of OOPSLA '97, (Atlanta, GA), ACM. pp. 184–199. ACM (1997)
7. Montemerlo, M., Roy, N., Thrun, S.: Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit. In: In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS). pp. 2436–2441 (2003)
8. Object Management Group: Real Time CORBA (2005)
9. Object Management Group: CORBA Component Model (CCM), version 4.0 (2006)
10. Object Management Group: Data Distribution Service for Real-time Systems (DDS), version 1.2 (2007)
11. Object Management Group: Common Object Request Broker Architecture (CORBA/IIOP) (2008)
12. Object Management Group: Robotic Technology Component (RTC), version 1.0 (2008)
13. Object Management Group: DDS for Lightweight CCM (DDS4CCM), in process version 1.0 beta 2. Available at: http://www.omg.org/spec/dds4ccm/1.0/Beta2 (2009)
14. Object Management Group: The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol specification, version 2.1 (2009)
15. Pablo Bustos and Pilar Bachiller and Luis Manso: RoboComp Project. Available at http://sourceforge.net/apps/mediawiki/robocomp (2010)
16. R. Srinivasan: Request for Comments 1832: XDR: External Data Representation Standard (1995)
17. Schmidt, D.C., Gokhale, A., Harrison, T.H., Levine, D., Cleeland, C.: Tao: a high-performance endsystem architecture for real-time corba (1997)
18. Wang, N., Schmidt, D., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J., Schantz, R., Gill, C.: Qos-enabled middleware, in middleware for communications, edited by qusay mahmoud (2003)
19. ZeroC: Internet Communications Engine. Available at http://www.zeroc.com/ (2010)