Experimental Evaluation of the Real-Time Performance of Publish-Subscribe Middlewares

Tizar Rizano, Luca Abeni, Luigi Palopoli
Dipartimento di Scienza e Ingegneria dell'Informazione
University of Trento, Trento, Italy
tizar.rizano@unitn.it, luca.abeni@unitn.it, luigi.palopoli@unitn.it

***Abstract*—The integration of the complex network of modules composing a modern distributed embedded systems calls for a middleware solution striking a good tradeoff between conflicting needs such as: modularity, architecture independence, re-use, easy access to the limited hardware resources and ability to respect real–time constraints. Several middleware architectures proposed in the last years offer reliable and easy to use abstractions and intuitive publish-subscribe mechanism that can simplify system development to a good degree. However, a complete compliance with the different requirements of assistive robotics application (first and foremost real–time constraints) remains to be investigated. This paper evaluates the performance of these solutions in terms of latency and scalability.**

I. Introduction

The recent developments in sensing and battery technologies and in embedded computing devices are creating the premises for the development of low cost robotic applications for a consumer market. The ever-increasing presence of robot vacuum cleaners in our homes, of robotic toys amusing our children, of robotic drones shooting impressive pictures from surprising points of view are witnesses of a clear market trend. At the forefront of this movement are robots created to assist older adults or people with different disabilities. One of the basic needs that can effectively be addressed by assistive robots is personal mobility.

These embedded systems integrate several modules and rely on different types of sensors that convey information on the surrounding environment. For example, they can use video sensors to detect moving objects or obstacles, or can use gyroscopes encoders, 3D cameras and RFID readers for localisation purposes. The same level of complexity is on the software architecture, that can include modules for video-analysis, mission planning, short term planning and control. All these services might interact with other components such as a geo spatial database that stores relevant information about the environment (in this case, the geo spatial database maintains a consistent description of the environment, where each model inserts additional information layers).

The integration of this complex network of modules calls for a middleware solution striking a good tradeoff between conflicting needs such as: modularity, architecture independence, re-use, easy access to the limited hardware resources and real–time constraints.

Several middleware architectures proposed in the last years offer reliable and easy to use abstractions and intuitive publish-subscribe mechanism that can simplify the development of complex robotic applications to a good degree. Examples are OpenDDS[1], which implements a standard proposed by the Object Management Group[1], ZeroMQ [2], which implements a publish-subscribe paradigm to support concurrent programming over socket connections using a publish-subscribe paradigm and is freely available[2], and ORTE [3], which implements a publish-subscribe mechanism over a real–time Ethernet connection (in particular, it is compliant with the RTPS - Real-Time Publish-Subscribe - protocol).

The three mentioned solutions have different reasons of interest: OpenDDS builds on top of the decennial experience made by the CORBA community and offers powerful abstractions, ZeroMQ is extremely lightweight and potentially interesting for its easy adaptation to embedded architectures, and ORTE is a product has been developed for a special care for its real–time performance.

Based on some previous experience [4], this paper evaluates the performance of the three middlewares in terms of latency, scalability, and communication throughput. This comparison will be used as a cornerstone for the development of a reliable software architecture for the DALi cognitive walker (cWalker), an embedded device designed to assist adults with non-severe cognitive abilities in the navigation of complex and crowded environments (e.g., an airport or a mall), which challenge the sense of direction and generate anxiety. However, this work is not limited to the cWalker, but is aimed at increasing the diffusion of real–time middlewares in a large class of robotic applications.

The rest of the paper is organised as follow. Section II offers a high level overview of the case study. Section III, shortly describes the three middleware analysed in the paper and compares their features. Section IV, reports the experimental results on the performance comparison between the

[1]http://www.opendds.org
[2]http://www.zeromq.org

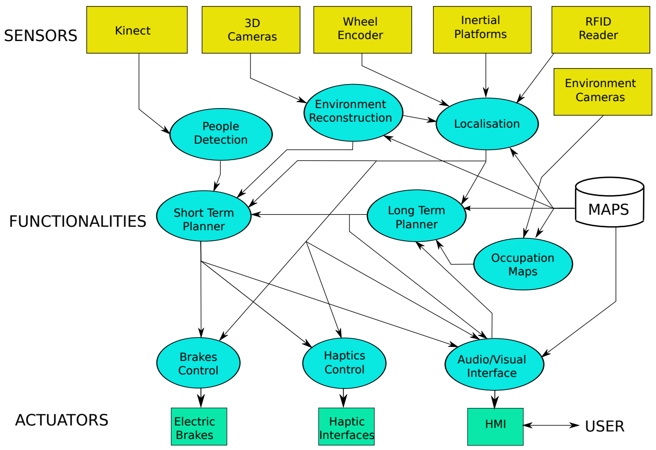

Figure 1. Simplified functional scheme of the DALi cWalker.

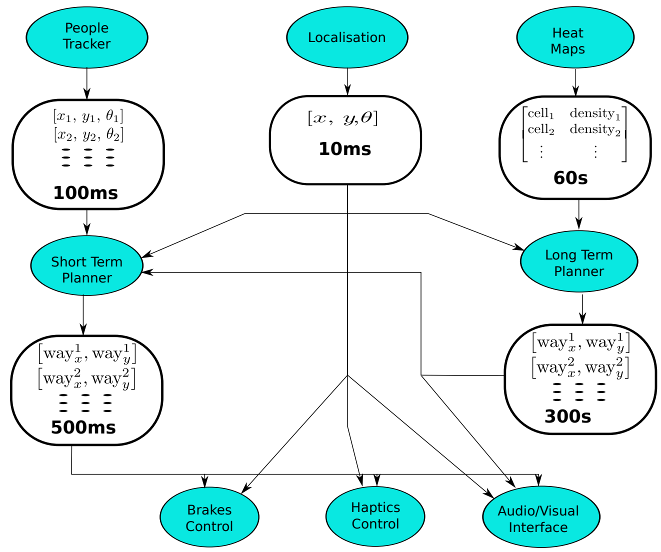

Figure 2. Publish-Subscribe architecture for some of DALi's components.

three different alternatives. Finally, Section V, presents some conclusions and a short discussion of future work directions.

II. Case–Study

An important motivational example for this work has been offered to us by a cooperative European project [3] coordinated by the University of Trento. The objective of the project is the development of a robotic assistant to help older adults with emerging cognitive impairments navigate large and challenging environments (e.g., a shopping mall, or an airport). Because the main focus of the project is to compensate for cognitive deficiencies, the assistant is called cWalker (cognitive walker). A simplified scheme of the most important functionalities of the cWalker is shown in Figure 1. The cWalker prompts the user for a sequence of target points in the environment that he/she wants to visit through a visual interface. The Long Term Planner finds the most convenient path using the map of the environment and the real–time information on the state of the place, which is acquired querying remote sensors (e.g., the surveillance cameras). When the users starts to move, the walker guides her/him along the path using electro-actuated brakes [5], haptic interfaces and audio/video interfaces. The guidance requires a real–time localisation system which tracks the position of the cWalker while it moves. Along the way, the cWalker localises the user in the environment, detects anomalies and the motion of people in the surroundings and plans deviation from the planned path when required (e.g., to avoid accidents or such behaviours as could violate the social rules). These tasks are performed by a Short–Term planner.

A description of the different functionalities is beyond the goals of the present paper, and can be found in previous work [4].

[3] http://www.ict-dali.eu

III. Publish-Subscribe Middlewares

The functional architecture described in Figure 1 suggests the following considerations:

1) Many of the components are re-usable across a wide family of applications and systems (e.g., the localisation module and the people tracker);
2) The computational demand and the physical constraints call for a distributed hardware implementation, in which the functionalities could be deployed in different nodes in different implementations or operating conditions (e.g., in response to a system failure);
3) The different components require varied expertise; the resulting development team is large and heterogeneous.

These requirements can be fulfilled by adopting a middleware infrastructure that implements publish-subscribe functionalities. Moreover, this solution simplifies the development and testing of the various modules, by permitting to decouple their development.

Figure 2 shows a possible implementation scheme for the communication between some of the modules. As an example, the people tracker publishes a sequence of positions and velocity of the people within the reach of the sensors with a periodicity of 100ms and this topic is subscribed to by the short term planner. The localisation module publishes a new position of the cWalker every 10ms and this information is used by various subscribers (at least those shown in the figure). Similarly in the graph one can read the topics published and subscribed to by other modules.

Since the cWalker modules are characterised by some real-time constraints (as shown in the previous example), the middleware implementing the publish-subscribe mechanism needs to be predictable and has to provide reasonable upper bounds for the communication latencies without compromising the throughput. Hence, the middleware has to be explicitly designed to support real-time communications. While the idea of real-time publish-subscribe communication is not new [6], a systematic comparison of multiple *open-source* alternatives is still missing.

The Object Management Group (OMG) published various standards regarding real-time data exchange based on a publish-subscribe protocol. In particular, the Data Distribution Service (DDS) standard defines a service for distributing application data between tasks (in distributed applications), and the Real-Time Publish-Subscribe (RTPS) standard defines an application-level protocol based on UDP/IP, which can be used for the real-time communications required by DDS.

The DDS specification defines both an application level interface for a service implementing the publish-subscribe functionalities (in real-time systems) and an additional layer that allows distributed data to be shared between applications based on DDS. The first interface (Data-Centric Publish-Subscribe - DCPS) is in charge of efficiently delivering the proper information to the proper recipients (according the publish-subscribe) and introduces a *global data space* to be used by applications for exchanging data.

The second part of the standard (Data-Local Reconstruction Layer - DLRL) is a higher level software layer based on DCPS and uses it to construct local object models on top of the global data space.

DDS does not specify a specific "wire protocol" to be used for data exchange and control, hence different DDS implementations can use different (and incompatible) protocols, being them TCP-based, UDP-based, or something different (for example, 2 modules running on the same node can communicate through shared memory to improve the performance).

RTPS is a possible wire protocol to be used by DDS (technically speaking, it is an application-level protocol, generally based on UDP). The RTPS protocol has been designed focusing on real-time requirements, hence it allows to trade the reliability of message delivery for low latencies. As a result, it often implements real-time communications on top of unreliable and connectionless transport protocols such as UDP (although TCP can also be used - see OpenDDS below). The protocol supports publication and subscription timing parameters and properties to allow some performance vs reliability trade-offs.

When using DDS, a publisher and a subscriber communicate by writing/reading data identified by two parameters: *topic* and *type*: the topic is a label that identifies each data flow while the type describes the data format.

To provide good real-time performance (and to properly scale, without having the communication latency affected by the number of publishers or subscribers), DDS and RTPS do not rely on an active service that receives messages from the publishers and forwards them to the proper subscribers. Instead, peer-to-peer connections between each publisher and the interested subscribers are created, based on a naming service that can be provided by some dedicated daemon.

Finally, DDS provides automatic data serialisation through an Interface Definition Language (IDL) compiler, so that components running on different architectures can easily interoperate and communicate (notice, however, that this feature is not strictly needed in the DALi context, since the distributed architecture is based on uniform nodes).

One of the goals of this evaluation is to quantify the overhead (if any) introduced by the various DDS and RTPS abstractions, in order to understand their costs and their benefits. Hence, three different middlewares (ranging from one that is fully compliant with DDS to one that is not compliant with any standard) have been considered: OpenDDS, ORTE, and ZeroMQ.

OpenDDS is fully compliant with the DDS standard forces to use the IDL compiler to serialise the data to be exchanged. ORTE is less flexible, but still implements the RTPS protocol (and is explicitly focused on respecting real-time constraints). Finally, ZeroMQ is not compliant with any specific standard, does not provide a naming service, but relies on simplicity to provide good performance. Hence, comparing the three middlewares allows to evaluate the cost and the benefits of the various features described in the standards and to estimate the overhead that the various features and abstractions might introduce. In more details:

OpenDDS
: is an implementation of DDS v1.2 using RTPS as a "wire protocol" (according to the DDS-RTPS standard v2.1). Both UDP and TCP can be used as a transport protocol below RTPS. It is implemented using the C++ language and is based on CORBA (using ACE/TAO) for the naming and discovery service and for serialising the data (through the TAO IDL). This allows OpenDDS to provide cross platform portability and to easily implement the DCPS layer;

ORTE (the Open Real-Time Ethernet)
: is a lighter implementation of the RTPS protocol which does not rely on external software and directly implement RTPS using UDP sockets. Serialisation can be performed directly by the application. It is implemented using the C language;

ZeroMQ
: is an open source based messaging library implemented in C++ providing support for the publish-subscribe communication paradigm over TCP. Serialisation is not considered. It is not compliant with any standard, and does not provide any kind of naming service (which is then application's responsibility). It exports an object-oriented API with bindings for various languages e.g. C, C++,

python and Java.

IV. Performance Evaluation

The three middlewares have been compared by evaluating their performance in terms of both worst case and average real-time latencies.

This evaluation has been performed by using some test programs implementing publish-subscribe communication, and using a setup similar to the one described in Figure 2.

Since the specific middleware that will be used in the DALi walker has not been decided yet (but only the needed features have been identified), an abstraction layer providing the needed publish-subscribe functionalities has been developed. Such an abstraction layer exports a simplified API that allow to create publishers and subscribers, publish and receive topics, and perform all the operations needed by the various DALi modules.

In particular, the abstraction layer is written in C++ and its API is composed by:

- A class modelling global data space abstraction, where data is published and received by the subscribers;
- A class modelling a Publisher. This class can be instantiated once a global data space has been defined, and can publish a topic on such a data space;
- A class modelling a Subscriber. Similarly to the publisher class, this class can be instantiated only once a global data space has been defined, and receives messages concerning a specified topic from such a data space.

The global data space class only provide a constructor, a destructor, and two methods to create a Publisher or a Subscriber. When creating a Publisher, it is possible to specify a name for the topic it publishes; the Publisher class then provides a `publish()` method that allows to send messages for this topic. When creating a Subscriber, it is possible to specify the name of the topic to subscribe to; the Subscriber class then provides a `register_callback()` method that allows to specify a callback to be invoked when a message for the specified topic is received.

The C++ classes then hide all of the implementation details (and the middleware API), allowing to write code using the publish-subscribe paradigm without relying on a specific middleware. The abstraction layer currently supports the three middlewares considered in this paper, but extending it to other middlewares based on the publish-subscribe paradigm should be simple.

Some preliminary experiments measured the performance of the middleware without considering the effects of the network (by running the experiments on a single node) and revealed that ORTE seems to perform slightly better than the other middlewares when only few subscribers are active, but ZeroMQ scales better [4]. In any case, on an Intel i7 CPU running at $2.8GHz$ the worst-case measured latency was smaller than $1ms$, for all the middlewares.

In this paper, the experiments have been performed using a setup that is more similar to the DALi hardware and software architecture. First of all, the embedded boards that will probably be used in the DALi cWalker (pandaboards[4], based on an OMAP4460 - powered by an ARM core running at $1Ghz$) have been used. Moreover, the experiments are performed on two identical pandaboards connected via fast ethernet switch (100 Mbps); hence, network effects have been accounted for in the experiments. The two boards run Ubuntu 12.04 with the 3.2.0 Linux kernel.

A first set of experiments, still based on the simple test programs used in the previous paper, compare the real-time performance of the three middlewares by measuring the latency between the generation of a message (from the publisher) and its arrival to the subscribers - this will be referred as "publish-subscribe latency". With respect to the previous experiments, the ones reported here are based on the pandaboard setup described above. First, some "single node" experiments (similar to the previous ones) have been run, and then the measurement have been repeated with the publisher running on one board and the subscribers running on the other one. As in the previous experiments, the middleware abstraction layer has been used to easily repeat the same tests with different middlewares.

The publisher is implemented as a single-threaded process scheduled with `SCHED_FIFO` and the maximum real-time priority. Each subscriber (maximum 4 subscribers) is also a high priority (`SCHED_FIFO`, maximum real-time priority) process. However, the process is multi-threaded, since all of the tested middlewares create at least two threads for each subscribers: main thread and the subscriber listener thread. For OpenDDS, there is an extra thread that run its ORB and several threads for non-CORBA transport IO. OpenDDS and ORTE are configured to use UDP as their transport protocol. However, ZeroMQ is configured to use TCP since UDP is not officially supported.

Figure 3 reports the results (worst-case and average latencies as a function of the number of subscribers) obtained when running publisher and subscribers on the same node. Respect to the results obtained on the x86-based PC, the worst-case latencies are about 10 times larger, and the ORTE behaviour is slightly worse than the ZeroMQ one (in the previous experiments, ORTE behaved better than ZeroMQ for small numbers of subscribers, but ZeroMQ scaled better).

Figure 4 reports the results of the same experiment executed in a distributed environment (publisher and subscribers on 2 different nodes). It is immediately possible to notice that the latencies increase even more, and only ZeroMQ stays below 10 ms in both average and worst case latencies for all the numbers of subscribers. Again, confirming the result obtained in [4] ORTE performs well with a limited number of subscribers while ZeroMQ scales better than the

[4]http://www.pandaboard.org

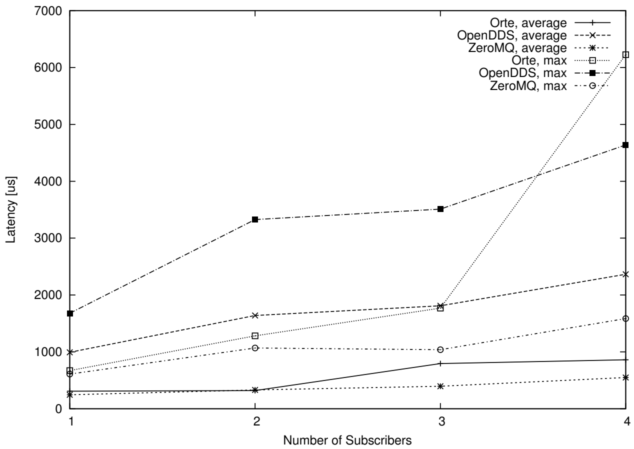

Figure 3. Single node Publisher/Subscriber latency as a function of the number of subscribers.

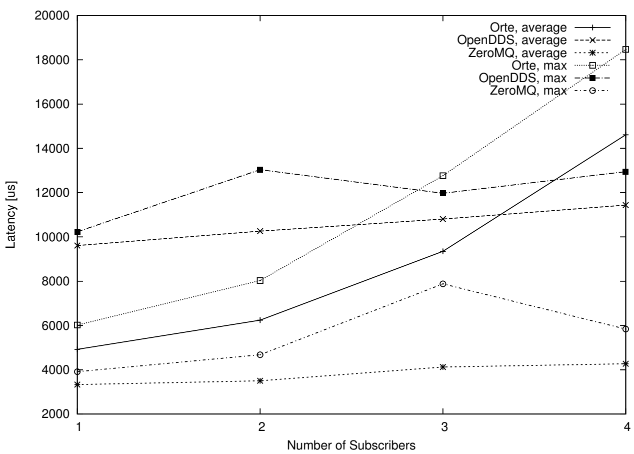

Figure 4. Multi node Publisher/Subscriber latency as a function of the number of subscribers.

other middlewares even in the distributed scenario.

Finally, Figure 5 reports the latencies as a function of the message size, showing that the average latencies of all middlewares scale well with message size up to 1000 bytes.

After running the first experiments with a simplified test application, a more realistic test case based on Figure 2 has been used to compare the three middlewares. The test is composed by 8 processes emulating the 8 software modules that will run on the cWalker: the *People Tracker* (PT), the *Localization module* (LOC), the *Heat Maps* (HM), the *Short Term Planner* (STP), the *Long Term Planner* (LTP), the *Brakes Control* (BC), the *Haptics Control* (HC) and the *Audio Visual Interface* (AVI). All the modules are modelled as periodic real-time tasks running with the periods indicated in Figure 2, subscribing to some topics, and eventually producing messages at each activation.

Each task/software module is statically assigned to a pandaboard, and different ways to distribute the tasks have

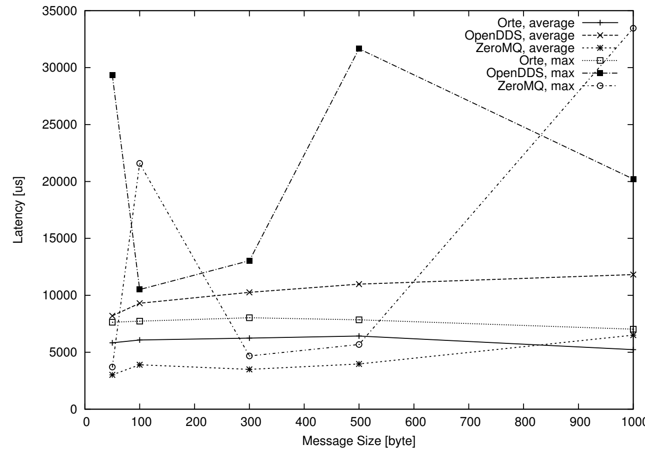

Figure 5. Multi node Publisher/Subscriber latency as a function of the message size

Mapping	protocol	max	avg	stdev
1	ZeroMQ	1740	523.41	183.68
	ORTE	7599	712.39	260.39
	OpenDDS	6135	2016.79	470.46
2	ZeroMQ	6752	2368.42	427.55
	ORTE	10170	4563.44	832.93
	OpenDDS	11268	4952.68	680.27
3	ZeroMQ	7851	3720.68	680.27
	ORTE	11940	5092.19	410.39
	OpenDDS	11482	6179.72	295.92

Table I
LATENCY IN MICROSECONDS

been tested. In particular, the results obtained with three different mappings of modules to embedded boards will be reported:

- **Mapping 1**: All modules run on pandaboard 1
- **Mapping 2**: The AVI, HM, and LTP modules run on pandaboard 1 while BC, HC, PT, LOC, and STP run on pandaboard 2
- **Mapping 3**: The AVI module runs on pandaboard 1 while all the other modules (BC, HC, PT, LOC, STP, HM, and LTP) run on pandaboard 2.

The worst-case and average latencies measured the output of the AVI module are reported in Table I. This set of experiments show the effect of distributed processes on the performance of the middlewares. The average latencies of all middlewares stay below the minimum period of the modules (10 ms). However, the worst case latencies of all middlewares except ZeroMQ are above the minimum period.

V. CONCLUSIONS

This paper presents the performance evaluation of three open-source publish-subscriber middlewares. The evaluation focuses on their real-time performance, to identify the solution that best suits the needs of modern robotic applications based on distributed embedded architectures. The experimental setup was designed taking inspiration from an existing robotic application.

Based on the result of the experiments, ZeroMQ is shown as the most suitable middleware for DALi application. Although the average latencies of both ORTE and OpenDDS are below the minimum period required by DALi application, their worst case latencies is above it. However, Their latencies remain below 7 ms for 99% of the time.

The goals of future investigations are manifold. One of the most important is to extend the analysis to other middleware solutions explicitly developed for robot applications such as ROS [7] and OROCOS [8].

References

[1] OMG, "Data distribution service for real-time systems – version 1.2," The Object Management Group, Tech. Rep., 2007.

[2] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly, 2013.

[3] P. Smolik, Z. Sebek, and Z. Hanzalek, "Orte–open source implementation of real-time publish-subscribe protocol," in *Proc. 2nd International Workshop on Real-Time LANs in the Internet Age*, 2003, pp. 68–72.

[4] T. Rizano, L. Abeni, and L. Palopoli, "Middleware for robotics in assisted living: A case study," in *Proceedings of the 15th Real-Time Linux Workshop*, Lugano, Switzerland, October 2013.

[5] D. Fontanelli, A. Giannitrapani, L. Palopoli, and D. Prattichizzo, "Unicycle steering by brakes: a passive guidance support for an assistive cart," in *Proceedings of the 52nd IEEE Conference on Decision and Control*, Firenze, Italy, December 2013.

[6] R. Rajkumar, M. Gagliardi, and L. Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation," in *Proceedings of the 1st Real-Time Technology and Applications Symposium (RTAS95)*, 1995, pp. 66–75.

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009.

[8] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3. IEEE, 2001, pp. 2523–2528.