μDDS: A Middleware for Real-time Wireless Embedded Systems

Apolinar González · W. Mata · L. Villaseñor · R. Aquino · Jose Simo · M. Chávez · A. Crespo

Abstract A Real-Time Wireless Distributed Embedded System (RTWDES) is formed by a large quantity of small devices with certain computing power, wireless communication and sensing/actuators capabilities. These types of networks have become popular as they have been developed for applications which can carry out a vast quantity of tasks, including home and building monitoring, object tracking, precision agriculture, military applications, disaster recovery, industry applications, among others. For this type of applications a middleware is used in software systems to bridge the gap between the application and the underlying operating system and networks. As a result, a middleware system can facilitate the development of applications and is designed to provide common services to the applications. The development of a middleware for sensor networks presents several challenges due to the limited computational resources and energy of the different nodes. This work is related with the design, implementation and test of a micro middleware for RTWDES; the proposal incorporates characteristics of a message oriented middleware thus allowing the applications to communicate by employing the publish/subscribe model. Experimental evaluation shows that the proposed middleware provides a stable and timely service to support different Quality of Service (QoS) levels.

Keywords Real-time middleware · Real-time distributed systems · Embedded systems

A. González (✉) · W. Mata · M. Chávez
Faculty of Mechanical and Electrical Enigineering,
University of Colima, Colima, Mexico
e-mail: apogon@ucol.mx

L. Villaseñor
CICESE Research Center. Ensenada, B.C. Mexico

R. Aquino
Faculty of Telematics, University of Colima, Colima, Mexico

J. Simo · A. Crespo
Polytechnic University of Valencia, Valencia, Spain

1 Introduction

Miniature computing devices are being embedded in an increasing range of objects around us including home appliances, cars, transport infrastructures, buildings, people, etc. Furthermore, the networking of such embedded environments is enabling advanced scenarios in which devices leverage off each other and exhibit autonomous and coordinated behavior. An RTWDES is formed by a large quantity of small devices with certain computational power, wireless communication and sensing/actuators capabilities [2]. The sensor/actuator nodes are generally disseminated on the region of study, where each sensor/actuator node is responsible for extracting data of the phenomenon of interest, such as, humidity, temperature, pressure, brightness, etc and in some cases it can carry out control actions. The sensor/actuator nodes are capable of processing and sending the collected data to one or more sinks, which are in charge of transmitting the data to the end user application.

The RTWDES have become popular as they have been developed for applications which can carry out a vast quantity of tasks, including home and building monitoring, object tracking, precision agriculture, military applications, and disasters recovery [1, 2, 5, 8, 23]. The paradigm of these nets differs from the habitual ones which are based on the information management in that RTWDESs have knowledge over what is happening in the environment where they are deployed, thus the decision making process depends on the analysis of the sensed variable along the area of interest.

With respect to the development of applications for RTWDESs, a middleware can be used to bridge the gap between the application and the underlying operating systems and networks [9, 14]. One of the basic purposes of any middleware is to satisfy the application requirements; in this work the middleware takes into account the specific features of an RTWDES like the restrictions in energy, communication and computing power [7]; consequently the design and development of the sensor networks is highly related to specific resources like battery, memory and processor capabilities, as well as, the communication models and the application requirements.

Programming language interoperability is the ability of micro Data Distribution Service (μDDS) to interoperate applications written in different programming languages. μDDS implements a subset of standard DDS APIs in C using gcc compiler for Linux and PaRTiKle [21] operating systems and Real-Time Java implementation, specifically jRate [6], which is an implementation of real-time java over Linux and our previous work a porting of jRate over PaRTiKle OS was realized [15]. Data Distribution Service (DDS) provides a platform independent model that is aimed to real-time distributed systems. DDS is based on publish-subscribe communications paradigm in which the components connect information by publishers and subscribers. One of the important features of our approach is the complete development platform which as you can see, the use of C language and a POSIX-Compliant

OS, make it easier to learn to develop RTWDES applications. The small footprint and low overhead obtained in the standalone application, are an important features because is in the order of the kilobytes and that is what it need it in the embedded applications by their limited resources. Most Middlewares do not implement some QoS Policies in their implementations, we establish the basis for implementing subsequently QoS requirements as they are needed.

This paper describes the design, implementation and performance of micro Data Distribution Service (μDDS). It discusses the layered architecture of μDDS, followed by a detailed description of each layer; it also presents a description of the application development process with C and Real-Time Java for the linux and PaRTiKle OS. This paper also evaluates the performance of μDDS by means of throughput and latency tests while considering a wireless network layer and some QoS policies such as TIME_BASED_FILTER and DEADLINE. μDDS is lightweight and efficient, and simplifies the development of a publish/subscribe approach for embedded real-time wireless networks applications. Furthermore, μDDS provides programming language interoperability which is the ability of μDDS to interoperate with applications written in different programming languages. μDDS implements a set of standard DDS APIs in C which makes use of the gcc compiler for Linux and PaRTiKle [21] operating systems and Real-Time Java implementation. specifically jRate [6], which is an implementation of real-time java for Linux. One of our previous works considered the porting of jRate for the PaRTiKle OS [15].

The remainder of the paper is organized as follows. Section 2 presents a description of the middleware proposals presented in the literature. Section 3 presents an overview of the Data Distribution Service for real-time applications. Section 4 details the components of μDDS for real-time wireless embedded systems and the architecture over Linux and PaRTiKle OS. Section 5 presents the design aspects on C and real-time Java. Section 6 presents an analysis of the performance advantages with communications and QoS tests. Finally, Section 7 summarizes the major aspects of this work and outlines future research directions.

2 Related Works

There are several middleware proposals in the literature, some of the most representative are described in this section. Cougar [3] implements a model based on consults, where the sensed data is considered to be in a virtual relational database. Mate [11] is a small virtual machine communication centered approach executed on TinyOS [10] and the developers, by means of the use of this architecture, can change in a simple way the set of instructions, the execution of events and the subsystems of the virtual machines. Impala [12] is a middleware for the ZebraNet project, it is composed of two layers: the upper layer that contains all the application protocols as well as the programs for ZebraNet, while the lower layer contains three middleware agents: the Application Adapter, the Application Updater, and the Event Filter. Garnet [25] presents an architectural framework that provides a data stream centric abstraction. In a fixed network the data is gathered by the applications which use the typical mechanisms of advertising, discovery, registration, authentication and publish/subscribe to identify, subscribe and receive the data streams of interest. It has some components which are receivers, sensors/actuator, filtering and dispatching

services, consumer processes and services, the Super coordinator, etc. MiLAN [16] sits on top of multiple physical networks, and has an abstraction layer that allows network-specific plugins to convert the MiLAN commands to protocols-specific commands which are transmitted through the usual network protocol stack, these plugins are important as they help to determine which sets of nodes satisfy the QoS requirements of the application.These middlewares have been developed for small systems in Wireless Sensor Networks (WSN) applications and do not support real-time characteristics. Real-time capability means that a timely response to time-critical events is assured, so it can use for control applications in industrial environments which require reliable and secure communications.

Publish/subscribe (or pub/sub) is an asynchronous messaging paradigm where senders (publishers) of messages are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into classes, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of which (if any) publishers may be available. This decoupling of publishers and subscribers provides an increased scalability and the support of a greater dynamic network topology.

Mires [24] presents a publish/subscribe model, and incorporates two additional services: routing component and additional services. The communication between the nodes is given in three phases. First, the nodes in the network announce their available topics, such as, temperature or humidity which are collected from local sensors. Second, the advertised messages are routed to the sink node using a multi-hop routing algorithm and the user application can connect to these nodes to monitor the desired topics. Finally, the subscribe messages are broadcasted down to the network nodes. Mires is located on top of the OS, encapsulating its interfaces and providing higher-level services to the node application. TinyDDS [4] is a publish/subscribe middleware for event detection and dissemination applications in WSNs. With its self-configuring event routing protocol, TinyDDS adaptively performs event publication according to dynamic network conditions and autonomously balances its performance among conflicting operational objectives. TinyDDS has the ability to interoperate with applications written in different programming languages. This middleware implements a set of standard DDS APIs in nesC and Java Micro Edition. TinyDDS and Mires do not implement features for real-time distributed embedded systems, its runtime support and programming languages do not include benefits for developing safety critical or real-time applications. On the other hand, our middleware developed using a subset of the OMG DDS specification, on top of real-time operating system with small footprint and low overhead, allows easily develop RTWDES applications, with the advantage of the C programming language and standard specifications, it can also allow migration to other hardware architectures.

This article describes the design, implementation and performance evaluation based on experiments of μDDS, which is a middleware based and compliant with a subset of the DDS implementation included in the PaRTiKLe OS; μDDS also implements a set of standard DDS APIs in C and real-time Java. The C implementation of μDDS operates directly on the POSIX Kernel PaRTiKle platform, whereas the Java implementation operates on the GCJ runtime, which is the GCC Java compiler with the javax.realtime classes and the native methods over POSIX Kernel PaRTiKle.

3 An Overview of Data Distribution Service For Real-time Systems

The OMG Data Distribution Service (DDS) specification [17] standardizes the software application programming interface (API), where a distributed application can use the publish/subscribe communication mechanism which is centered in the data. It is based on Model Driven Architecture (MDA) [18–20], which defines a Platform Independent Model (PIM) that is a view of a system from the platform independent viewpoint, as a result the middleware developers can derive any Platform Specific Model (PSM) which can be adjusted to the application requirements; thus allowing the construction of different DDS implementations dedicated to very specific needs [17].

3.1 DDS Conceptual Model

DDS introduces two levels of interfaces, which are:

- DCPS (Data-Centric Publish/Subscribe), a low-level mandatory API that provides the functionality required for an application to publish and subscribe the values of data objects. This layer provides support for 21 QoS policies as we will see later;;
- DLRL (Data Local Reconstruction Layer), an optional high-level API that allows a simple integration of the Service into the application layer.

According to [20] the advantages of this infrastructure can be listed as:

- It is based on a simple publish/subscribe communication paradigm;
- it has a flexible and adaptive architecture that supports auto-discovery of new stale endpoint applications;
- the low overhead can be used with high-performance systems;
- it has a deterministic data delivery;
- it is dynamically scalable;
- it provides an efficient use of transport bandwidth
- it supports one to one, one to many, many to one and many to many communications;
- and it has a large number of configuration parameters that provide to the developers a complete control of each message in the system.

The information flows with the aid of the constructors as it is shown in Fig. 1. The Publisher and DataWriter are on the sending side while the Subscriber and DataReader are on the receiving side. The Topics are used to provide the basic connection between Publishers and Subscribers. The Topic of a given Publisher on one node must match the Topic of an associated Subscriber on any other node. If the Topic does not match, then the communication will not take place. The Publisher is responsible for the distribution of the different data types, and the DataWriter is used to communicate to a Publisher the existence and value of the data. Meanwhile the Subscriber is responsible for receiving the published data and making it available to the receiving application and the DataReader is used to access the received data [17].

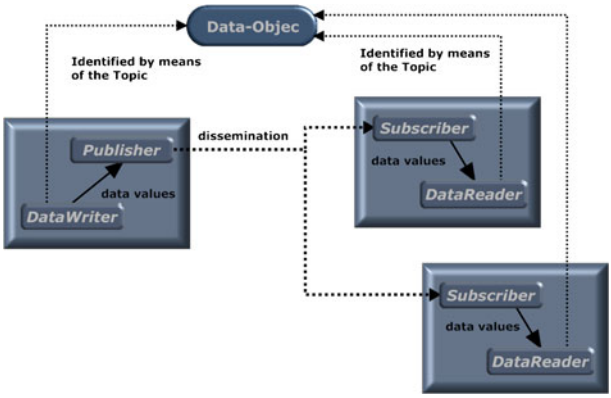

Fig. 1 DDS Entities

3.2 Quality of Service in DDS

One of the important aspects to consider is the Quality of Service (QoS), which is a concept used to specify certain behavior of a service. QoS provides the ability to control and limit the use of resources like network bandwidth, memory, reliability, timeliness, and persistence, among others. The DDS QoS model implements a set of classes which are derived from QoSPolicy. DDS provides USER_DATA QoS policy, TOPIC_DATA QoS policy, DURABILITY QoS policy, DEADLINE QoS policy and other policies. Further details regarding these policies can be studied in [17].

4 Architectural Overview of a Middleware for Real-time Wireless Embedded Systems

The proposed architecture is shown in Fig. 2 , where the PaRTiKle real-time OS is executed directly on an ARM or x86 hardware; nodes can communicate with other devices using a ZigBee communication module; The DDS middleware sits between the node and user applications, and the PaRTiKle OS.

4.1 PaRTiKle OS Architecture

PaRTiKle [13, 15, 21, 22] is a recent embedded real-time operating system designed to be compatible with the POSIX 5.1 standard. PaRTiKle has been designed bearing the following ideas in mind:

- it must be as portable, configurable and maintainable as possible.
- it must support multiple execution environments, thus allowing to execute the same application code (without any modification) under different environments, such as, in a bare machine, a Linux regular process and as a hypervisor domain.
- it must support multiple programming languages; currently PaRTiKle supports Ada, C, C++ and Java

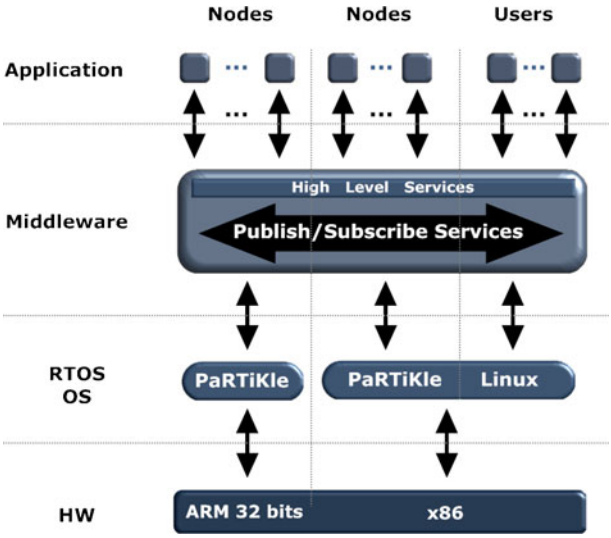

Fig. 2 Global architecture

PaRTiKle has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for this type of applications. Figure 3 shows the PaRTiKle architecture that has been designed as a real-time kernel with a clean and well defined separation between kernel and application execution spaces. All kernel services are provided via a single entry point, which improves the robustness and also greatly simplifies the work to port PaRTiKle to other architectures and environments.

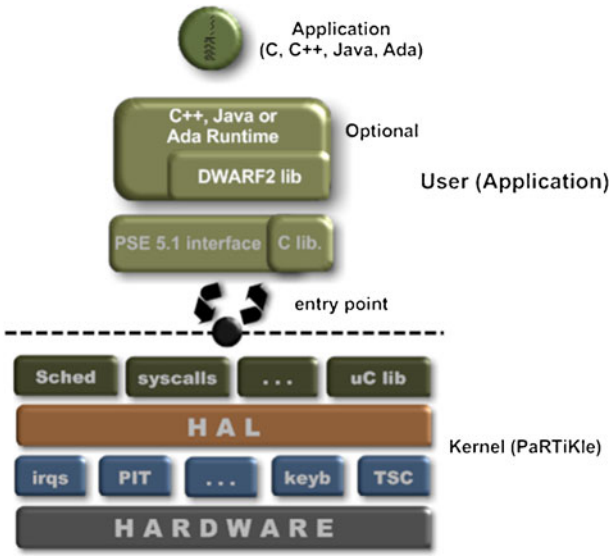

Fig. 3 PaRTiKle architecture

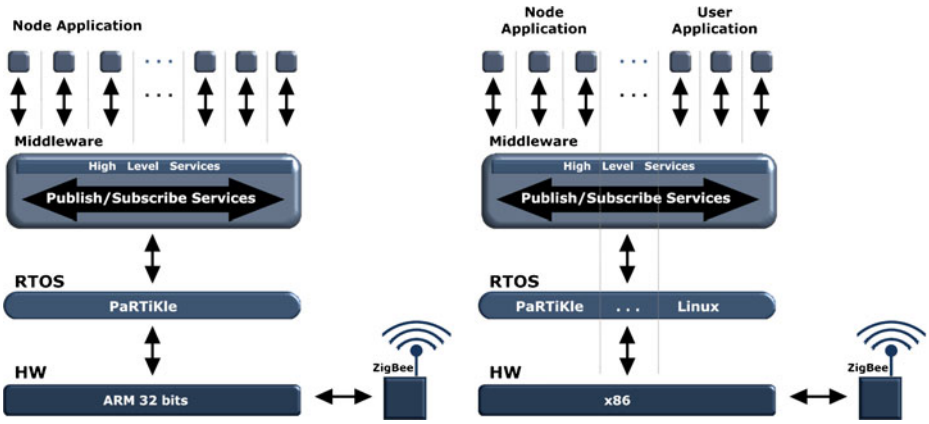

Fig. 4 Communication and advanced node architecture

4.2 μDDS: A Middleware for Real-time Wireless Embedded Systems

μDDS is a publish/subscribe middleware for real-time wireless embedded systems based on DDS specification and implements a subset of standard interfaces for event subscriptions and publication to be used by applications. Applications implemented on top of μDDS can disseminate and collect data through a publish/subscribe interface provided by the middleware. Different routing protocols can be used to implement the overlay network; the middleware is currently implemented on 802.15.4 standard devices which can support star, tree and mesh topologies. The nodes can communicate with other devices using a ZigBee communication module. Figure 4 show how different kinds of nodes with ARM and x86 hardware architectures and zigbee communication module.

5 Building Process with a μDDS Application

The programmer can automatically generate a template for the end application, by means of the autogen tool, which was designed to create program files that contain repetitive text with varied substitutions. The generated codemakes use of a definition file which is completely separate from the template file, the use of this definition file increases the flexibility of the template implementation and the respective generated code. There are three main elements of the development model, DDS Middleware and DDS library, user application and PaRTiKle Kernel. Figure 5 illustrates the development process of a μDDS application running on PaRTiKle which makes use of a bash script named mkkernel. As part of the building process the following steps must be performed: first, the application (i.e. the application previously generated with autogen and modified by the developer with the changes needed for the end user application) must be linked with the μDDS middleware; next, the resulting object is linked with the kernel object to create an executable file (*.prtk) containing all the components.

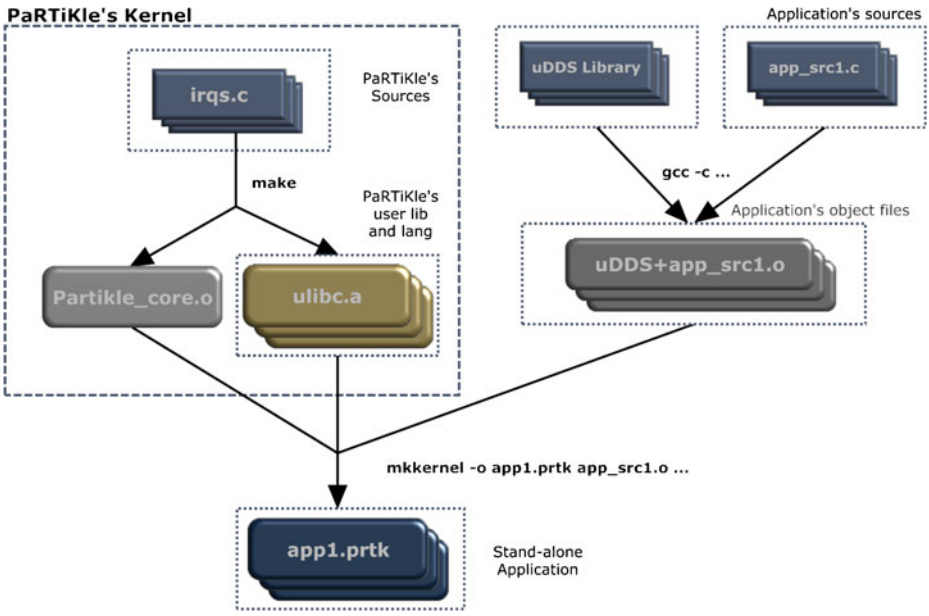

Fig. 5 Building process of a μDDS application

The mkkernel script requires the following parameters:

$ mkkernel -f <output> <file1.o> [<file2.o> ...]

where <output> is the name of the target executable once the process of building the application has concluded, and <file1.o>, <file2.o>, etc., are the object files obtained when the application is compiled using GCC with the option -c. The steps performed by this script are:

1. It links the application against the user C library and the suitable run-time (the run-time is selected depending on the language used to implement the application).
2. It turns every applications symbol into a local symbol, except user entry point.
3. Eventually, the script links the resulting object file together with the kernel object file to create the executable (*.prtk).

For example, consider the source file example.c; this files is compiled, using GCC with the -c flag.

$ gcc -c example.c -I <headers>

Where <headers> is the path to the PaRTiKle C user header files (ulibc/include). The result of this compilation is a file called example.o. After that, we invoke the mkkernel script as follows:

$ mkkernel -o example example.o

The result is a binary file named example.prtk, which is ready to be executed in the selected execution environment.

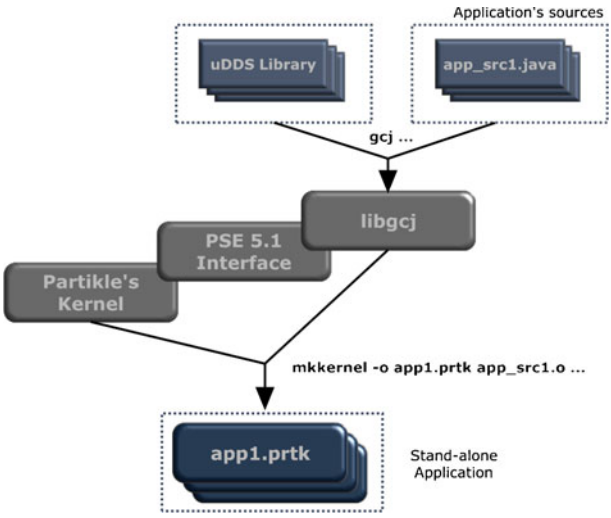

Fig. 6 Java application build process

For a development model in Real-Time Java, a compiled application must include the GCJ runtime with the javax.realtime classes and the native methods as shown in Fig. 6. GCJ, which is the GCC Java compiler, links with the libgcj by default, which includes a complete java runtime on the order of some megabytes with a set of characteristics that are not necessary for small platforms. The idea is to remove any functionality which is not necessary for safety critical systems and also not necessary for real-time systems with hard timing restrictions. We started from scratch, copying the source code necessary to the directories where we had installed GCJ on PaRTiKle. The code was compiled generating a new, reduced, version of libgcj. The size obtained is in the range of hundreds of kilobytes for the complete application, which is composed of the application code (code developed by the end user), DDS Middleware and the PaRTiKle operating system. The reduced version of libgcj is a subset of the runtime support and the CLASSPATH. Finally, this version of libgcj has all of the support to execute applications using the porting and adaption of jRate on the PaRTiKle OS, a subset of java.lang and java.io, support for thread execution, and OS interfaces. The garbage collector, support for graphical interfaces, runtime classloading, bytecode interpretation, reflection, finalization, serialization, file and network I/O, and many parts of java.lang, java.util, and java.io, that were not considered essential, have been removed, for real-time and safety critical applications.

6 Implementation and Evaluation

To study the performance of publish/subscribe systems we implement the μDDS as our middleware base for wireless embedded applications and have been compared to the performances of DDS implemented by the company Real-Time Innovations (RTI), which is one of the most complete and representative implementations of DDS. The purpose of our experiment is to run latency and throughput performance tests with different message size and number of subscribers in a practical

environment. In our middleware we implement a Publisher which is responsible for the dissemination of topics with its respective DataWriter that allows an application to offer samples of a specific topic to the subscribers. Once the topic has assigned values to each field it is necessary to serialize the data which is later sent to the network, to support this functionality we implement a serialize method which packages all the data which is later sent to other nodes through the established network.On the Subscriber side a deserialize method is employed to assemble the topic for its use at the user application. The test was performed using eight devices based on an ATOM N270 processor running at 1.60 GHz, with 1MByte RAM and 802.11 b/g 54 Mbps for the wireless communication, Linux operating systems (Ubuntu 9.10) with kernel version 2.6.31 and gcc 4.1.3 compiler.

6.1 Throughput Test

Throughput is defined as the total number of bytes received per unit time in different 1-to-n (i.e., 1-to-4, 1-to-8, and 1-to-12) publisher/subscriber configurations. In this test, the publisher sends data where the size varies from 16 bytes to 4 Kb and is sent to one or more subscriber applications (1-to-8). The throughput is the total number of messages received per second by all the subscribers in the system divided by message fanout. The test code contains two applications: one for the publishing node and the other for the subscribing node(s). The publisher applications are started first followed by the subscriber applications, then the publication application sends a burst of data and repeats the cycle for a specified duration. The phases of the algorithm is as follows:

1. The publisher signals the subscriber applications that it will commence, and then starts its own clock.
2. The subscriber starts to count the number of messages received.
3. After the desired duration is over, the publisher signals the subscribers that one experiment is over. The subscriber will then divide the number of samples received by the elapsed time to report the throughput observed at the receiver.

Figure 7 shows the performance results were obtained in the throughput tests between μDDS and DDS implemented by RTI. This graph shows sustainable one-to-one (point-to-point) publish/subscribe throughput in terms of network bandwidth (megabits per second). Accounting for Ethernet, UDP overhead, the maximum bandwidth available for message data (and metadata) is slightly over 800 megabits for μDDS and 700 megabit for RTI DDS implementation. Maximum throughput is achieved when the publisher sends as fast as the subscribers can handle messages without dropping a packet. That is, the maximum throughput is obtained somewhere between the publisher sending too slowly and the publisher swamping the subscriber. In general, throughput for Data Distribution Service (DDS) is higher than most other messaging and integration middleware

6.2 Latency Test

Latency is defined as the roundtrip time between the sending of a message and reception of an acknowledgment from the subscriber. In our test, the roundtrip latency is calculated as the average value of 10,000 round trip measurements.

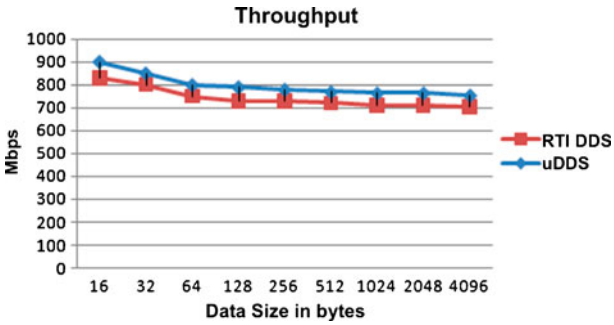

Fig. 7 Throughput test

Similarly to the throughput test, the publisher sends data with varying size from 16 bytes to 4 Kb to one or more subscriber applications in which the latency is estimated as a half of the roundtrip time of a message. The test code contains two applications: one for the publishing node, and one for the subscribing node(s). The publisher application is started first, followed by each subscriber, then the publisher starts publishing data. The test ends when all the messages have been sent and the same number of replies has been received by the publisher. Figure 8 shows the average one-way latency in microseconds for publish/subscribe messaging for μDDS and RTI DDS implementation (DD1). This data shows that, at small message sizes, latency remains consistently low. At larger messages sizes, which are network-limited, latency is proportional to message size. The main reason for the result in Fig. 8, standard DDS has fewer layers than other standard pub/sub platforms, so it incurs lower latency. Both graphs show the same trend, however, μDDS has a much lower latency because it implements a subset unlike the implementation done for RTI, which implements the full DDS specification. From this point of view, μDDS has a lower overhead, additionally it can be easily ported to other architectures, mainly to support real-time applications and safety critical systems.

6.3 QoS Test

For the QoS policy, μDDS implement some of the QoS model of DDS, such as deadline and time based filter. The Deadline QoS indicates the minimum rate at

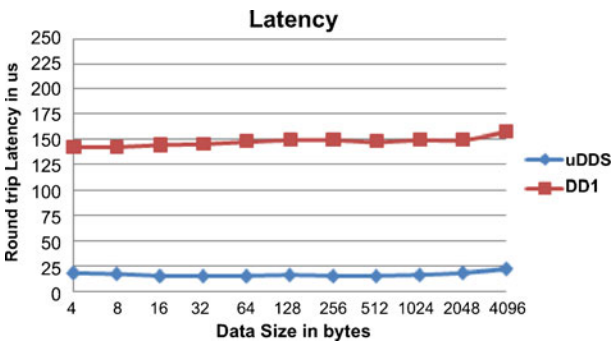

Fig. 8 Latency test

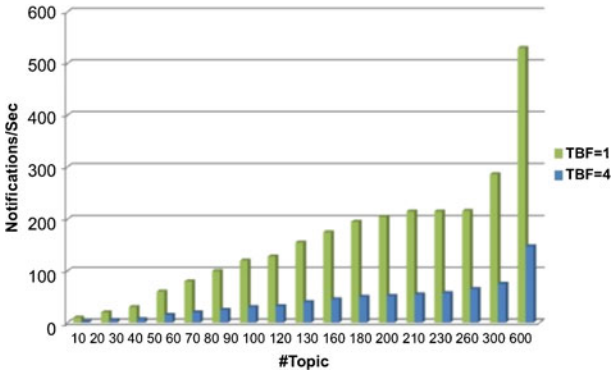

Fig. 9 Time_Based_Filter test

which a DataWriter will send data. The Time-Based Filter provides a way to set a minimum separation period, which is used to specify that a DataReader wants new messages no more often than this time period; according to the specification, if the value of this QoS policie is 0, it means that the DataReader wants all values.

Figure 9 shows the performance of dealing with notifications per second with different Topics and TIME_BASED_FILTER (TBF) settings. The result shows that DDS achieves maximal processing capacity for notifications per second when the Topic Number grows up to 600 and TIME_BASED_FILTER = 1.

Figure 10 shows the Deadline Missed Ratio as a function of the number of Topics while considering different DEADLINE periods. The result show that Deadline increases slowly when Topic Number is bigger than 160 for Deadline = 2 and 200 for Deadline = 4. These results have been obtained with a service that just runs with a topic number ranging from 10 to 500, which was observed 20 times in order to obtain the average value.

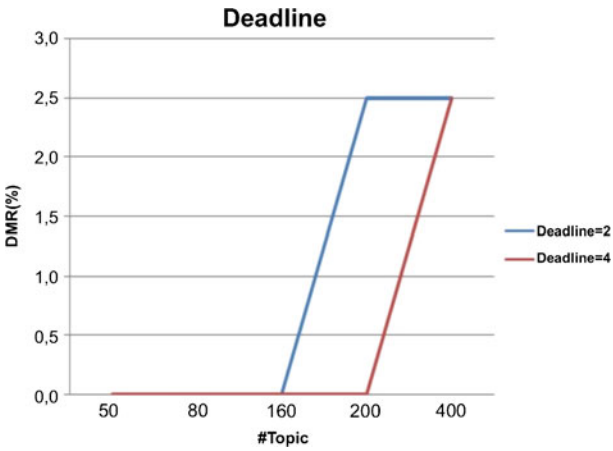

Fig. 10 Deadline test

7 Conclusions

In this paper, a new DDS compatible real-time middleware has been presented; μDDS is a publish/subscribe middleware that allows real-time wireless embedded applications to interoperate with each other, and is capable of supporting different QoS levels for various applications. The combination of a compact Java environment, the μDDS middleware and the PaRTiKle OS, has resulted in a very small footprint, low latency, and highly reliable platform for time critical Java applications. In order to validate and evaluate the performance of our implementation, several tests have been designed and performed. All the testing realized in this work, shows that the performance of this implementation is very efficient, achieving very good results in terms of throghput, latency and QoS. Evaluations results demonstrate that μDDS is lightweight and efficient, and the use of the μDDS middleware simplifies the development process of real-time wireless embedded publish/subscribe applications. In the future we will implement the proposed software architecture in other hardware architectures such as XScale and PPC, and we will make use of the TLSF memory model which is supported by the PaRTiKle operating system, this will require us to integrate the adaptation and implementation of RTSJ which we have already developed for the PaRTiKle OS.

Acknowledgement This work was developed as a part of the D2ARS Project supported by CYTED. UNESCO code 120325;330417;120314;120305.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Commun. Mag. **40**, 102–114 (2002)
2. Aquino, R., González, A., Rangel, V., García, M. Villaseñor, L.A., Edwards-Block, A.: Wireless communication protocol based on EDF for wireless body sensor networks, k. Journal of Applied Sciences and Technology **6**(2), 104–114 (2008)
3. Bonnet, P., Gehrke, J.E., Seshadri, P.: Querying the physical world. IEEE Pers. Commun. **7**(5), 10–15 (2000)
4. Boonma, P., Suzuki, J.: TinyDDS: an interoperable and configurable publish/subscribe middleware for wireless sensor networks. In: Hinze, A., Buchmann, A. (eds.) Handbook of Research on Advanced Distributed Event-based Systems. Publish/Subscribe and Message Filtering Technologies, IGI Global (2009)
5. Cerpa, A., Elson, J., Hamilton, M., Zhao, J.: Habitat monitoring: application driver for wireless communications technology. ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, Costa Rica (2002)
6. Corsaro, A., Schmidt, D.C.: The design and performace of real-time java middleware. IEEE Trans. Parallel Distrib. Syst. **14**(11), issn 1045–9219, 1155–1167 (2003)
7. Culler, D.E., Hong, W.: Wireless sensor networks introduction. Commun. ACM **47**(6), 30–33 (2004)
8. Estrin, D., Govindan, R., Heidemann, J.S., Kumar, S.: Next century challenges: scalable coordination in sensor networks. In: Mobile Computing and Networking, pp. 263–270 (1999)
9. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S.: Middleware to support sensor network applications. IEEE Netw. **18**, 6–14 (2004)
10. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. ACM SIGOPS Oper. Syst. Rev. **34**(5), 93–104 (2000)
11. Levis, P., Culler, D.: Mate: a tiny virtual machine for sensor networks. In: Proceedings of the 10th International Conference on Achitectural Support for Programming Languages and Operating Systems. San Jose, CA (2002)

12. Liu, T., Martonosi, M.: Impala: a middleware system for managing autonomic, parallel sensor systems. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego, CA (2003)
13. Mata, W., González, A., Aquino, R., Crespo, A., Ripoll, I., Capel, M.: A wireless networked embedded sistem with a new real-time Kernel PaRTiKle. Electronics, Robotics and Automotive Mechanics Conference, CERMA 2007. ISBN 0-7695-2974-7. Cuernavaca, México (2007)
14. Mata, W., González, A., Crespo, A.: A proposal for real-time middleware for wireless sensor networks. Workshop on Sensor Networks and Applications (WseNA'08). Gramado, Brasil (2008)
15. Mata, W., González, A., Fuentes, G., Fuentes, R., Crespo, A., Carr, D.: Porting jRate(RT-Java) to a POSIX real-time Linux Kernel. Tenth Real-Time Linux Workshop. Colotlán, Jalisco México (2008)
16. MiLAN Project: Available: http://www.futurehealth.rochester.edu/milan (2008)
17. OMG, Data Distribution Service for Real-Time Systems Version 1.2. OMG Technical Document (2007)
18. OMG, Model Driven Architecture (MDA), Document Number ormsc/2001-07-01. Technical report, OMG (2001)
19. OMG, Overview and guide to OMGs architecture, OMG Technical Document formal/03-06-01 (2003)
20. Pardo-Castellote, G., Farabaugh, B., Warren, R.: An Introduction to DDS and Data-centric Communications. Available: http://www.omg.org/news/whitepapers/Intro_To_DDS.pdf (2005)
21. Peiro, S., Masmano, M., Ripoll, I., Crespo, A.: PaRTiKle OS, a replacement of the core of RTLinux. In: 9th Real-Time Linux Workshop (2007)
22. Peiro, S., Masmano, M., Ripoll, I., Crespo, A.: PaRTiKle LPC, port to the LPC2000. Tehth Real-Time Linux Workshop. Colotlán, Jalisco M'exico (2008)
23. Pottie, G.J., Kaiser, W.J.: Wireless integrated networks sensors. Commun. ACM **43**(5), 52–58 (2000)
24. Souto, E., Guimaraes, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: a publish/subscribe middleware for sensor networks. Pers Ubiquit Comput **10**(1), 37–44 (2006)
25. St Ville, L., Dickman, P.: Garnet: a middleware architecture for distributing data streams originating in wireless sensor networks. In: Proceedings. 23rd International Conference on Distributed Computing Systems Workshops (2003)