A Best-Effort Approach for Run-Time Channel Prioritization in Real-Time Robotic Application

Ali Paikan, Ugo Pattacini, Daniele Domenichelli,
Marco Randazzo, Giorgio Metta and Lorenzo Natale

***Abstract*— Application domains of robotic systems are growing in complexity. It seems therefore plausible that robotic software will continue to be designed to be executed on distributed computer architectures interconnected through a network. It is a common practice today to rely on best-effort performance and assume that the latter are adequate given enough computational and networking resources. This approach however does not make best use of the available resources and, maybe more importantly, does not guarantee that performance remain constant over time. Real-time and Quality of Service become therefore important aspects in the software architecture of a robot. This article describes an approach for introducing these concepts in a publish-subscribe software middleware. The key contribution of our approach is that it leverages on the services provided by the operating system (scheduling priority and packet QoS) and abstracts them in a set of levels of priority that can be assigned dynamically, and with the granularity of individual communication channels. We implemented our approach on the YARP middleware and performed an experimental evaluation that demonstrates its benefit for increasing determinism and reducing latency in data communication. We further demonstrate this in a real-robot experiment that shows increased performance in a closed-loop scenario.**

I. Introduction

Recent advancements in different fields of robotic research are making within reach applications of growing complexity integrating force control, vision, speech and learning. It is therefore conceivable that robotic software applications will continue to require a mixture of on-board and remote computation. Many state-of-the-art humanoid robots such as iCub [1], Armar [2] and HRP [3] rely on distributed computation on a cluster of computers. A de–facto standard in terms of software architecture is the adoption of distributed frameworks (ROS [4], YARP [5], OROCOS [6] and Open-RTM [7]). This is dictated not only by computational requirements but also by established best practices in software engineering, i.e. the so-called separation of concerns [8]. In this context the publish/subscribe [9] paradigm is widely adopted thank to the levels of decoupling it offers such as anonymity, asynchronism and time decoupling. In a publish/subscribe model, a publisher (sender) registers itself into a central event service as an entity that can provide specific events (e.g., characterized by a type). In an asynchronous way, subscribers (receiver) express interests and receive one or more of the available events without any knowledge of the number and identity of publishers. Overall this allows achieving a level of decoupling that has fostered the adoption of the publish/subscribe paradigm in different domains including robotics.

A. Paikan, U. Pattacini, D. Domenichelli, Marco Randazzo, G. Metta and L. Natale are with the Istituto Italiano di Tecnologia (IIT), Genova, Italy. Emails: ali.paikan, ugo.pattacini, daniele.domenichelli, marco.randazzo, giorgio.metta, lorenzo.natale @iit.it

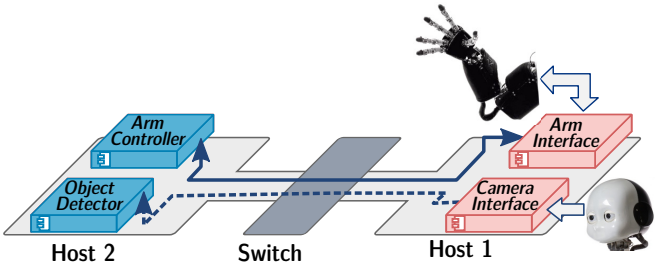

Fig. 1: An example of distributed control architecture in a network consisting of two nodes (Host1, Host2). The arrows represent communication and the data–flow between components.

Many robotic applications require real-time functionalities especially when timing constraints on task execution, data processing and synchronization are crucial. In the past, many frameworks for real-time robotics have been proposed [10]. They focus on supporting software engineering practices for code reuse, interface standardization or component-based programming. There are also some real-time frameworks (such as any-time algorithm [11] or SeART [12]) which focus on tasks reconfiguration and run-time prioritization in case the computational resources are not adequate to schedule all tasks. However, these frameworks do not address communication issues. In a distributed architecture these are non-negligible aspects: care must be taken to avoid mutual interference between components in the communication layer. In this paper we propose to deal with this problem by providing some functionalities in the middleware that allow assigning different priorities to individual communication channels (we call this approach "channel prioritization").

This approach is better explained by considering the distributed robot control example in Fig. 1. The network here is composed of two computer nodes (Host1 and Host2) which are connected using an Ethernet switch. Host1 is a machine at the robot side which provides different interfaces to access actuators and sensors. For example, *Arm Interface* is a software component which allows to remotely read encoders and to command the actuators of the arm. *Camera*

Interface captures images from the cameras and streams them out across the network. In addition, Host2 executes the *Arm Controller* component which controls the arm in close-loop, sending e.g. velocity commands to the motors. It needs to access the encoders data and to command the motors within few milliseconds by communicating with the *Arm Interface* over the network. We consider the case in which the robot is visually tracking and reaching for an object; in this case a requirement is that the *Object Detector* component receives visual feedback within a few decades of milliseconds. The *Arm Controller* component is time-critical: it requires higher controlling rate with lowest jitter during the execution of the application. To achieve this we can prioritize the communication channel between *Arm Interface* and *Arm Controller* (bold line) so that bandwidth and resources used by other channels (*Camera Interface – Object Detector*, but also other data streams from modules that are not shown here) do not interfere with the messages traveling from the first channel. In other words, messages from *Arm Controller* need to be guaranteed (in a best–effort sense) to reach *Arm Interface* with lowest latency and vice versa. Depending on the application context, priority of channels may need to be dynamically changed at run–time as demonstrated in [12].

This paper presents an approach for run-time prioritization of communication channels in robotic applications. The approach focuses on publish/subscribe architecture and leverages the operating system functionalities to prioritize specific communication channels between publishers and subscribers. We extend the properties of individual connection channels with a priority level which affects the priority of the threads that handle the communication and the network packet's type of service. We implemented our approach on the YARP [5] middleware. We performed experimental analysis that demonstrate a significant improvement in the communication performance and in the performance of a controller in a closed-loop scenario. The approach does not require specific components for message prioritization and it does not add any overhead to the communication. In addition and, more importantly, it allows for remote configuration of Quality of Service (QoS) and for run-time, dynamic prioritization of communication channels. The rest of the paper is organized as follows. Section II outlines the related work. Section III describes the common problems of channel prioritization and explains how it can be integrated in a publish/subscribe middleware. The actual implementation of channel prioritization in the YARP middleware and its experimental evaluation are presented in Section IV. Finally, Section V presents the conclusions and future work.

II. Overview and Related works

The problem of message prioritization in real-time communication is not new and has received a lot of attention in the field of real-time networking, sensor networks, high-performance computing and robotics. Depending on how the real-time communication is implemented, the frameworks for message prioritization can be categorized in different classes. Some frameworks use customized protocols or specific network for real-time communication. These frameworks usually replace, either partially or entirely, the standard protocol stack of the operating system with their own implementation for packet prioritization and scheduling. For example, RTnet Stack [13] modifies the standard Linux IP stack to provide a framework for exchanging data under hard real-time constraints. Within this category, many other protocol and frameworks can be found in the fields of sensor networks and Fieldbus such as RAP [14] and EtherCAT [15].

Other frameworks are based on an unreliable (i.e. standard Ethernet) network links and extend them with different functionalities for Quality of Service (QoS) to achieve deterministic message exchange. Real-time CORBA [16] model is an extension of the CORBA Event Service to support low latency, periodic rate-based event processing, efficient event filtering and correlation. TAO is an implementation of real-time CORBA which uses Event Channels to dispatch events to consumers on behalf of suppliers. In TAO, message prioritization can be achieved by configuring the real-time event channels with multiple scheduling policies (e.g. maximum urgency first). Similarly, channels can be built with varying levels of support for preemption. This flexibility allows applications to request allocation of resources for different application requirements.

Various architectures aims at providing publish/subscribe in QoS-enabled component middlewares such as RTSE [16] and RTNS[17]. Deng et al. [18] provided a comprehensive survey of these architectures and described different design choices for implementing real-time publish/subscribe services. The actual implementation, however, varies on the point in the middleware in which a service is integrated. For example, for CORBA [19], the different choices can be the component itself, the container or the component server. A well-known example of this type of frameworks is the OMG Data Distribution Service (DDS) [20], which represents a standard for QoS-enabled publish/subscribe communication for mission-critical distributed systems. DDS is designed to achieve location independence, scalability and platform portability. Multiple implementations of DDS are available, either commercially (RTI Connext DDS [21]) or from open-source projects OpenSplice [22]).

In robotics, Kuo et al. [23] proposed a distributed real-time software framework based on CORBA. It provides facilities such as priority-banded connection to integrate and simplify the real-time CORBA API. Martinez et al. [24] described the adaptation of some DDS implementations in a robotic middleware based on ICE (RoboComp) and compares it with previous implementation demonstrating noticeable improvements in terms of throughput, latency and jitter.

The approach proposed in this paper falls in the second category of the work described above, as it does not need customized protocols nor specific communication link/hardware: it relies only on standard Ethernet network and exploits the operating system functionalities for scheduling and prioritizing threads and data packets. Our approach aggregates QoS parameters and thread scheduling priorities providing a simple abstraction consisting in

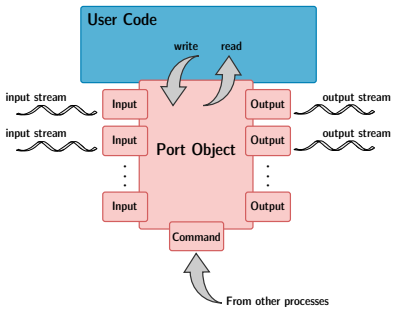

Fig. 2: The internal structure of a YARP port object.

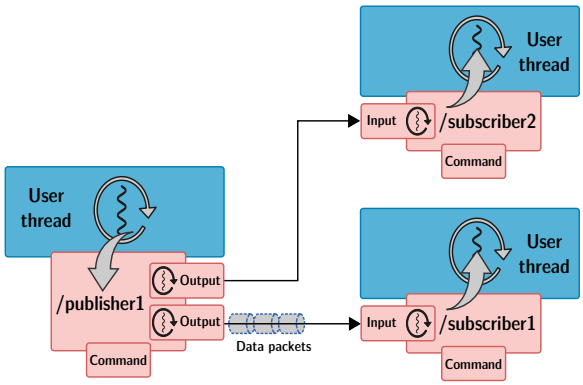

Fig. 3: An example of components asynchronous communication in YARP. A publisher is pushing messages to two different subscribers using separate dedicated threads.

different priority levels. Such priority levels are not assigned to topics/data sources but to individual connections. This means that the same data source can have different priority levels, from the highest value when it needs to be delivered to time-critical components (e.g. closed-loop controller) to the lowest when requested for unimportant tasks (e.g. visualization). Finally, priority levels can be assigned dynamically and tuned depending on the context and application. At the implementation level, our solution is completely distributed (i.e. it does not rely on a centralized broker) and it directly interfaces with the sockets of the operating systems and thus does not require any additional dependency.

III. Communication Channel Prioritization

A way to implement a publish/subscribe system is by using an intermediate broker to which publishers post messages. The broker then performs a store-and-forward function to deliver messages to subscribers that are registered with it. To implement message prioritization in such scenario the broker can simply route messages with a desired order. For example a subscriber to a specific topic may need to receive a copy of a message before another subscriber. The centralized approach, however, becomes easily inefficient and does not scale well. A more efficient approach is to let components share meta–data (describing for example the type of messages) and establish peer-to-peer connections between publishers and subscribers. For example, Data Distribution Service (DDS) uses IP multicast, YARP and ROS use a centralized name server for storing meta–data and perform naming lookup (the approach implemented in YARP will be described in the following section). Achieving message prioritization in such distributed systems is more difficult because there is no central authority that can control message delivery. In the following sections we use YARP to describe how channel prioritization can be implemented in a publish/subscribe framework without any centralized broker.

A. Overview of YARP

YARP is a multi–platform distributed robotic middleware which consists of a set of libraries, communication protocols, and tools to keep software modules and hardware devices cleanly decoupled. Communication uses special objects called “port”. Using ports publishers can send data to any number of receivers (subscribers), either within the same process, crossing the boundaries of processes and, using network protocols, even across machines. YARP manages connections in a way that decouples publishers and subscribers. A port is an active object that can manage multiple connections either as input or output (see Fig. 2). Two ports can be connected after naming lookup on a central server and exchange data in a distributed, independent way. It is worth noting that each connection has a state that can be manipulated by external (administrative) commands, which in turn manage the connection and/or obtain state information from it. Ports can be connected using standard protocols (e.g. TCP, UDP, MCAST) either programmatically or at runtime using administrative commands. A single port may transmit the same message across several connections using different protocols including custom “carrier” objects that implement new protocols [25], [26].

B. Integration in YARP

Asynchronous communication in YARP can be achieved in different ways. One way to do this in a protocol–independent manner is to configure the port object to send and receive user data in separate threads. A conceptual example is depicted in Fig. 3 in which an asynchronous publisher (Publisher 1) pushes data to two different subscribers using a separate dedicated thread for each communication channel. This decouples timing between a publisher and its subscribers and reduces the amount of time spent in the user thread for sending data. Inside the subscribers a dedicated thread reads data from a communication channel.

Generally speaking, using dedicated threads for communication introduces extra computational time to the component execution due to thread scheduling and context–switching overhead. However, dedicated threads can provide a better implementation abstraction and potentially can be exploited for the implementation of prioritized communication channels. Real–time properties and QoS attributes can be configured at different scopes, i.e., user thread, dedicated communication thread and data packets. Configuring real–time properties such as priority

or scheduling policy of the user thread can be done either programmatically from the user code or automatically using component middleware functionalities and dedicated tools [12] (although this is beyond the scope of this paper).

In our approach real–time properties can be configured separately for each communication thread. In other words, we modified YARP adding the possibility to change the priority of the thread that handles data transmission over a communication channel. When a publisher writes data to a port, it passes it to the corresponding thread. At this point, the operating system takes over being in charge of scheduling the threads with respect to their real–time properties so that highest–priority threads can write data to the socket before the others.

As shown in Fig. 2, a port object can also subscribe to multiple publishers using separate input channels. In such a case, thread prioritization can be also applied at the subscriber side to ensure that messages in a specific channel will be delivered to the user with high priority (i.e. minimum jitter).

C. Data packets prioritization

Configuring the real–time priorities of communication threads guarantees that messages are written or read from channels with specific priorities. Normally, when messages are written to a socket they are handed over the operating system and there is no control on the order in which they are actually transmitted to the transport layer. Generally speaking, there is no silver bullet to data packet prioritization in computer networks. Some partial solutions exist and are highly dependent on the network topology, infrastructure and communication protocol. However in Ethernet local area networks (LAN) data packets can have configurable properties that specify priority of delivery with respect to time and order.

To clarify the issue, we consider the network architecture from Fig. 1. There are two places in our network in which packet traffic congestions can potentially happen: *i)* in the OS level (i.e. inside the network driver) when outbound data from multiple applications are written to the network interface controller of the Host1, *ii)* in the switch, when packets from different ports (Host1 and 2) are forwarded to a single port (Host3). These are common bottlenecks in computer networking that become particularly critical when the infrastructure does not have enough resources for routing all traffic.

A driver queue bridges the IP stack and the network interface controller (NIC). In some operating systems (e.g., Linux) there is an intermediate layer between the IP stack and the driver queue which implements different queuing policies. This layer is responsible for diverse traffic management capabilities including traffic prioritization. For example in Linux distributions, the default queuing policy (i.e., pfifo_fast QDisc) [27] implements a simple three band prioritization scheme based on the IP packet's TOS [28] bits. Within each band, packets follow a FIFO policy. However, prioritization happens across bands: as long as there are packets waiting in higher–priority band, the lower bands will not be processed. To hand the user–level messages to the network controller in a prioritized manner, it is enough to provide a mechanism so that the TOS bits of the data packets can be adjusted according to the desired priority of the channels.

Multilayer network switches (i.e. operating on OSI layer 3 or 4) are capable of implementing different QoS such as packet prioritization, classification and output queue congestion management. They commonly use Differentiated Services Code Point [29] (DSCP) which is the six most significant bits in the TOS byte to indicate the priority of an IP (V4 and V6) Packet. Differentiated services enable different classes of prioritization which can be used to provide low latency to critical network traffic while providing simple best–effort service to non–critical applications. To guarantee low–latency packet transfer from a publisher to subscriber, the TOS bits can be adjusted properly to fall into the highest–priority band of queuing policy and to form a high–priority class of differentiated service in network switches.

IV. Implementation and results

As described earlier port administrative commands provide a rich set of functionalities to monitor and change the state of a port and its connections. To implement the channel prioritization in YARP, these functionalities were extended to allow tuning QoS and real–time properties of port objects with the granularity of individual connections. In the current implementation, the port administrator provides two set of commands that affect the priority of a communication channel: setting the scheduling policy/priority of a communication thread and configuring the TOS/DSCP bits for the data packets it delivers. For example, we can simply configure real–time properties of the output entity of `/publisher1` from Fig. 3 using the YARP tools as follows:

```
$ yarp admin rpc /publisher1
>> prop set /subscriber1 (sched
                          ((policy SCHED_FIFO)
                           (priority 30)))
```

The first line "`yarp admin rpc`" simply opens an administrative session with the port object of `/publisher1`. The second line is the real command to the administrative port. It adjusts the scheduling policy and priority of the thread in `/publisher1` which handles the connection to `/subscriber1` respectively to SCHED_FIFO and 30 on Linux machines [1].

For packet priorities we have chosen four predefined classes of DSCP. These classes are selected so that packets can be treated similarly by the OS queuing policy (if available) and in the network switch. For example a packet with priority class `Low` will be in the lowest priority band (Band 2) of the Linux queuing policy and will have the lowest priority in the network switch. Table I provides a list of these classes.

[1]The thread scheduling properties and policies are highly OS dependent and a proper combination of priority and policy should be used.

TABLE I: Predefined classes of packet priority

Class	DSCP	QDisc
Low	AF11	Band 2
Normal	Default	Band 1
High	AF42	Band 0
Critical	VA	Band 0

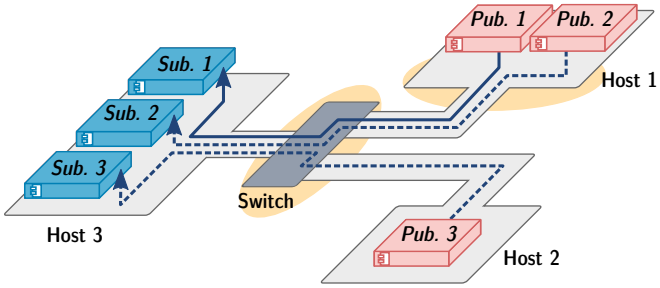

Fig. 4: An example of publish/subscribe architecture in a LAN network consisting of three nodes (Host1, Host2 and Host3) connected through an Ethernet switch. The arrows represent data–flow from publishers to subscribers.

Analogously, data packet priority can be configured via administrative commands by setting one of the predefined priority class (or by directly configuring the DSCP/TOS bits):

```
$ yarp admin rpc /publisher1
>> prop set /subscriber1 (qos ((priority HIGH)))
```

This simply sets the outbound packets priority to HIGH for the connection from `/publisher1` to `/subscriber1`.

These two set of parameters can be set for every channel in the same way and jointly define the actual priority of a communication channel in our publish/subscribe architecture.

A. *Evaluation of message round-trip time*

To evaluate the effect of channel prioritization on the messages round-trip time, we have devised two different test cases. The first case deals with evaluating channel prioritization when traffic congestion happens at the network card (OS level) while the second one investigates performance improvement due to channel prioritization at the network switch.

Figure 4 demonstrates the network architecture for both test cases. The nodes (Host 1 to 3) are Linux machines with PREEMPT-RT kernel which are connected using Gigabit Ethernet and a QoS–enabled switch (CISCO Catalyst 2960). In each test case, only two separate channels between the publishers and the subscribers are used. We measure the round–trip time of messages in the first channel (from Pub.1 to Sub.1). This is done via acknowledgment packets from Sub.1 to Pub.1 for each messages received by Pub.1. The second channel (from Pub.2 to Sub.2 for the first case and from Pub.3 to Sub.3 for the second case) produces arbitrary but controllable network load. These channels are shown as dashed-lines in Fig. 4.

For each test case, two different set of experiments have been performed to measure packet trip time with and without

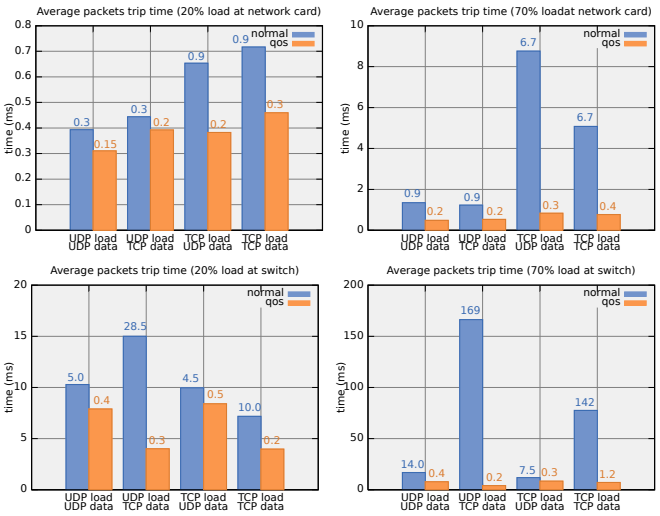

Fig. 5: Evaluation results of channel prioritization at the network card and switch for different loads and transport protocols. The bars labeled as "qos" represent the test results in presence of channel prioritization. The bars labeled as "normal" represent the test results in absence of any prioritization. The number above the bars represent the standard deviations of samples.

channel prioritization. To achieve the highest priority for the channel from Pub.1 to Sub.1, the scheduling policy and priority of communication threads are respectively set to SCHED_FIFO and 30. The thread priority is chosen so that it is higher than the other processes during the experiment but lower than network interrupt priorities. The packet priority for the corresponding channel is also set to HIGH (AF42/Band 0). The tests are repeated with two different network loads (corresponding respectively to 20% and 70% of the maximum bandwidth) generated by Pub.2 and Sub.2. Moreover, to see the effect of the underlying communication protocol on channel prioritization we have repeated the tests with different combination of TCP and UDP both for the load and the measurement channels.

Fig. 5 illustrates the measured average and standard deviation of the packets trip time with (bars labeled as "qos") and without channel prioritization (bars labeled as "normal") for different protocols and network loads. The number above the bars represent the standard deviations of samples.

Top plots from Fig. 5 demonstrate the comparison when the two publishers (Pub.1 and Pub.2) are on the same machine and produce outbound traffic congestion at the network card only. In general, as it can be seen in the plots, channel prioritization produces slightly lower (in average) and more deterministic (smaller standard deviation) packet trip time. This effect is more remarkable when the network is loaded at the 70% of the maximum bandwidth. Notice that in this case, Pub.1 and Pub.2 are located on the same machine (see Fig. 4) and the network load is generated by Pub.2. In this case traffic congestion happens at the level of the network card driver with consequent higher latency in packet

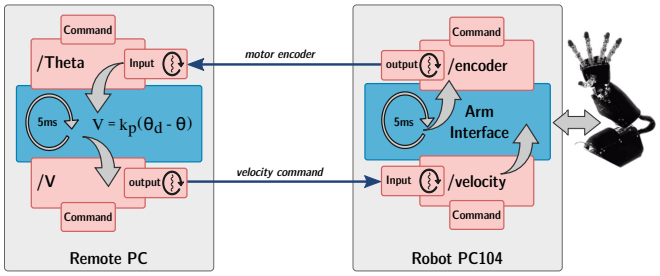

Fig. 6: The velocity control loop.

delivery time. By prioritizing the measurement channel (bold arrow in Fig. 4), the communication thread in Pub.1 receives higher priority by the operating system. Moreover, since packets from Pub.1 are prioritized (AF42/Band 0), they get highest priority also in the network queue and are pushed to the network physical layer before the packets from Pub.2.

Bottom plots from Fig. 5 demonstrate the results of the second test case when publishers (Pub.1 and Pub.3) from separate machines are used for the experiment and traffic congestion occurs at the network switch. The only difference in this case is that Pub.1 pushes larger packets to Sub.1. The reason for this is that larger data packets create higher traffic congestion in the switch. This explains why the packet trip times measured in this experiment are slightly higher than in the previous case. However, as it can be seen in Fig. 5, channel prioritization greatly improved the performance (resulting in lower latency) especially when the network is highly loaded (70%). It can also be observed a big difference in packet trip times when different communication protocols are used. The reason is that the TCP communication protocol uses the bandwidth in a smart way to achieve lower latency. Moreover, QoS–enabled network switches also have different routing policies for different packet sizes. However, as it can be seen from the results, messages transmitted through prioritized channels are comparatively less affected by different transport protocols.

B. A velocity control experiment

To further validate the advantages of channel prioritization, we have carried out an experiment involving the control of a joint of a real robot. At this aim we have used the iCub robot and its distributed system. As shown in Fig. 6, two tasks running on two different PCs. They are employed in a classical closed-loop scheme whose aim is to attain a desired joint angular position by sending proper velocity commands to the corresponding motor. The *Arm Interface* on the Robot PC104 provides a YARP port (`/encoder`) that publishes robot motor encoders data every 5*ms* and another port (`/velocity`) that receives velocity commands. A controller task on the *Remote PC* reads the encoder values (every 5*ms*) from its input port (`/Theta`) which is connected to the `/encoder` port. It then calculates the velocity command and sends it to *Arm Interface* through the channel connecting `/V` to `/velocity`.

Similarly to the previous section, we prioritized the two

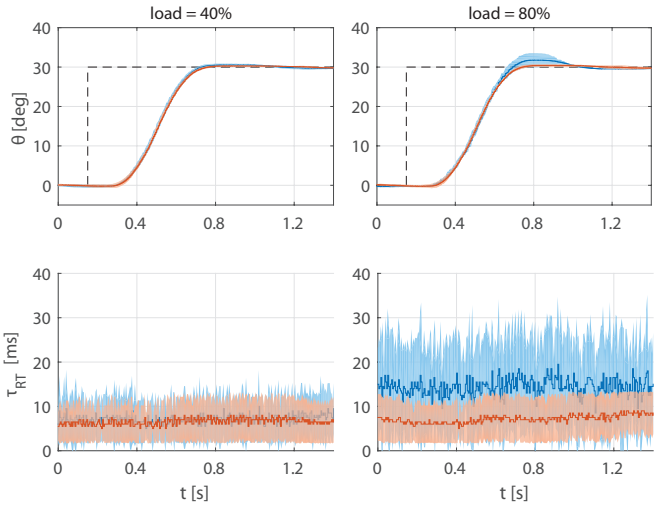

Fig. 7: Evaluation of the joint velocity control experiment. Both system step responses θ and round-trip time τ_{RT} are depicted in the case of 40% (left) and 80% (right) network loads. The responses corresponding to the use of QoS are in red, whereas the system responses without channel prioritization are in blue. Results are reported in terms of average and confidence interval of 95%.

communication channels in the control architecture. We also used some arbitrary channels to create two different network loads (i.e. 40% and 80%) during the experiment. We recorded the joint step response θ of the system under the disturbance of such network loads and compared the effects in the overshoots and round-trip time τ_{RT}[2] with and without QoS (channel prioritization).

Fig. 7 depicts the average and the 95% confidence interval computed over 10 successive trials. It clearly illustrates how the system behaved robustly in the case of 40% load, that is the performances were almost equivalent irrespective of the use of our QoS. Conversely, when the network load was increased up to 80%, the QoS was remarkably able to keep the step response unchanged. Importantly, in this latter case, the system without channel prioritization overshoots the target (its response is under-damped). This significant performance variation was caused by network collisions that produce an increased delay in the closed-loop. This is well visible in Fig. 7, i.e. when QoS is not enabled the round-trip time is approximately twice as high (see also Tab II). This experiment demonstrates that channel prioritization is a key factor that prevents external disturbances from negatively affecting the control loop delay, thus preserving the original stability margins of the system.

[2]Round-trip time in this case is computed in *Remote PC* by embedding a time-stamp in the velocity commands. This time-stamp is received by *Robot PC104* and propagated in the subsequent encoder message back to *RemotePC* which computes τ_{RT}. In this experiment, τ_{RT} is affected by the rates of the controller and *Arm Interface* threads (both running with period $T_s = 5ms$). Therefore, it holds $\tau_{RT} = n \cdot T_s$, where the positive integer n might vary instantaneously depending on the channel's conditions.

TABLE II: Results of the joint velocity control experiment. Maximum values of the overshoot are reported as well as the average of the round-trip time τ_{RT} along with the maximum of the 95% confidence interval (in parentheses).

	Overshoot [%]		τ_{RT} [ms]	
Network Loads	40%	80%	40%	80%
Normal	3	11.8	7.14 (18.25)	14.37 (35.12)
QoS	1.3	2.5	6.57 (13.16)	7.15 (13.33)

V. Conclusions

This article described an approach to integrate channel prioritization in a publish/subscribe middleware. In our approach specific communication channels can be prioritized to ensure, in a best–effort sense, the minimum message delivery time from publishers to subscribers.

The approach simply leverages the operating system functionalities such as real–time thread scheduling and IP packet type of service bits to deal with the typical bottlenecks that cause network congestions in local network. In addition, our approach does not require centralized broker for message prioritization and for this reason it can be applied to peer-to-peer publish-subscribe architectures. Finally (although not investigated in this paper), it allows for remote and dynamic configuration of the parameters that control the communication priorities.

We implemented our approach in the YARP middleware and evaluated it in different scenarios demonstrating significant improvement in jitter and latency of message delivery, especially in presence of heavily loaded network. Moreover, using a classical closed-loop control experiment we demonstrated that channel prioritization guarantees that stability margins remain unvaried and prevent increase of the delays in the control loop under heavy network loads. These results make our approach particularly useful in distributed time–critical applications. Future work will investigate mechanisms for monitoring communication channels for automatically selecting optimal prioritization policies.

Acknowledgment

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement No. 270273 (Xperience), project No. 611832 (WALK-MAN) and project No. 612139 (WYSIWYD).

References

[1] G. Metta, G. Sandini, and D. Vernon, "The iCub humanoid robot: an open platform for research in embodied cognition," *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pp. 50–56, 2008.

[2] T. Asfour, K. Regenstein, P. Azad, J. Schröder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, "ARMAR-III: An integrated humanoid platform for sensory-motor control," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, 2006, pp. 169–175.

[3] K. Kaneko, K. Harada, F. Kanehiro, G. Miyamori, and K. Akachi, "Humanoid robot hrp-3," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008, pp. 2471–2478.

[4] 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*, 2009.

[5] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29–45, Jan. 2008.

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

[7] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: OpenRTM-Aist," *Simulation, Modeling, and Programming for Autonomous Robots*, pp. 87–98, 2008.

[8] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 1, pp. 100–112, 2010.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, June 2003.

[10] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[11] T. L. Dean and M. S. Boddy, "An analysis of time-dependent planning." in *AAAI*, vol. 88, 1988, pp. 49–54.

[12] F. Mastrogiovanni, A. Paikan, and A. Sgorbissa, "Semantic-aware real-time scheduling in robotics," *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 118–135, 2013.

[13] J. Kiszka and B. Wagner, "RTnet - a flexible hard real-time networking framework," *IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1, 2005.

[14] B. Blum, T. Abdelzaher, and J. Stankovic, "RAP: a real-time communication architecture for large-scale wireless sensor networks," *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 55–66, 2002.

[15] M. Felser, "Real-time ethernet–industry prospective," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1118–1129, 2005.

[16] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba event service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.

[17] P. Gore, I. Pyarali, C. D. Gill, and D. C. Schmidt, "The design and performance of a real-time notification service," in *Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2004, pp. 112–120.

[18] G. Deng, M. Xiong, J. Balasubramanian, and G. Edwards, "Evaluating real-time publish/subscribe service integration approaches in QoS-enabled component middleware," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2007, pp. 222–227.

[19] OMG, "Common Object Request Broker Architecture (CORBA/IIOP).v3.1," OMG, Tech. Rep., Jan. 2008.

[20] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops*. IEEE, 2003, pp. 200–206.

[21] R. Connext, "RTI Connext DDS Professional." [Online]. Available: http://www.rti.com/products/dds/

[22] PrismTech, "OpenSplice Data Distribution Service." [Online]. Available: http://www.prismtech.com/vortex/vortex-opensplice

[23] Y.-h. Kuo and B. Macdonald, "A Distributed Real-time Software Framework for Robotic Applications," in *International Conference on Robotics and Automation*, no. April, 2005, pp. 64–69.

[24] J. Martinez, A. Romero-Garcés, L. Manso, and P. Bustos, "Improving a Robotics Framework with Real-Time and High-Performance Features," in *Simulation, Modeling, and Programming for Autonomous Robots (Simpar)*. Springer-Verlag New York Inc, 2010, p. 263.

[25] A. Paikan, P. Fitzpatrick, G. Metta, and L. Natale, "Data Flow Port Monitoring and Arbitration," *Software Engineering for Robotics*, vol. 5, no. 1, pp. 80–88, 2014.

[26] A. Paikan, V. Tikhanoff, G. Metta, and L. Natale, "Enhancing software module reusability using port plug–ins: an experiment with the iCub robot," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.

[27] W. Almesberger, J. H. Salim, and A. Kuznetsov, "Differentiated services on linux," in *Globecom99*, no. LCA-CONF-1999-019, 1999, pp. 831–836.

[28] P. Almquist, "Type of service in the internet protocol suite," 1992.

[29] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers," 1998.