Data Distribution Service (DDS): A Performance Comparison of OpenSplice and RTI Implementations

Paolo Bellavista, Antonio Corradi, Luca Foschini, Alessandro Pernafini
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
University of Bologna, Italy
{paolo.bellavista, antonio.corradi, luca.foschini}@unibo.it; alessandro.pernafini@studio.unibo.it

Abstract— **Data distributions systems with guaranteed Quality of Service (QoS) levels, such as the data-centric Data Distribution Service (DDS) standard specification, have gained more and more success in the last decade. These systems represent suitable solutions for effective and high-performance data communication for challenging application scenarios with real-time requirements, such as air traffic management, industrial automation, smart grids, and, more recently, financial applications. Notwithstanding the last decade has witnessed the diffusion and consolidation of some major implementations, only a very few, in some sense obsolete, performance analysis studies are available in the literature. To fill that gap and to facilitate future IT decision processes, we propose a thorough analysis of the DDS implementations proposed by the two main stakeholders in the DDS market, namely, PrismTech and Real-Time Innovations (RTI). The reported experimental results point out the pros and cons of both solutions in terms of data delivery performance, also by precisely evaluating bottlenecks and overhead, for instance in terms of CPU and memory resource usage.**

Keywords: Data Distribution Service (DDS), Performance Evaluation, OpenSplice, RTI

I. INTRODUCTION

A growing number of mission-critical systems, spanning from air traffic control and embedded systems to automated stock trading systems, require QoS-enabled data-centric middleware platforms able not only to provide distributed applications with the needed information (often high-volume data) but also to get the whole information content in a timely manner (often with high data-rate). To overcome those limitations in 2004, almost one decade ago, the Object Management Group (OMG) has introduced the Data Distribution Service (DDS) standard specification [1]. DDS adopts the publish/subscribe model (pub/sub) and implements a backbone for QoS-enabled data dissemination in a timely and dependable manner. DDS obtains interoperability with guaranteed QoS via standard i) language-independent interfaces and ii) transport protocols that allow DDS-based applications to dynamically interconnect, to publish/subscribe information of interest, and to define QoS policies aimed to negotiate QoS levels for information delivery, received, and locally processed.

Notwithstanding the central position and recognized acceptance of DDS by the industry of mission critical services, only a few works in literature studied and compared the performance of existing DDS supports; in addition, most of them refer to the early stages of the DDS standard specification and focus on seminal implementation efforts [2, 3, 4, 5]. Now that standards and implementations have consolidated, we claim it is the right time to present a thorough quantitative performance analysis of the two main DDS products in the field, namely, OpenSplice DDS by PrismTech (i.e., OpenSplice) and Connext DDS by Real-Time Innovations (i.e., RTI) [6, 7].

The paper adopts a practical approach and considers several different performance metrics useful to benchmark the two implementations, first applied to a more limited *lab environment*, and then to a large-scale *production environment*. We truly believe that the paper can provide a significant technical contribution to help IT managers in making informed decisions on implementation selection/adoption because our analysis pinpoints both the advantages and limitations of OpenSplice and RTI, by evaluating also their main bottlenecks and related overhead when using DDS standard functions. Let us anticipate that the main result of our study is that, although both implementations perform very well under heavy load conditions, OpenSplice suits better the delivery of data of limited size, while RTI has been optimized for large messages and can achieve higher throughputs.

The paper is structured as follows. Section 2 gives the necessary background and definitions about DDS in general. Section 3 presents and qualitatively analyzes OpenSplice and RTI implementations. Section 4 shows the extensive experimental results, collected, respectively in the lab and production environments. Section 5 overviews related work in the field. Finally, conclusive remarks end the paper.

II. DATA DISTRIBUTION SERVICE BACKGROUND

This section briefly introduces the DDS architecture and some definitions/terms useful for the analysis of the considered DDS implementations.

A. The DDS Standard

The OMG DDS standard adopts the publish/subscribe distribution paradigm and the data-centric approach defined in [8]; the underlying data distribution model uses a Global Data Space (GDS) to identify data circulating in the system. GDS provides also built-in data isolation mechanisms, called domains and partitions, to improve system scalability: middleware nodes can be organized into physical (domains) and logical (partitions) groups and hierarchies that promote multiple views of the same GDS and physical DDS deployment.

978-1-4799-3755-4/13/$31.00 ©2013 IEEE

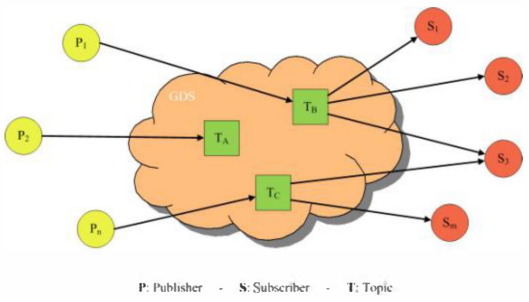

Figure 1. DDS distributed architecture.

Moreover, the DDS specification defines high-level standardized interfaces and functions dividing them into two layers: Data Local Reconstruction Layer (DLRL) and Data-Centric Publish/Subscribe (DCPS), in order to differentiate and isolate application problems from low-level implementation details. DLRL is an optional high-level Application Programming Interface (API) to provide an object-oriented view of data exchanged by the (lower) DCPS layer. DCPS is a mandatory low-level layer that enables efficient delivery of data injected by publishers to subscribers; in this paper, we will focus on DCPS only.

The producers and consumers of data, called Publishers and Subscribers, automatically discover and match each other when they have a compatible Topic (see Fig. 1) [1]. Publishers (see Fig. 1) send their data into the common GDS, and the DDS support propagates the information to all interested Subscribers, that make data locally available to applications. The underlying DDS data distribution overlay is completely decentralized and adopts a peer-to-peer model that does not require any centralized broker, thus granting high reliability and robustness because any failure of participating nodes does not compromise the whole system, but only the local participants of the crashed locality.

While DCPS describes distributed DDS components, a separate standard, the Real-time Publish-Subscribe (RTPS) DDS Interoperability Wire Protocol, specifies the network protocols used to dynamically discover DDS entities and to exchange data between publishers and subscribers [9]. In particular, RTPS defines all low-level mechanisms to optimize communication aspects, such as caching. For instance, for each Topic data sample pushed by the Publishers in the GDS, the middleware delivers it to all subscribed Subscribers that maintain a copy of it in their local caches. In other words, DDS supports push interactions only, mainly to save network resources, but also to maintain at each Subscriber a local database with all the latest values of each Topic, so to have the possibility to locally perform query operations without introducing any additional network traffic. Let us note that, especially for local deployments, some DCPS-compliant DDS solutions, such as OpenSplice, may employ proprietary data delivery protocols.

Additional details about DCPS and RTPS can be found in the OMG specification [1, 9].

B. *Design Guidelines and Implementation Directions*

As for all standards, DDS specifications define component interfaces, policies (related to QoS), and protocols (for data exchange and interoperability). At the same time, the standard intentionally leaves DDS provider companies the freedom to adopt different implementation choices that, of course, affect the performance of the final products brought to the market.

In the following, we introduce some main possible design guidelines and implementation directions. Let us note that our analysis focuses on the characteristics of DDS implementations in fully-controlled Local-Area Network (LAN) environments because most products, including the two considered market leaders, have consolidated mainly for this deployment setting. At the same time, especially in the academia, some solutions are starting to employ optimized and proprietary pluggable transport protocols for Wide-Area Network (WAN) deployments where multicast is usually not feasible; anyway, these efforts are out of the scope of this paper [10, 11].

A first important design guideline and implementation direction is given by the adopted *communication* model. In particular, the DDS specification allows DCPS implementations and applications to take advantage of various transports, including *unicast*, *multicast*, and *broadcast* transports.

A second direction relates to the *distributed architecture* model adopted for the participant discovery and data distribution phases. Starting from the discovery phase, the *centralized* model is the simplest one, relying on a central component that acts as repository for DCPS information in order to store the data needed to create/manage connections between DDS participants and to enable their mutual discovery and event flows. The centralized model has been adopted by seminal implementations, such as Object Computing inc. (OCI) OpenDDS [12], that uses a centralized CORBA-based component acting as a repository for DCPS meta information; in that implementation, data sample distribution anyway occurred via peer-to-peer exchanges. More recent implementations, including OpenSplice and RTI, tend to use *peer-to-peer* communications also for the discovery phase. Discovery of both participants and matching publications-subscriptions is performed through a pair of sub-protocols, respectively called Simple Participant Discovery Protocol (SPDP) and Simple Endpoint Discovery Protocol (SEDP), that leverage publications and subscriptions on some built-in DDS topic [9].

By focusing on data distribution, instead, the two main distributed architecture models, both peer-to-peer, are *decentralized* and *federated*. The *Decentralized* model places both the specific application logic and the DDS-based communication/configuration capabilities into the same user process, although application and DCPS support (handling DDS communications via either RTPS-based or proprietary protocols) typically run in separate threads. That allows

avoiding application and DCPS support in a separate daemon process by making the DDS-based application self-contained. The *Federated* architecture, on the contrary, requires a separate DCPS support daemon process for each network interface, to be available and started on each node before participants can communicate. That allows to establish, based on reliability requirements (e.g., reliable or best-effort) and transport addresses (e.g., unicast or multicast), overlays of DCPS daemons and optimized communication channels between different participant nodes.

Finally, a third direction is associated with a possible *data exchange optimization* model to be applied over traversing data samples. The main optimization mechanisms adopted in DDS products for LAN deployment settings is *data batching*, which tends to buffer and concentrate in the same datagram multiple data samples to be sent to participants on different nodes.

III. OpenSplice and RTI at a Glance

This section presents a brief comparison of OpenSplice and RTI implementations for a better understanding of the quantitative performance results reported in the next section.

Starting from OpenSplice and focusing first on the communication model, although in its first versions this product supported broadcast and multicast transports only [4], more recent versions support unicast as well, in particular to effectively target WAN interconnections. As regards the adopted distributed architecture, OpenSplice implements a federated model. On the one hand, from the application development perspective, this allows a clear separation of the applications, that run in a separate user process, from the DCPS configuration and communication-related details. On the other hand, from the communication point of view, the federated architecture fosters the de-/multiplexing of data coming/going from/to different DDS participants deployed on the same node; in other words, it intrinsically enables the bundling and data batching optimization that OpenSplice uses by default. At the same time, as a possible side effect, the DCPS daemon is a potential communication bottleneck at very high data rates and a single point of failure within each physical host. To overcome such limitations, OpenSplice supports also the decentralized model, although it is less optimized and not considered the default setting. Moreover, the OpenSplice DCPS implementation supports both an optimized proprietary data delivery protocol, called RTNetworking [13], suggested as the default one for local LAN deployments, and the standard RTPS protocol for interoperability with other DDS solutions.

Passing to RTI, and starting from the communication model, similarly to OpenSplice, RTI supports both unicast and multicast, while it does not support broadcast. About the distributed architectural model, we claim that this aspect is particularly relevant and represents the main difference between the two products. RTI adopts a decentralized architecture where applications are self-contained (no need for a separate daemon); therefore, latency is usually reduced: the performance bottleneck issue often associated with the DCPS/RTPS communication daemon is solved, and there is no single failure point. The main drawback is that decentralized implementations make it more difficult to employ data batching optimizations between multiple DDS applications executing on a single node (and this is the main motivation why RTI does not support that by default). The consequence is a reduction of possible scalability benefits, especially for exchanges of short data chunks. Let us note that, differently from OpenSplice, RTI employs as default, and as the only supported data delivery protocol, the standard RTPS protocol.

Let us conclude the section by summarizing the characteristics and by proposing a synoptic view of the two DDS implementations at a glance. First of all, both systems support all communication models. As for participant discovery, they both adopt a peer-to-peer approach. The main differences, instead, lie in the distributed architecture adopted for data exchange: OpenSplice adopts a federated model and supports data batching by default; RTI uses a decentralized model and does not support batching by default (although a data batching extension for data coming from the same application may be set [14]). Let us anticipate that those differences relevantly influence the performance of the two solutions, especially for high data rates and when the DDS support is used close to situations of resource saturation. As a general consideration, useful also for other possible future DDS implementations, federated models more easily enable local optimizations; for instance, they can boost system performance for exchange of small data. However, in some cases, the additional overhead (e.g., needed to execute local data batching operations) may overcome the benefits. Centralized models, on the contrary, tend to perform worse when far from saturation, but are more performing for high data rates and large-size messages, and, in general, present a more predictable performance trend.

IV. Experimental Results

This section presents the extensive set of experimental performance results collected over our two main testbeds, called respectively lab and production environments in the following. First we report some details about the employed methodology, performance metrics, and measurement tools; then, we report performance results for the two considered deployment scenarios. Finally, for the most challenging production environment, we analyze the local DDS overhead in terms of CPU and memory used at both publisher and subscriber nodes.

A. Methodology & Performance Metrics

To compare the performance of OpenSplice and RTI implementations we have run two main types of test experiments and used the following metrics, widely accepted in the related literature [4]. The first one is a stress test where the publisher sends a million packets, received by all subscribers, back-to-back and without any interval between subsequent transmissions. In the second experiment, instead, the publisher sends 5000 packets and waits an explicit application-level confirmation message from one of the

subscribers (one subscriber only replies, so to avoid altering the collected performance results due to confirmation message congestion) before sending the next message. All collected results are average values, evaluated over 33 runs.

For the stress test, we considered the following metrics:

- *Samples per-second*: the number of samples that the subscriber is able to receive per time unit (second);
- *Throughput*: total number of bytes received by the subscriber per time unit (second).

For the second test, instead, we measured:

- *Round-Trip Time (RTT)*: the time interval between the sending of a message and the reception of the associated acknowledgment from the chosen subscriber.

Finally, for both types of experiment, we quantify the overhead introduced at end nodes by the DDS support in terms of:

- *CPU*: the percentage of CPU used by all involved processes, collected via the `top` Linux command;
- *Memory*: the percentage of RAM memory used by all involved processes, again collected via the `top` Linux command.

In addition, in the following experiments we employ multicast communications with all nodes (publisher and subscribers) attached to the same LAN and non-reliable UDP communications (i.e., without explicit RTPS ACKs/NACKs). We do not report about packet losses because they were always very low and not significant for our goal. In all our experiments (including the stress tests) the DDS support was always able to deliver at least 99.999% messages. Finally, we have carefully tuned the configurations for both lab and production environments and adopted the default optimizations, namely, data batching active only for OpenSplice.

B. Lab environment

Before working on a real industrial production environment, we have thoroughly tested and evaluated the performance of OpenSplice and RTI in a more limited lab environment at our campus. Our University testbed consists of 2 Linux boxes with Intel Pentium Dual CPU at 1.8 GHz and 2 GB RAM, connected through a 100 Mbps LAN by following the deployment scheme. As for the software configuration, we used Linux Ubuntu 11.10, Java jdk1.6.0_16, the OpenSplice V5.5.1OSS Community edition with the RTNetworking protocol, and RTI Connext 5.0.0. We have one only Publisher, running on one of the two hosts, while 5 Subscribers are all deployed on the other host. Finally, we have collected performance metrics for different message sizes, ranging from 128 B (very small data chunks) to 512 KB (big messages, exceeding UDP 64 KB datagram size).

Starting from the first stress test (see Fig. 2), OpenSplice, due to the applied batching optimization, outperforms RTI for small data dimensions, from 128 B to 1 KB; that is also confirmed by the very high sample rates obtainable within in this range. After such a threshold, RTI because of the adoption of its self-contained decentralized architecture performs better than OpenSplice: it presents almost the same sample rate, but its RTPS protocol implementation is able to better exploit all available bandwidth resources. The good RTI trend continues until 32 KB message size, when the RTI implementation saturates the whole bandwidth available in our 100 Mbps FastEthernet, showing at the application level a throughput around 90 Mbps. Let us note that, after 64 KB it is necessary to split the payload into multiple UDP packets (that's also one of the reasons why the throughput continues to stay almost constant while the sample rate decreases as shown in Fig. 2-b and Fig. 2-a), and the RTI de-/fragmentation function seems to be highly optimized. OpenSplice, instead, cannot terminate the test and collapses at 32 KB; let us anticipate that this is due to the saturation of the DCPS daemon that, under such heavy system overload, continues to accumulate and consume RAM and CPU until

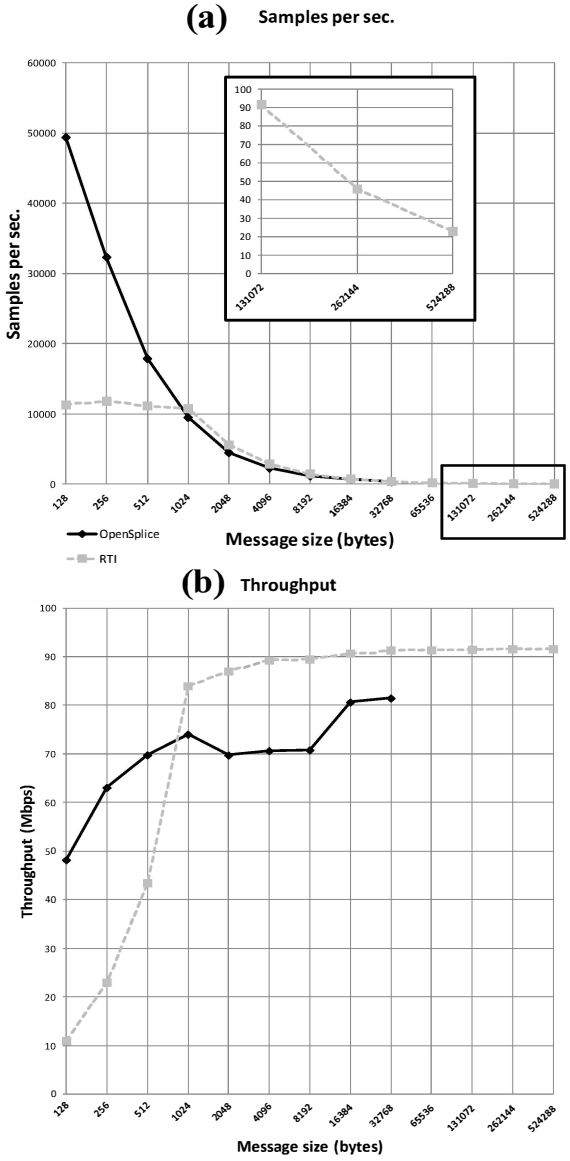

Figure 2. Stress test: (a) Samples per second and (b) Throughput in

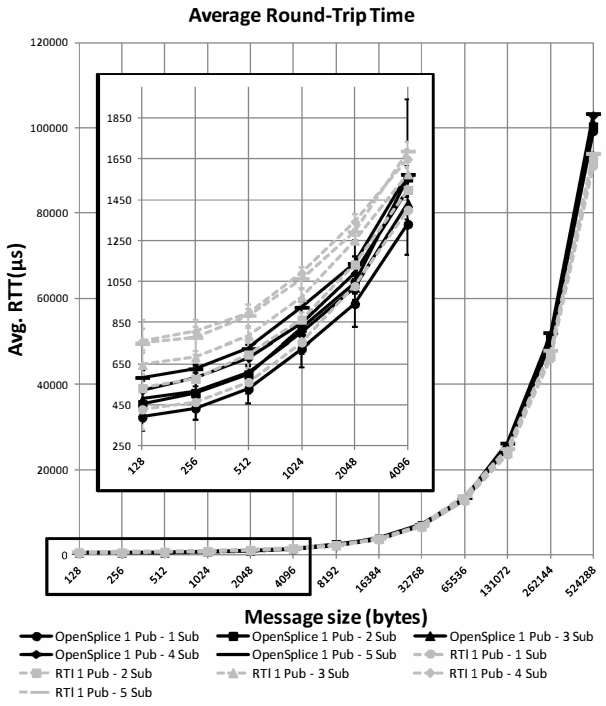

Figure 3. RTT in *Lab environment*.

its sudden crash (see overhead results, reported in the next section).

In the second test, OpenSplice federated architecture enables smaller RTT with very limited standard deviations, in the first [128, 4096] B interval (see Fig. 3 and the zoomed graphic therein); for higher data sizes, RTI again demonstrates a slightly better behavior that confirms its better scalability. In fact, for 64 KB, the RTI RTT graph crosses the OpenSplice one and, for the highest 512 KB data size, RTI average RTT is of 93 ms compared to the 103 ms RTT of OpenSplice. Finally, by using the multicast communication model, the increase of RTT at the increase of the number of subscribers is always very limited, for both RTI and OpenSplice.

C. *Production environment*

Passing to the real industrial production environment, the deployment consists of 21 Linux boxes with 2x Intel Xeon CPU 5440 quadcore at 2.67 GHz and 16 GB RAM, connected through a 1 Gbps LAN. As for the software configuration, we used Red Hat Enterprise Linux Server 5, Java jdk1.6.0_16, the OpenSplice OSPL-V 5.7.4p11_120423 enterprise edition with the RTNetworking protocol, and RTI Connext 5.0.0. We have one only Publisher, running on one of the hosts and 20 Subscribers deployed on other hosts (one per host).

In this environment, we repeated the same two tests shown before for the same message data sizes. We anticipate that, for both tests, collected performance results confirm the same trends, by scaling up due to the availability of a more powerful network and computing deployment environment. Starting from the stress test (see Fig. 4-a and Fig. 4-b),

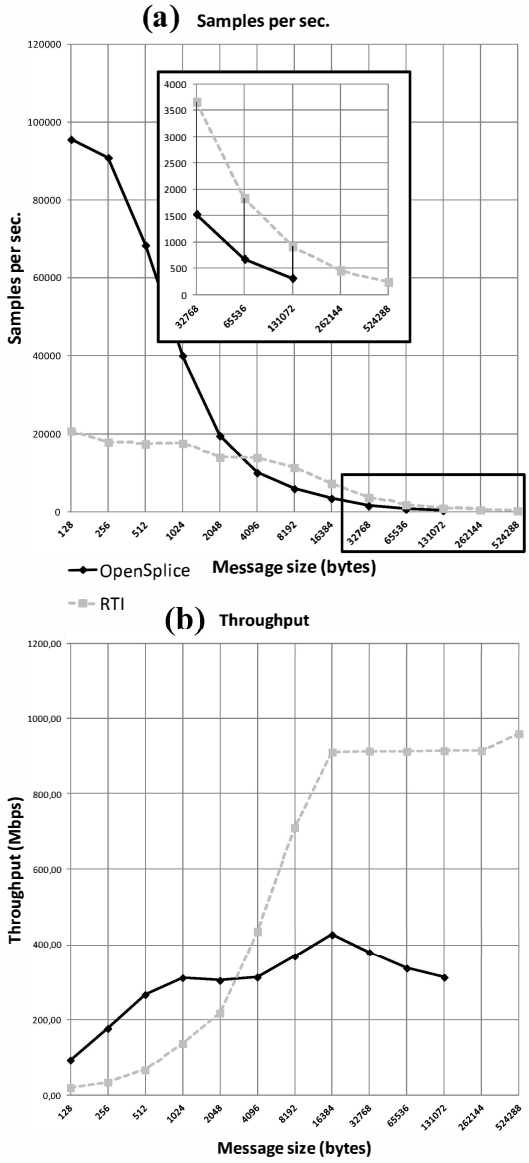

Figure 4. Stress test: (a) Samples per second and (b) Throughput in *Production environment*.

OpenSplice, that adopts a federated architecture and data batching, can obtain a very high data rate of 95,600 data delivered per second for the smallest 128 B message size. When the message size increases, RTI exhibits a better behavior and, while OpenSplice crashes at 112 KB, RTI continues to keep the pace with the increasing load even when, for data bigger than 16 KB, the 1 Gbps Ethernet saturates.

Focusing on RTT, the production environment emphasizes some behavior already observed in the lab one. The first and most notable is that the 1 Gbps Ethernet can grant very low RTT (latencies), almost one order of magnitude below the values measured in the Lab deployment, and always below 14 ms as Fig. 5 shows; in addition, the use of multicast communications can grant no significant differences in the increased RTT, even when the number of Subscribers consistently increases, up to 20

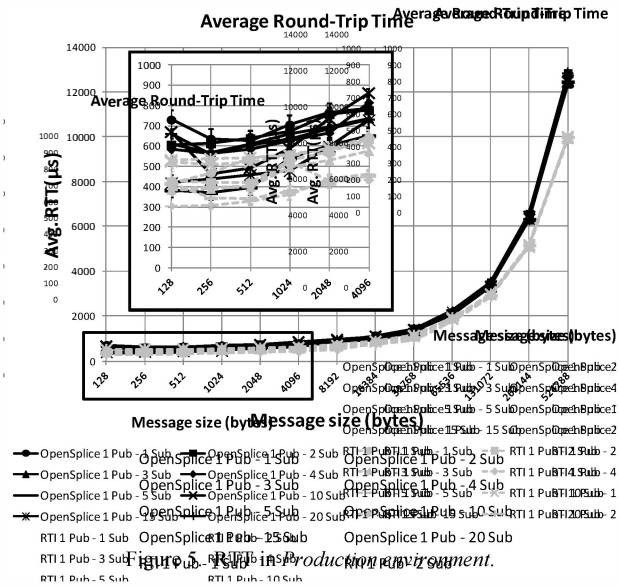

Figure 5. RTT in a *production environment.*

Subscribers. Moreover, collected results confirm a good scalability of RTI: at the most challenging 512 KB message size, RTI RTT is always below 10 ms while OpenSplice RTT is around 12.8 ms.

To conclude our performance analysis, we also measured the overhead introduced by the DDS communication middleware at all involved peer-hosts in terms of CPU and RAM resource costs (see Fig. 6).

We report the overhead for both stress test (Fig. 6-a and Fig. 6-b) and RTT test (Fig. 6-c and Fig. 6-d), for both CPU (Fig. 6-a and Fig. 6-c) and RAM (Fig. 6-b and Fig. 6-d) usage percentage. Moreover, for the first stress test, we focus on the Publisher and one of the Subscribers (representative of the behavior of all Subscribers), while for the RTT test, we also show the results of the responding Subscriber and of one representative of all other (non-responding) Subscribers. Starting from OpenSplice, test results demonstrate that the federated model based on the separation of the test application and the networking daemon and using the default RTNetworking protocol and data batching optimization can become a bottleneck of the DDS implementation at the publisher side. In fact, these two processes compete for the CPU resources by almost saturating it, while the networking daemon continues to accumulate and consume memory. We cannot ascertain whether that behavior is due to a memory leak, but results with both community and enterprise OpenSplice editions confirm the same behavior. RTI implementation, based on the decentralized model with a unique heavy process, shows a better and stabler behavior with reasonable a CPU overhead still far from saturation, even in the stress test, and very low memory consumptions (below 1%).

So, we can conclude that OpenSplice optimizations and better performance especially for small data are obtained at a rather high overhead cost that may hinder implementation scalability for very high message rate and sizes. RTI, instead, performs worse for small data, but then it scales very well due to a more performing message fragmentation and memory management functions.

V. Related Work

Since its first standardization, DDS has been employed for the development of highly scalable and fault-tolerant communication infrastructures in several different areas spanning from mission-critical systems to mobile systems [10, 11, 15], from data dissemination to Cloud monitoring infrastructures [16, 17], and many more. In the following, without any pretence of being exhaustive, we focus only on DDS performance evaluation studies, by surveying a selection of most evaluation significant efforts in this specific area.

Two seminal works in this area are [2] and [3]. [2] analyzes RTI performances of the under different load conditions, such as different publication rates and number of topics; however, the scalability at the growth of the number of subscribers is not analyzed, and all tests were carried out with the (limited) hardware deployments available in 2003. [3] is one of the first comparisons of RTI and OpenSplice showing that, for small messages, OpenSplice imposes lower network load, but at the same time it scales worse than RTI at the increase of the number of subscribers; the main drawback is that this study do not consider large-size messages.

Let us conclude this overview with a more recent contribution that compares qualitatively and quantitatively OpenSplice and RTI architectures [4]. This study confirms some of the performance trends we have obtained, but it does not inspect product performances under stress conditions and for large-size messages; most important, it does not evaluate the additional overhead at the communicating peers.

VI. Conclusive Remarks and Ongoing Work

The original and practical research work presented in this paper should provide researchers and IT administrators with a useful guide for the selection and the configuration of the DDS implementation product that best fits their application and deployment requirements. In particular, our work shows that different distributed architectures induce different tradeoffs: federated models tend to enable better local optimizations, such as local data batching for optimizing small-size data exchange; centralized models, instead, are more suitable for high data rates and large-size messages, especially when working close to resource saturation conditions (more predictable performance trend, usually more suitable for highly demanding industrial exploitation).

The interesting performance results measured and presented in this paper have stimulated us to proceed in our DDS experimentation and benchmarking work along two primary directions. First of all, we are working to extend our measurements campaign by considering the concurrent utilization of DDS with differentiated QoS levels for different classes of exchanged data. Secondly, we are

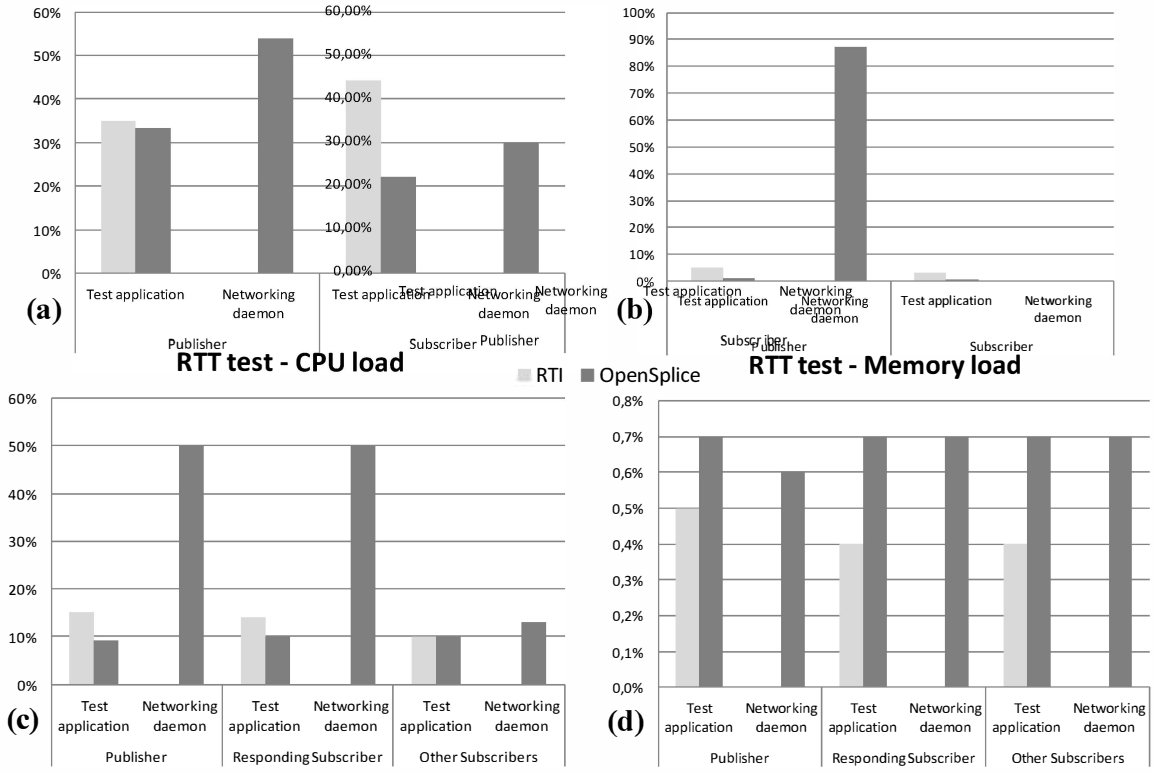

Figure 6. OpenSplice and RTI overhead in *Production environment*: (a) throughput test - CPU load; (b) throughput test - memory load; (c) RTT test - CPU load; (d) RTT test - memory load.

benchmarking the performance results achievable by including application-level mechanisms to either reduce message size (through transparent fragmentation and reconstruction of exchanged data) or increase it (through transparent merging/batching of messages when the consequent growth of data latency is allowed) at runtime depending on dynamic application requirements and the currently experienced DDS performance.

ACKNOWLEDGMENT

This research was supported in part by CIRI, Center for ICT technology transfer of the University of Bologna; we also thank Ente Nazionale per l'Assistenza al Volo (ENAV) for the support in the thesis of Alessandro Pernafini.

REFERENCES

[1] Object Management Group (OMG), "Data Distribution Service for Real-time Systems", http://www.omg.org/spec/DDS/1.2.

[2] S. Sierla, J. Peltola, K. Kiskinen, "Evaluation of a Real-Time Distribution Service", 2003, URL: http://www.rti.com/docs/RTPS_Overview_HUT.pdf.

[3] B. McCormick, L. Madden, "Open Architecture Publish-Subscribe Benchmarking", OMG Real-time and Embedded Systems Workshop, 2005, URL: http://www.omg.org/news/meetings/workshops/RT_2005/03-3_McCormick-Madden.pdf

[4] M. Xiong *et al.*, "Evaluating the Performance of Publish/Subscribe Platforms for Information Management in Distributed Real-time and Embedded Systems", OMG Data Distribution Service Portal, 2006, http://portals.omg.org/dds/content/document/evaluating-performance-publishsubscribe-platforms-information-management-distribute.

[5] L. David *et al.*, "A Large-scale Communication Middleware for Fleet Tracking and Management", Brazilian Symp. on Computer Networks and Distributed Systems (SBRC), 2012.

[6] PrismTech, OpenSplice DDS, web page (last visited in February 2013): http://www.prismtech.com/opensplice/

[7] Real Time Innovations, Connext DDS, web page (last visited in February 2013): http://www.rti.com/products/dds/

[8] P. Eugster *et al.*, "The many faces of publish/subscribe", ACM Computing Surveys, vol. 35, no. 2, Jun. 2003.

[9] OMG, "The Real-time Publish-Subscribe Wire Protocol - DDS Interoperability Wire Protocol Specification", Version 2.1, 2009, http://www.omg.org/cgi-bin/doc?formal/09-01-05.pdf

[10] C. Esposito *et al.*, "Achieving Reliable and Timely Event Dissemination over WAN", Int. Conf. on Distributed Computing and Networking (ICDCN), 2012.

[11] J.M. Lopez-Vega *et al.*, "A content-aware bridging service for publish/subscribe environments", Elsevier Journal of Systems and Software, vol. 86, no. 1, Jan. 2013.

[12] OCI OpenDDS, web page (last visited in February 2013): http://www.ociweb.com/products/opendds/

[13] PrismTech OpenSplice RTNetworking, web page (last visited in February 2013): http://www.prismtech.com/opensplice/products/communication

[14] Real Time Innovations, RTI Connext Core Libraries and Utilities User's Manual, Version 5.0, 2012.

[15] A. Corradi, L. Foschini, L. Nardelli, "A DDS-compliant infrastructure for fault-tolerant and scalable data dissemination", IEEE Symp. on Computers and Communications (ISCC), 2010.

[16] A. Corradi, L. Foschini, J. Povedano-Molina, J.M. Lopez-Soler, "DDS-enabled Cloud management support for fast task offloading", IEEE Symp. on Computers and Communications (ISCC), 2012.

[17] P. Bellavista, A. Corradi, L. Foschini, "Enhancing Intra-Domain Scalability of IMS-based Services", early access article in IEEE Transactions on Parallel and Distributed Systems, DOI: 10.1109/TPDS.2012.312, 2012.