An Investigation on Flexible Communications in Publish/Subscribe Services

Christian Esposito[2], Domenico Cotroneo[1], and Stefano Russo[1,2]

[1] Dipartimento di Informatica e Sistemistica (DIS)
Università di Napoli Federico II, Napoli, 80125 - Italy
[2] Laboratorio CINI-Item "Carlo Savy"
Consorzio Interuniversitario Nazionale per l'Informatica (CINI),
Napoli 80125 - Italy
{christian.esposito,cotroneo,sterusso}@unina.it

Abstract. Novel embedded and ubiquitous infrastructures are being realized as collaborative federations of heterogeneous systems over wide-area networks by means of publish/subscribe services. Current publish/subscribe middleware do not jointly support two key requirements of these infrastructures: timeliness, *i.e.*, delivering data to the right destination at the right time, and flexibility, *i.e.*, enabling heterogeneous interacting applications to properly retrieve and comprehend exchanged data. In fact, some middleware solutions pay more attention to timeliness by using serialization formats that minimize delivery time, but also reduce flexibility by constraining applications to adhere to predefined data structures. Other solutions adopt XML to improve flexibility, whose redundant syntax strongly affects the delivery latency.

We have investigated the consequences of the adoption of several lightweight formats, which are alternative to XML, in terms of flexibility and timeliness. Our experiments show that the performance overhead imposed by the use of flexible formats is not negligible, and even the introduction of data compression is not able to manage such issue.

Keywords: Flexibility, Timeliness, Serialization Formats, Pub/Sub Services.

1 Introduction

Typically, embedded systems have been architected according to a "*closed world*" perspective: a series of computing machines were interconnected by dedicated networks, but with limited, or no, cooperation with the outside world. Therefore, frameworks that have to carry out complex control activities, such as the Air Traffic Management (ATM) or the Power System Control (PSC), have been fragmented into "*islands of automation*", *i.e.*, they are composed by several autonomous and independent systems, each one in charge of controlling an isolated portion of the overall framework, but with no reciprocal cooperation. Inefficiencies of such traditional perspective and recent developments in networking are causing an evolution of embedded systems, leading to the so-called Large scale

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 204–215, 2010.
© IFIP International Federation for Information Processing 2010

Complex Critical Infrastructures (LCCI). Such infrastructures adopt a federated, "*open world*" architecture, *i.e.*, LCCI consist of dynamic Internet-scale hierarchies/constellations of interacting heterogeneous systems, which cooperate to carry out critical functionalities. Many of the ideas behind LCCI are increasingly "in the air" in several current projects that aim to devise innovative critical embedded systems. For example, EuroCONTROL has funded a project, called "*Single European Sky ATM Research*" (SESAR)[1], to develop the novel European ATM framework as a seamless infrastructure allowing control systems to cooperate in order to better handle the growing avionic traffic.

This novel perspective is also enforcing the integration of several different kinds of IT infrastructures, which are usually strictly distinct and separated from critical embedded systems. A practical example is provided by another EU project, called "*Total Airport*"[2], whose scope is the integration of all subsystems for land-side and air-side activities and their information flows. Specifically, critical embedded systems and the relative federation middleware under development within the context of SESAR are going to be integrated with all the IT components for airport management, and also with ubiquitous systems for safe and secure passenger and luggage management, so to realize a seamless "*door-to-door*" control (*i.e.*, from when entering the departing airport until leaving the arriving airport). *E.g.*, passengers can access to certain ATM information via their smart phones to know the status or schedule of their flights, to track their luggages, to locate themselves within the airport map or even to receive commercial ads. Such innovative embedded and ubiquitous systems require several non-func-tional requirements to be satisfied, among which there is flexibility, *i.e.*, interacting entities must be able to comprehend each other even if they do not know the structure applied by the data source to the exchanged messages. The widely-adopted middleware solutions in federating heterogeneous systems are the ones that adopt the publish/subscribe interaction model, called pub/sub services, due to its intrinsic decoupling properties that enforce efficient and scalable data dissemination. However, most of these solutions adopt serialization formats that negatively affect the flexibility offered by the middleware. A widely-adopted solution to resolve such drawback is to adopt XML as serialization format; however, this is not a winning choice due to the high performance overhead of XML-based communications.

In this paper we study the use of two lightweight flexible formats in Subsection 3.2, and discuss their performance in Section 4. Our experiments have shown that flexibility is always obtained at high expenses of performance. Therefore, we investigate in Section 6 the effects of introducing data compression (briefly described in Section 5) to reduce the performance drawbacks of flexible formats. The conducted measurement campaigns have revealed that data compression is not able to provide a considerable improvement of performance, but is able to better tolerate message losses.

[1] www.eurocontrol.int/sesar/

[2] www.eurocontrol.int/eec/public/standard_page/EEC_News_2006_3_TAM.html

2 Problem Statement

Large-scale systems are rarely built ex-novo, but it is more probable that they are developed starting from already-existent legacy systems by using a middleware and other proper abstractions to federate them. Federating legacy systems, built by different companies at different times and under different regulation laws, raises the so-called *Data Exchange Problem* [1]. Specifically, let us consider an application, namely A_{source}, which is a data source and is characterized by a given schema, indicated as S_{source}, for the produced data, and another one, namely A_{dest}, which is a data destination and is characterized by a given schema, indicated as S_{dest}. When the two schemas diverge, a communication can take place only if a mapping M between the two schemas exists. This allows the destination to comprehend the received messages and to opportunely use them within its application logic. When the two schemas are equal, the mapping is simply the identity. On the contrary, when several heterogeneous legacy systems are federated, it is reasonable to have diverging data schemas, and the middleware solution used for the federation needs to find the mapping M and to adopt proper mechanisms to use it in the data dissemination process. Moreover, the communication pattern adopted in collaborative infrastructures is not one-to-one, but one-to-multi or multi-to-multi. So, during a single data dissemination operation, there is not a single mapping M to be considered, but several of them, *i.e.*, one per each destination.

EUROCONTROL has tried to find a solution to this problem by specifying a standard schema for the flight data exchanged among ACCes, called ATM Validation ENvironment for Use towards EATMS (AVENUE)[3]. However, even if neglecting the strong additional overhead introduced by mapping to/from the standard data structure both at publisher and subscriber side, this workaround does not completely resolve the issue. In fact, over the time, a standard data format is likely to be changed in order to address novel issues or to include more data (in fact, in the last three years AVENUE has been updated several times). However, not all the systems may be modified to handle new versions, so there may be systems with different versions. This brings back the Data Exchange Problem when a publisher produces events with a certain version of the standard data structure and subscribers can comprehend only another versions.

To completely resolve such issue, the viable solution is to realize a *flexible communication*: the data source does not care about the schemas of the receivers. On the other hand, data destinations are able to introspect the structure of received messages and use such information to properly feed data instances.

3 Serialization Formats in Publish/Subscribe Middleware

Pub/sub services are very appealing to efficiently interconnect several systems due to their intrinsic decoupling properties that promote scalability [2]. In fact, a recent OMG specification for pub/sub services, defined by OMG and called Data

[3] `www.eurocontrol.int/eec/public/standard_page/ERS_avenue.html`

Distribution Service (DDS) [3], has been chosen by EUROCONTROL as the reference technology for its novel European ATM framework under development within the context of the SESAR project. To study if pub/sub services are able to provide flexible communication to address the Data Exchange Problem presented in the previous section, it is crucial to analyze the features of serialization formats that they adopt. Therefore, in the following Subsection 3.1, we present the main serialization formats adopted in the current pub/sub services, and discuss their pros and cons to support flexible communication. Instead, in Subsection 3.2 we introduce new serialization formats as suitable alternatives.

3.1 Current Serialization Formats

CDR. Some of the available pub/sub services adopt serialization formats that can be defined as *binary*, and a practical example is the *Common Data Representation* (CDR) [4], adopted by all products compliant to DDS specification. Binary formats are based on a positional approach: serialization, and relative deserialization, operations are performed according to the position occupied by data within the byte stream. To better explain how binary formats work, let us consider a publisher and subscriber exchanging a certain data instance. The publisher goes through all the fields of the give data instance, converts the content of each field in bytes, and stores it in a byte stream, treated as a FIFO queue. On the subscriber side, the application feeds data instances with information conveyed by received byte streams. Specifically, knowing that the serialization of the first field of type T requires a certain number, namely n, of bytes, the subscriber extracts the first n bytes from the byte stream. Then, it casts such n bytes in the proper type T and assigns the obtained value to the field in its own data instance. Such operation is repeated until the entire data instance is filled.

CDR does not support a flexible communication. In fact, the ability of the subscriber to comprehend received messages, *i.e.*, to obtain original data instances starting from received byte streams, is coupled to the knowledge of the data structure at the publisher side. On the other hand, since only instance content is delivered throws the network, formats such as CDR exhibit a serialization stream characterized by a minimal size.

XML. When a middleware solution wants to provide flexible communication, it typically uses serializations formats defined as *tree-based*, *i.e.*, they embodies in the serialization stream not only instance content, but also meta-information, organized as a tree, about the internal structure of the data instance. Such meta-information allow decoupling interacting applications from the reciprocal knowledge of the internal structure of the data that they are exchanging. The widely-adopted tree-based format is *XML*, which specifies the structure of data content by a combination of opening and closing tags. In fact, there has been an increasing demand for XML-based pub/sub services, which support flexible document structures and subscription rules expressed by powerful languages, such as XPath and XQuery [5].When using XML, the publisher transforms a data

instance in an XML document by placing the content of its fields between tags with the same name of the given field. Such document is further converted into a byte stream and delivered to the subscriber, which uses such XML document to feed a data instance by assigning to each field the value between tags equal to the field name.

Adopting XML allows the subscriber to be not aware of the data structure applied at the publisher side since stream structure is no more implicit, but explicit into the tags. So, flexible communication is supported; however, such flexibility is achieved at the expenses of delivery latency. In fact, XML syntax is redundant or larger with respect to binary formats of similar data, and this redundancy may affect application efficiency through higher transmission and serialization costs.

3.2 Lightweight Flexible Serialization Formats

Some application scenarios for collaborative IT infrastructures show a time-critical behaviour: delivering information out of time boundaries could lead to instability (*timeliness*). So, it is important to minimize the transmission latency. For this reason, XML is not the viable solution to support flexibility, but other tree-based formats that exhibit lower performance costs are needed. In literature there are available other tree-based formats that are simpler than XML while maintaining its flexibility guarantees. In fact, they have been specifically designed as a data interchange format, not a markup language. So, the dimension of the serialized stream is smaller than the one obtainable with XML, because there is no redundant syntax, *e.g.*, no closing tags. In the rest of this subsection, we will present two of these "lightweight" flexible serialization formats.

JSON. Java Script Object Notation (JSON)[4] is a lightweight data-interchange format, based on a subset of the JavaScript Programming Language[5]. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages. It is built on two structures: a collection of name/value pairs, and an ordered list of values. When using JSON, serialization and deserialization is performed as seen in the case of XML, but using a collection of name/value pairs allows saving bytes in the serialization stream.

YAML. YAML Aint Markup Language (YAML)[6] is a data serialization format that takes concepts from languages such as XML, C, Python, Perl, as well as the format for electronic mail as specified by RFC 0822[7]. Its syntax is relatively

[4] `www.json.org/index.html`
[5] Standard ECMA-262, 3rd Edition - December 1999. More details available at `www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf`
[6] `www.yaml.org`
[7] `www.ietf.org/rfc/rfc0822.txt`

straightforward, with data structure hierarchy maintained by outline indentation, which facilitates easy inspection of the data structure. YAML can waste bytes, since each space of the indentation must be translated as a character in the serialization stream. To overcome this problem, its possible to use a compact version of YAML by replacing indentation with brackets.

Comparison. Both JSON and YAML share very similar syntax; however, they also exhibit certain differences. JSON is trivial to generate and parse. It also uses a lowest common denominator information model, ensuring any JSON data to be easily processed. On the other hand, YAML is more complex to generate and parse. In addition, YAML ventures beyond the lowest common denominator data types, requiring more complex processing.

4 Experimental Evaluation of Serialization Formats

The goal of this section is to compare all the serialization formats illustrated in the previous section by analyzing their quality in terms of two measures: *serialization efficiency*, *i.e.*, how many bytes in the serialization stream are added to the instance content, and *latency*, *i.e.*, how much time is needed to exchange data instances from a publisher to a subscriber.

4.1 Experiment Setup

We have realized a prototype to exchange data instances that are structured according to the AVENUE type, which is characterized by a complex structure made of about 30 nested fields and a size of almost 100 KB. In addition, we have used an implementation of DDS, provided by RTI, as pub/sub service to exchange messages between a publisher and a subscriber. Moreover, we have implemented a component, named Parser and placed it between the application and the middleware, which takes data instances from the application and returns byte streams to the middleware and vice versa. Within such component, we have embodied the following parsers: (*i*) an in-house developed parser for CDR and YAML, (*ii*) XERCES parser[8], using both DOM and SAX, for XML, and (*iii*) JOST parser[9] for JSON.

The experiments conducted to evaluate latency adopt a "ping-pong" pattern, illustrated in Figure 1. Specifically, the publisher feeds a data instance with randomly-generated content and passes it to the parser, which returns a byte stream to the middleware for disseminating it. On the subscriber side, the middleware receives the byte stream, which is passed to the subscriber application. Then, the subscriber application immediately makes a copy of the received stream and sends it back to the publisher, which receives the original message after the stream passed through the parser component. Along the path from the

[8] www.apache.org/xerces
[9] ddsbench.svn.sourceforge.net/viewvc/ddsbench/trunk/jost/

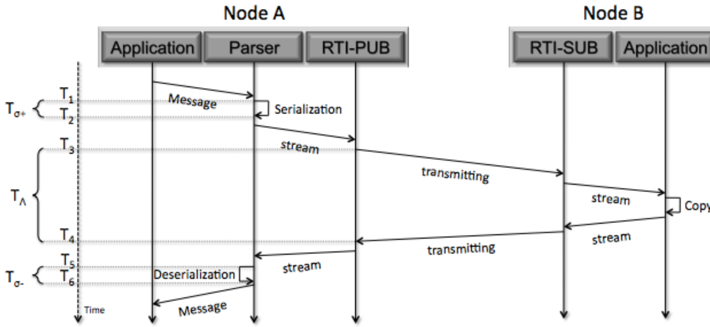

Fig. 1. Serialization and Deserialization operated according to YAML

publisher to the subscriber and backward, we take several timestamps in order to characterize the achievable latency in terms of the following three contributions:

1. *serialization time* $(T_2 - T_1)$, *i.e.*, time spent by the parser to convert data instances into byte streams;
2. *deserialization time* $(T_6 - T_5)$, *i.e.*, time taken by the parser to convert byte streams into data instances;
3. *delivery time* $(T_4 - T_3)$, *i.e.*, time needed for a message to go from the publisher to the subscriber and backward.

4.2 Results

Figure 2 illustrates the outcomes of our experimental campaign. It is not unexpected that CDR presents the highest efficiency, but it is surprising how bad the tree-based formats perform, exhibiting a mean efficiency of 0,0854, which is pretty far from the efficiency achieved by CDR. Among the tree-based formats, the ones with the better efficiency are JSON and the compact version of YAML (respectively with an efficiency equal to 0,114 and 0,105), while the worst efficiency has been registered for XML (*i.e.*, the efficiency is equal to 0,0678). Efficiency affects the measured delivery time, shown in Figure 2D, which is higher when using data format with lower efficiency due to the higher number of bytes to exchange. Figures 2B and 2C, which respectively illustrate serialization and deserialization time, are more interesting. They show that CDR has the best performances due to the simplicity of the parsing operations. With respect to the tree-based formats, JSON and compact version YAML realize again the best performance considering serialization time, while full version of YAML and XML[10] have the worst one. Considering the deserialization, Full and Compact versions of YAML present performance closer to XML with SAX, while XML with DOM achieves the highest measured deserialization time, and JSON presented the lowest deserialization time.

[10] Even if we used SAX in our experiments, DOM is still used to carry out the serialization duties.

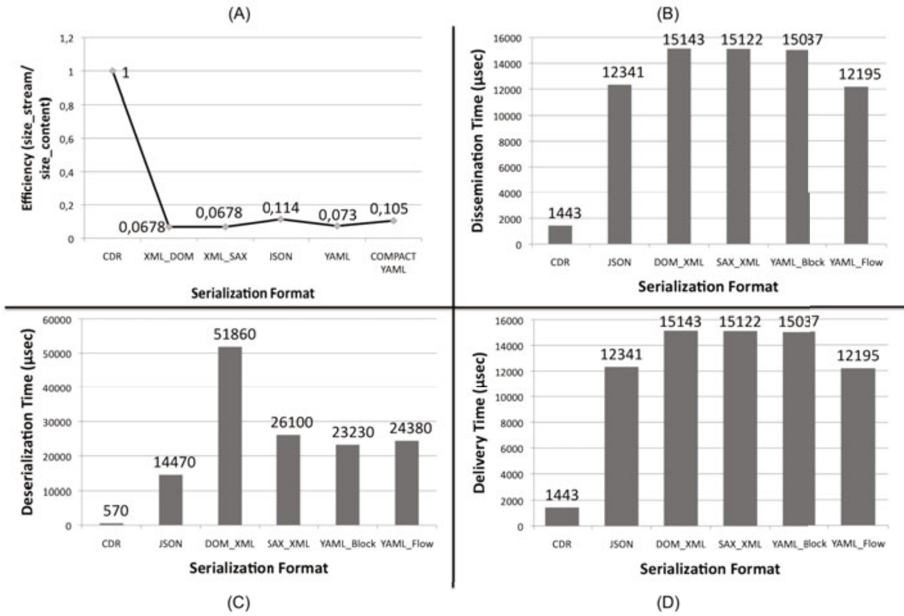

Fig. 2. Experimental Results analyzing the following metrics: (A) serialization efficiency, (B) serialization time, (C) deserialization time and (D) delivery time

5 Data Compression

Previous experimental results have clearly proved the considerable performance overhead implied by tree-based formats, making them inapplicable in application scenarios where timeliness is also a key requirement to be satisfied. A possible solution to limit this drawback is to use *data compression techniques* [6] after the parser and before the middleware layer. During the last years, several data compression techniques have been presented by academia or industry. Such techniques can be classified in two distinct classes: *lossy*, *i.e.*, some pieces of information may be lost after decompression, and *lossless*, *i.e.*, pieces of information are never lost after decompression. Since we do not want to incur in any occurrence of data losses, we have preferred techniques belonging to the second class. The most used lossless compression techniques are the following ones: (i) optimal coding of Huffman, (ii) Lempel-Ziv (LZ) algorithm, and (iii) Run-length encoding (RLE). Such techniques are known to achieve between 50% and 30% as compression efficiency (*i.e.*, the compressed stream presents 50% - 30% less bytes than the original stream). If higher compression efficiency is needed, the literature is rich of *hybrid schemas* that combine the previous techniques: (i) zlib[11], which adopts the "*DEPLATE*" method to combine LZ and Huffmann

[11] www.zlib.net/

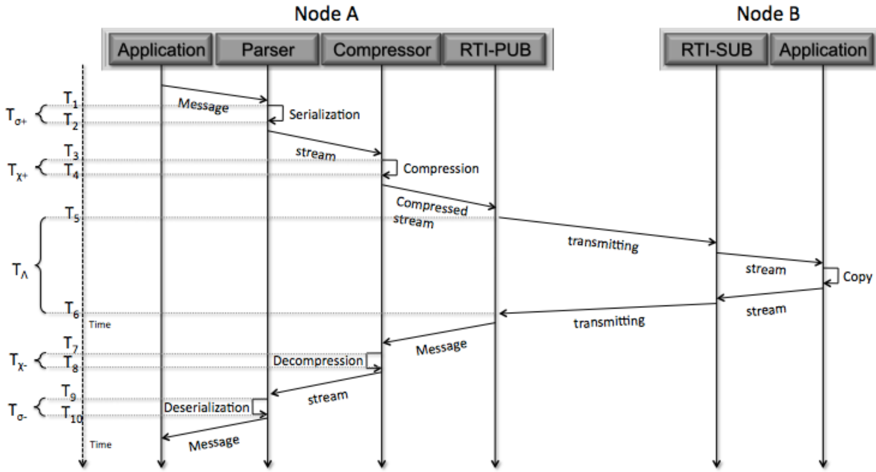

Fig. 3. Serialization and Deserialization operated according to YAML

Coding, (*ii*) bzip2[12], which uses the Burrows-Wheeler block sorting technique and Huffman coding, and (*iii*) Lempel-Ziv-Oberhumer (LZO) algorithm[13].

6 Experimental Evaluation of Serialization Formats with Data Compression

This section presents the two kind of experiments that we have conducted. The first campaign is similar to the one presented in Section 4 and aims at showing the improvement in performance and efficiency when tree-based formats are teamed up with compression techniques. On the other hand, the goal of the second one is to study the behaviour of the developed prototype in a real case scenario that uses Internet as Interconnection network.

6.1 Experiment Setup

We have modified the prototype used in the previous experiments by introducing an additional component, named Compressor, which embodies data compression techniques, as clearly shown in Figure 3. Therefore, we have characterized the achievable performance as in Subsection 4.1 but adding two more contributions:

– *compression time* ($T_4 - T_3$), *i.e.*, time spent to perform compression;
– *decompression time* ($T_8 - T_7$), *i.e.*, time elapsed to make decompression.

[12] www.bzip.org/
[13] www.oberhumer.com/opensource/lzo/

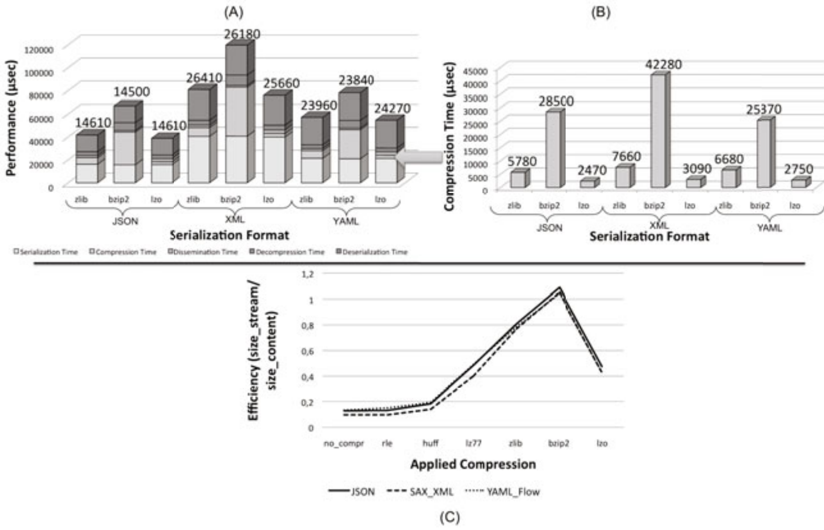

Fig. 4. (*A*) performance, (*B*) compression time, and (*C*) efficiency of first campaign

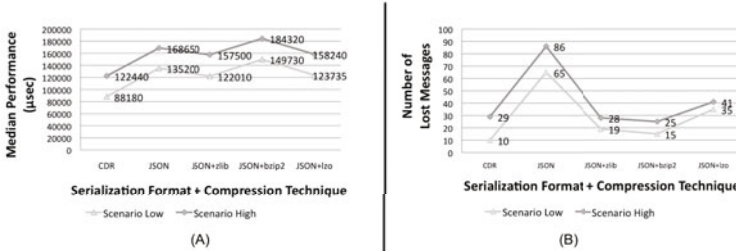

Fig. 5. (*A*) performance and (*B*) resiliency of the second experimental campaign

With respect to the campaign using a real case scenario, *i.e.*, applications interconnected by wide area networks as Internet, experiments could not be performed using a realistic testbed, such as PlanetLab[14], because it has not been designed to perform controlled experiments and to achieve reproducible results [7]. Thus, we have chosen to adopt an emulation approach: the publisher and subscriber are executed on distinct machines, interconnected by a network emulator called Shunra Virtual Enterprise (VE)[15], which allows users to recreate a specific network behavior according to a defined model. The model adopted in this work is the *Gilbert Model* [8], one of the most-commonly applied amodel in performance evaluation studies, due to its analytical simplicity and the good results it provides in practical applications on wired IP networks [9]. The Gilbert Model is a 1-st order Markov chain model characterized by two states: state 0, with no

[14] www.planet-lab.eu
[15] www.cnrood.nl/PHP/files/telecom_pdf/Shunra_Virtual-Enterprise.pdf

losses, and state 1, with losses. There are four transition probabilities: (*i*) the probability to pass from state 0 to state 1 is called P, (*ii*) the probability to remain in state 0 is (1 - P), (*iii*) the probability to pass from state 1 to state 1 is called Q, and (*iv*) the probability to remain in state 1 is (1 - Q). Given PLR and ABL, P and Q are computed as follows: $P = \frac{PLR \cdot Q}{1-PLR}$ and $Q = ABL^{-1}$. The values for PLR and ABL have been obtained by a measurement campaign conducted on PlanetLab along some European Internet paths. Using these Measurements, we have defined two different scenarios: *Low Scenario*, with low values for PLR and ABL (respectively 1,35 and 0,59), and *High Scenario*, with higher values for PLR and ABL (respectively 5,05 and 1,44).

6.2 Results

First Experimental Campaign. In Figure 4C, the efficiency of the three best tree-based formats (*i.e.*, JSON, XML with SAX and compact YAML) is analyzed without compression and with each of the techniques described in Section 5. The highest efficiency, even better than the one of CDR, is achieved by using hybrid compression schemas. Figure 4A shows that bzip2 (*i.e.*, the techniques with best efficiency) is the one with worst performances, while the other two are quite similar. Figure 4B reveals that LZO achieves faster compression (and also decompression, not shown in the paper due to the limited available space), however, such strength is nullified by its lower compression efficiency, as shown in Figure 4C. Last, we can conclude that the best technique is zlib since it realizes the optimal trade-off between efficiency and performance.

Second Experimental Campaign. Figure 5 shows results with the testbed made with the Shunra emulator. Figure 5A proves that zlib is able to reduce the performance of tree-based formats both in Low and High Scenarios, behaving better than the other two hybrid schemas. However, there is still a strong difference than using CDR. However, another result shown in Figure 5B is surprising: data compression is a mean to increase the resiliency of the middleware. In fact, reducing the number of exchanged bytes can bring to less packets losses. Specifically, the technique with highest efficiency, *i.e.*, bzip2, presents the lower number of losses, achieving a resiliency degree close to CDR.

7 Conclusion

In the present article we have discussed the issue of providing flexible communication in pub/sub services. We argued that the commonly-used serialization formats, such as CDR and XML, are unsuitable since they do not exhibit an optimal trade-off between flexibility and performance. We have investigated the use of other two formats, such as JSON and YAML, which are claimed to be light-weighted version of XML. However, our results proved that, even using such formats, the performance overhead is still higher than the case of using the inflexible CDR. Then, we have proposed the introduction of data compression

to reduce such weakness. However, benefits in performance thanks to reducing the exchanged bytes is strongly mitigated by the time needed to perform compression and decompression operations. Last, we have conducted experiments in real case scenarios, which revealed us that data compression is also an effective solution to increase the resiliency by reducing the number of lost messages.

Acknowledgment

This work has been partially supported by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) “DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures”.

References

1. Kolaitis, P.G.: Schema Mappings, Data Exchange, and Metadata Management. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 90–101 (June 2005)
2. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)
3. Object Management Group, Data Distribution Service (DDS) for Real-Time Systems, v1.2, OMG Document (2007)
4. Object Management Group, Common Object Request Broker Architecture (CORBA), v3.0, OMG Document, pp. 15.4–15.30 (2002)
5. Zhao, J., Yang, D., Gao, J., Wang, T.: An XML Publish/Subscribe Algorithm Implemented by Relational Operators. In: Dong, G., Lin, X., Wang, W., Yang, Y., Yu, J.X. (eds.) APWeb/WAIM 2007. LNCS, vol. 4505, pp. 305–316. Springer, Heidelberg (2007)
6. Sayood, K.: Introduction to Data Compression, 3rd edn. Series in Multimedia Information and Systems. Morgan Kaufmann, San Francisco (2005)
7. Spring, N., Peterson, L., Bavier, A., Pai, V.: Using PlanetLab for network research: myths, realities, and best practices. ACM SIGOPS Operating Systems Review 40(1), 17–24 (2006)
8. Yu, X., Modestino, J.W., Tian, X.: The Accuracy of Gilbert Models in Predicting Packet-Loss Statistics for a Single-Multiplexer Network Model. In: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM ’05), vol. 4, pp. 2602–2612 (March 2005)
9. Konrad, A., Zhao, B., Joseph, A.: Determining Model Accuracy of Network Traces. Journal of Computer and System Sciences 72(7), 1156–1171 (2006)