Compact and Efficient Agent Messaging

Kai Jander and Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{jander | lamersd}@informatik.uni-hamburg.de

Abstract. Messages are considered to be a primary means of communication between agents in multi-agent systems. Since multi-agent systems are used for a wide variety of applications, this also includes applications like simulation and calculation of computer generated graphics which need to employ a large number of messages or very large messages to exchange data. In addition, other applications target hardware which is resource constrained by either bandwidth or processing capacity. As a result, these applications have different requirements regarding their messages. This paper proposes a number of useful properties for agent messages and evaluates them with regard to various types of applications. Based on this evaluation a message format for Jadex called Jadex Binary is proposed, which emphasizes properties that are not traditionally the focus of agent message formats and compared them to some well-known formats based on those properties.

1 Introduction

Multi-agent systems enable the development of scalable and highly dynamic applications, facilitating their deployment on infrastructure such as structurally or spatially distributed systems, and the integration of mobile devices in such systems. An important means for agents to coordinate within an multi-agent application are messages passed between them. This mechanism is one of the enabling factors for autonomous behavior of agents, which enables them to be protected from direct influence by the rest of the system and establish a measure of robustness.

Nevertheless, certain classes of applications deployed on such systems have special requirements which appear to be in conflict with the focus of traditional agent message formats. Examples of this type of applications include real-time audio and video communication, distributed simulation and real-time distributed computer generated image (CGI) animation.

Traditionally, these requirements have not been the focus of agent messaging, which tends to target other useful properties that can also be important in multi-agent applications. As a result, it would broaden the application scope of agent systems if they specifically supported the requirements of such applications by providing an alternative message format.

In the following section we will attempt to identify typical requirements for agent messages and distill some that are especially important to the aforementioned classes of applications. We will then introduce some typical message formats used in agent systems and attempt to identify which application requirements they attempt to fulfill. Finally, we will present a compact message format that caters to the special class of application with real-time and bandwidth-restricted sets of requirements and compare it to traditional agent message formats, demonstrating key advantages for this special set of applications.

2 Features of Agent Message Formats

Since multi-agent applications cover a large spectrum of potential applications, there is an equally large number of features associated with agent messages which are potentially useful for different classes of applications. In addition to different application classes, different points in the development cycle may also emphasize the importance of certain features over others. While there is a large number of arguably useful features, we propose the following six features which are commonly mentioned and requested for multi-agent applications:

- *Human Readability* allows humans to read messages with standard tools like text viewers without the help of decoders or other special tools.
- *Standard Conformance* requires messages to conform to a published message format standard or language standard, allowing interaction between systems conforming to those standards.
- A *Well-formed Structure* defines a valid form for messages, allowing the system to distinguish between valid and invalid messages.
- *Editability* goes beyond human readability by allowing users to edit and restructure messages using standard tools such as text editors.
- *Performance* describes the computational requirements to encode and decode messages.
- *Compactness* evaluates the size of encoded messages.

In order to evaluate these features, we propose an example set of four common classes of applications. While there are many more potential applications, these applications are very common and would benefit from the use of agent technology. The first type of applications are *real-time applications*, where latency is a primary concern. Examples of this type of application can be found in any real-time communication application such as voice or video conference systems. High latency is generally unacceptable in such applications and may severely inhibit their functionality.

The second type of applications are *cross-platform applications*. For example, Agentcities[1] allows the use of multiple agent platforms and multiple types of agents, requiring precise definitions and standards among them. Correct interpretation of messages from other agents or platforms is key for such applications.

Another common type of applications involve *enterprise backend applications*. These applications often run on application servers on high-performance

intranets. It is important for such applications to provide quick access to the services required by the business in order to maintain high productivity.

The final type of applications are *mobile applications*, where a large number of nodes in the application are physically mobile and are typically connected using wireless connections. This means that the nodes are often restricted in terms of computational resources and network bandwidth. Energy supply is a key factor, stipulating modest use of resources even when more would be available.

	Real-time Applications	Cross-platform Applications	Enterprise Backend Applications	Mobile Applications
Human Readability	low	medium	low	low
Standard Conformance	low	high	medium	low
Well-formed Structure	low	high	low	low
Editable	low	medium	medium	low
Performance	high	low	high	high
Compactness	high	low	low	high

Fig. 1. Importance of message format features for different types of applications

Figure 1 shows the application types and the importance of the message format features. Some application types such as real-time applications and mobile applications have similar feature importance profiles for different reasons. While latency requires prudent use of resources for real-time applications, it is the energy and physical restrictions that make it a necessity for mobile applications. For cross-platform application the ability to interpret messages is key, so a standard-conformant and well-formed message format takes precedence over compactness and performance. Enterprise backend applications are more mixed, in that while the intranet typically provides abundant bandwidth, the large number of requests still requires good performance.

While an agent may have the option to open raw connection to other agents, bypassing the platform messaging service and supplying its own encoding and protocol, this is usually not advisable for the following reasons: On the one hand, developing an efficient transfer protocol involves a non-trivial amount of effort. It would therefore ease development effort if the agent message layer could be used. On the other hand, the agent system may be running within a restricted environment. For example, enterprise applications typically run on servers where the communication is tightly controlled for both support and security reasons. As a result, an agent may not be allowed to make connections outside what is provided by the agent platform.

Furthermore, there is another aspect concerning application development. In practice, there is often a distinction between the development phase of an application and production use in a business. For example, during development, applications often include additional logging and debug code to identify faults, include the use of assertions to validate program invariants and use tests to validate functionality. During production use, these features are often omitted in favor of higher throughput or lower latency.

	Development	Production
Human Readability	high	low
Standard Conformance	medium	medium
Well-formed Structure	high	low
Editable	high	low
Performance	low	high
Compactness	low	high

Fig. 2. Importance of message format features during development and production use

Figure 2 demonstrates this difference between the two stages with regard to agent message formats. During development the ability to easily read and modify messages supports the developer in finding protocol errors and other implementation errors. In addition, a well-formed structure allows the use of validation tools to ensure message correctness; however, this changes during production use. Good encoding and decoding performance and message compactness aids both system throughput and latency. During production use, this takes precedence over issues like message readability, since the development has completed and it is no longer necessary for humans to read agent messages.

The next section will take a look at common agent message formats that have been traditionally used by multi-agent systems and show how well they support the proposed message format features. This will show that there is potential for improvements for both performance and compactness if other features are less of a concern.

3 Related Work

Over time, multi-agent system have used a variety of message formats. Early system used simple ad-hoc languages in string-based formats; however, this resulted in languages that were specific to the application and made it difficult for multi-agent systems to interact. As a result, languages were developed to allow interchange between agent applications and agent platforms. One early attempt at defining an agent language was the Knowledge Query and Manipulation Language (KQML) [2]. However, it was quickly recognized that a standard language is useful for allowing communication between different agent systems.

Accordingly, the Foundation of Intelligent Physical Agents (FIPA) proposed two standards, the FIPA Agent Communication Language (ACL) [3] for the message structure and a specific language for the message content called FIPA SL [4] with different levels of complexity reaching from FIPA SL0 to FIPA SL2, both of which are used in popular agent platforms such as Jade [5].

This distinction between structure and content is retained in later formats as well. For example, while the Jadex Agent Platform [6] only uses a single XML-based format called Jadex XML, it distinguishes between message and content encoding. However, it uses Jadex XML for encoding both the message and content. Since the bulk of the message for the types of applications being targeted tends to be the content, the focus of this paper will be content encoding.

Nevertheless, as demonstrated by Jadex XML, the same principles can be applied to message encoding as well.

	FIPA SL	Java Serialization	Jadex XML	Jadex Binary
Human Readability	=	-	+	-
Standard Conformance	+	=	=	-
Well-formed Structure	+	=	=	-
Editable	=	-	+	-
Performance	-	+	-	+
Compactness	-	=	-	+

Fig. 3. Feature support by different methods of agent content encoding

When considering the agent format features proposed in Section 2, it becomes clear that even though some features are well-supported, other features were not the focus for those formats (cf. Fig. 3). For example, FIPA SL as a text-based format is quite readable and editable by humans, provides a definition of a well-formed structure and is a standard for agent messaging with wide support for many agent platforms. Jadex XML on the other hand, while not being a widely-used standard, has a well-defined and openly accessible schema and allows a human user to easily read and edit agent messages. Nevetheless, neither compactness nor performance seem to be the focus for either language. This is likely due to compactness and performance being in conflict with other features. For example, a compact format tends to be hard for humans to read.

The Jade platform recognizes the need for compact messages with good performance in some applications. It supports these features by adding content objects to messages, instead of a string-based content, but discourages this approach for lack of standard conformance. Jade uses the Java language serialization feature[7] to encode such messages; however, while this approach is fairly compact and certainly providing good performance, it has multiple drawbacks. First, for a number of reasons listed in the specification it only supports classes that explicitely declare to implement a marker interface. While it is trivial to add the interface to classes, the source code of classes used in legacy applications may be unavailable. In addition, some useful built-in classes like BufferedImage do not implement this marker interface and there is no way to easily retrofit the serialization system to support this class. Furthermore, there appear to be compatibility issues, requiring a versioning convention using a marker field and carefully monitoring of the Java Virtual Machines used by the system. Finally, as we will show, the compactness of the serialization format can be further improved upon, especially without an additional compression cycle.

In the next section we will introduce an agent message format for Jadex called Jadex Binary which focuses solely on the compactness and performance features. This message format will be an alternative to the default Jadex XML used by Jadex, which can be used by application that have a strong need for those two features and do not require feature better supported by Jadex XML. We

will then evaluate this new format based on the performance and compactness features based on a comparison with other agent message formats.

4 Format Description

Since the primary goal of the Jadex Binary format is to emphasize the compactness and performance properties of the format, it uses binary instead of string-based encoding. The primary concern is the serialization of the objects representing the message, such as, but not exclusively, ontology objects. In addition, some techniques are employed to prefer the compact encoding of common cases of data over rare cases, providing some simple compression based on the meta-information available from the objects and typical use cases. The format is based on a set of techniques to encode primitive types which are then used to encode more complex data. The following subsections will start with the primitive types and then proceed to more complex types.

4.1 Variable-sized Integers

A key concept used in Jadex Binary are variable-sized integers. The goal is to encode unsigned integer values in a variable-sized format that encodes small values with less space than larger ones. The technique is based on the encoding technique of element IDs in the Extensible Binary Meta-Language (EBML)[8], which again is based on variable encoding scheme used for UTF-8[9].

Bytes	Format	Value Range
1	1#######	0 to 127
2	01###### ########	128 to 16511
3	001##### ######## ########	16512 to 2113663
4	0001#### ######## ######## ########	2113664 to 270549120

Fig. 4. Examples of variable-sized integers and their value ranges

A variable-sized integer is byte-aligned and consists of at least one byte (cf. Fig. 4). The number of zero bits starting from the highest-order bits before the first one-valued bit denotes the number of additional bytes called extensions that belong to this variable integer. The rest of the byte is then used to encode the highest-order bits of the integer value, the extensions then provide the lower order bits of the value. This value is then shifted by a constant equal to the end of the previous value range plus one. This technique of storing integer value uses less space to encode small values at the expense of additional space of high values.

The next part will describe how the format encodes boolean values which can be used in to extend variable integers to support negative values. Furthermore, variable integers are also heavily used as identifiers during string encoding.

4.2 Boolean Values

At first glance, encoding boolean values appears to be trivial since it only requires storing a single bit. However, a data stream that is not byte-aligned requires a considerable amount of processing to shift and pad bits during encoding and decoding, impacting the performance property of the format. As a result, a byte-aligned format is preferable. The Java language solves this issue by simply using a full byte to encode a single boolean value, however, this approach wastes almost 88% of the available space, which is incompatible with a compact language format.

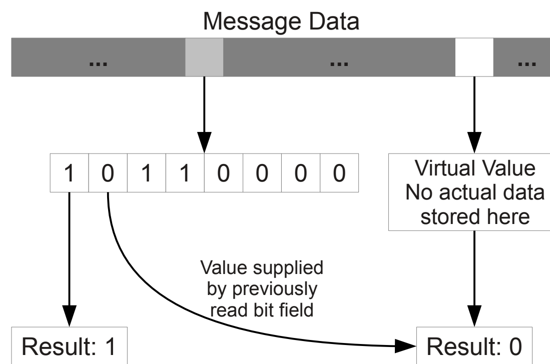

Fig. 5. Encoding of boolean values in the message: The first boolean value writes a byte-sized bit field that is reused by the next seven values

As a result, multiple boolean values are packed into a single bit. This is accomplished by writing a full byte where the first boolean value is written and updating that byte whenever additional boolean values are added (cf. Fig. 5). When the byte is filled with eight boolean values, another byte is written to the stream when the ninth boolean value is written. While the update cycles require some additional overhead on the part of the encoder, by having to update an earlier part of the byte stream, it reduces overhead for the decoder. During decoding, the byte is read when its first boolean value is read and then buffered for later reads.

This approach enables efficient storage of boolean values. This can be combined with the variable integer encoding to provide support for signed variable integers by writing a boolean sign flag before writing the absolute value as a variable integer.

4.3 Strings

Since string values tend to occupy a large part of typical messages, string encoding is a key part to ensure compactness. When a string is written by the

encoder, it is first checked if the string is already known. If not, the string is assigned a unique numerical ID and added to the set of known strings called the *string pool*.

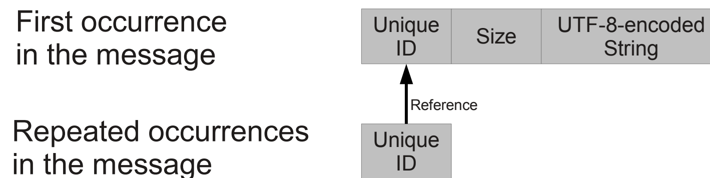

Fig. 6. When a string is occured for the first time in a message, it is encoded in full and assigned a unique ID allowing later occurences to be encoded by referencing the ID

The encoder then uses variable integer encoding to write the ID to the stream (cf. Fig 6). The string is then encoded using UTF-8 and its encoded size is written to the stream as a variable integer, followed by the encoded string itself. If the string is already known by the encoder, the encoder simply writes its ID as a variable integer, avoiding duplicate storage of the string.

Since the number of unique strings in a message is usually less than 128, a single byte is sufficient to encode any following occurence of a string using variable integer encoding. Furthermore, the size of strings tend to be short, generally less than 16511 bytes or even 127 bytes, especially if few characters are used outside the first 128 unicode characters is used, allowing the string size to be encoded in one or two bytes.

All strings share the same string pool, whether it is used to encode an actual string value of the object or if it is used for other internal purposes such as type encoding. This maximizes the chance of finding duplicate strings in the pool, reducing message size.

4.4 Other Primitives

Other primitive values consist of integer and floating point types byte, short, int, long, float and double. All of these values are simply translated into network byte order[10] and added to the message. The only exception are 32-bit integer values. In many cases, these values are used as a kind of default integer type. For example, the Java language treats all untyped integer literals as this type. This leads to a disproportionately large set of 32-bit integer values to consist of mostly small numbers.

As a result, we found it to be advantageous with regard to the compactness property to encode 32-bit int values as variable-sized integers in the message data. While this can lead to large values exceeding the 4 bytes occupied by a 32-bit integer value, for common values the size is actually lower than 4 bytes, providing an overall net advantage in terms of size.

4.5 Complex Objects

Complex objects are needed to encode messages containing objects derived from ontologies such as concepts and their relations. It is also needed to encode sub-objects such as attributes. Aside from certain special cases which are discussed in the following subsections, complex objects generally have a type or class and contain a number of fields that can either be primitives or other complex objects. For this reason they can be traversed recursively, encoding each sub-object it contains as a complex object. The encoder only needs to keep track of objects that have already been encountered in order to avoid reference loops (Object A containing Object B containing Object A) and encode object references instead.

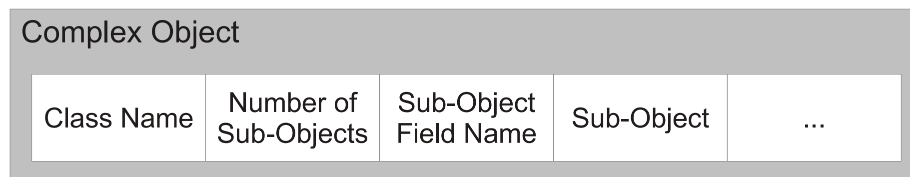

Fig. 7. A complex object is encoded using its class name, the number of sub-objects and pairs of field names and encoded sub-objects

As a result, the format for complex objects can be straightforward (cf. Fig 7). It starts with the fully-qualified class name, defining the type of the object. This is written to the message data using the string writing technique described in Section 4.3. Since some sub-objects may not be defined (i.e. reference null), not all of the sub-objects need to be encoded. Therefore, the class name is followed by the number of encoded sub-objects. This number is written to the message data as a variable integer as described in Section 4.1. Then the sub-objects are written by first writing the name of the field in the object containing the sub-object, then recursively encoding the sub-object itself. During decoding, the decoder first reads the class name and instantiates an object of that class. It then reads the number of sub-objects and finishes by decoding the sub-objects themselves, adding them to the object fields using the appropriate accessor methods.

Generally, it is expected that the objects offer accessor methods as described in the Java Bean specification and the encoder will only encode fields for which such accessor methods are available. However, using annotations, a class may declare that the encoder should encode the field regardless of the existence of accessor methods. In this case the fields are accessed directly using the Java reflection API.

4.6 Arrays

Arrays are encoded in a manner similar to complex objects, starting with the array type and concatenating the number of array components and the compo-

nents themselves. However, since the array type already provides type information about the array components, this information can be used to reduce the amount of information that needs to be stored in the message. The array encoding therefore supports two modes, a raw mode which is used for primitive type values and a complex mode for objects. In raw mode, the components are just written without additional information about the type of each component. This can only be done in case of primitive types for two reasons: First, primitive types cannot be subclassed, thus the array type is always sufficient to derive the component type. Second, primitive values are passed by value and do not have a null reference that requires special encoding.

This is not the case for objects. The array type may only refer to a superclass of the component object and the component object may simply be a null reference. This means that type information about each array component needs to be included. However, in most cases it is safe to assume that the array type describes the type of the component. Therefore, each component is preceded by a boolean value indicating whether further component type information is stored or if the type information can be derived from the array type. Only if this flag indicates that type information is stored, for example when dealing with subclasses or null values, the flag is followed by the type information. Otherwise, the component object is directly appended after the boolean flag.

In the next section we will evaluate Jadex Binary in terms of the performance and compactness features and compare it to three other message formats. This was done with a number of tests in which the message formats were measured and evaluated.

5 Evaluation

In order to evaluate the performance and compactness feature of Jadex Binary, we conducted a series of tests. The experiments were conducted using an Intel i5 750 processor with four cores clocked at 2.67 GHz. The machine was supplied with 8 GiB of memory; however, the Java heap size was limited to 2 GiB. The Java environment used was the Oracle Java SE 6 Update 31 which was running on a current version of Gentoo Linux compiled for the x86-64 instruction set with an unpatched Linux 3.2.2 kernel.

The content formats used were FIPA SL, the built-in Java serialization, Jadex XML and Jadex Binary. The Jade agent platform 4.1.1 using the BeanOntology was used as a representative of Java SL encoding. A test class representing an agent action was used as data set to be encoded. The agent action sample contained a 514 byte string literal, a second string containing a randomized long value encoded as string, a single integer value, an array of 20 integer literals, an array of boolean values and finally an array of objects of the class itself to represent recursively contained sub-objects.

For the compactness tests, this array contained 100 further instances of the class, which itself had the field set to null. For the performance tests, the number of objects in that array was varied, starting at 10000 objects and increasing in

steps of 10000 up to 100000 objects. The compactness was measured by counting the number of bytes of the encoded content. If the encoded content was a string, it was converted to a byte representation using UTF-8 encoding (other encodings like UTF-16 would have been possible but would have resulted in worse results). For the performance, the time between start and end of the encoding cycle was measured, other tasks like encoder and object setup were not considered.

In the following, we will present the results of both the performance and the compactness features. While the test data is certainly artificial, we have tried to supply what we think is a good cross-section of possible data cases.

5.1 Performance

In order to avoid interference with lazy initialization procedures and the just-in-time compilation of the Java VM, all performance tests were run twice, with the results of the first test run being discarded. This allowed the Java environment to compile the code and the encoding framing to initialize constants during the first pass so that the real performance figures could be obtained during the second pass.

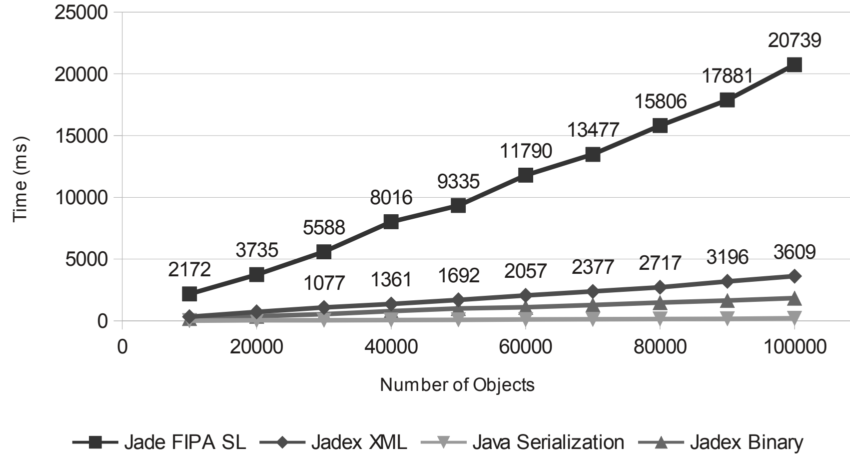

Fig. 8. Results for the performance measurements, Jade FIPA SL encoding requiring a disproportionally long time

The result of the tests can be seen in Figure 8. While all encoding time measurements seem to increase linearly with the number of objects, the FIPA SL encoding provided by Jade appears to require an unusually long time. As expected, both Jadex Binary and the Java serialization mechanism provide substantially better results; however, even Jadex XML which is not intended to be

optimized for this format feature still offers substantially lower encoding times than Jade FIPA SL encoding.

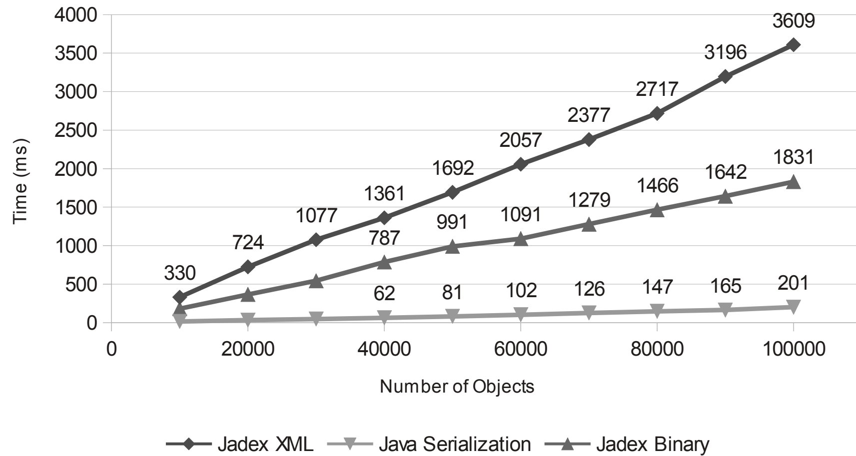

Fig. 9. Performance results using the same data set as Fig. 8 with Jade FIPA SL encoding excluded

Figure 9 provides a closer look at the three highest performing formats. While Jadex Binary clearly provides an advantage over Jadex XML by roughly a factor of two, the Java serialization is almost an order of magnitude faster. Initial analysis seems to suggest this is due to the use of the Java Reflection API used by both Jadex XML and Jadex Binary, which the Java serialization mechanism can avoid due to its built-in nature. However, Java serialization has further drawbacks as outlined in Section 3, meaning it is not a general solution to the problem and has a more narrow scope of environments in which it can be useful.

5.2 Compactness

In order to test for content compactness, the test object was passed to the encoder and the number of bytes of the encoded object was measured. Since the test object contained a fair amount of test data, the resulting content sizes were expected to be large.

As can be seen in Figure 10, the differences between the four formats are quite substantial. Jadex XML is barely half the size of the FIPA SL encoding and both Java serialization and Jadex Binary are substantially smaller still. In fact, Jadex Binary clearly provided the most compact representation of the test object, being smaller than even the Java serialization format by a factor of roughly 2.5.

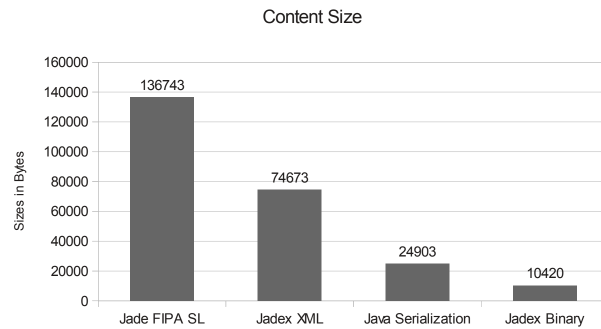

Fig. 10. Content sizes in bytes after encoding the object using the message format without compression

A fair amount of this is likely to be due to redundant information, especially of string values, which Jadex Binary can exploit (though Jadex XML uses a similar mechanism). In addition, text-based formats like FIPA SL and Jadex XML use a large amount of redundant strings to represent their formatting, such as tags in the case of XML.

In order to test this assumption, another set of tests was performed, which were identical to the previous tests but added an additional compression pass, converting it to the gzip-format, which uses the DEFLATE algorithm[11] to reduce data redundancy.

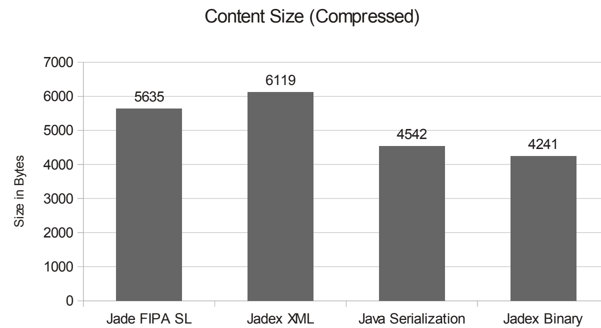

Fig. 11. Content sizes in bytes after encoding the object and further compressing the format with gzip

The results shown in Figure 11 substantiate the assumption. The DEFLATE algorithm drastically reduced the redundancies in both FIPA SL and Jadex XML with FIPA SL now even coming out ahead of Jadex XML. Nevertheless, both

Java serialization and Jadex Binary still show an advantage in compactness with Jadex Binary maintaining a slim margin over Java serialization.

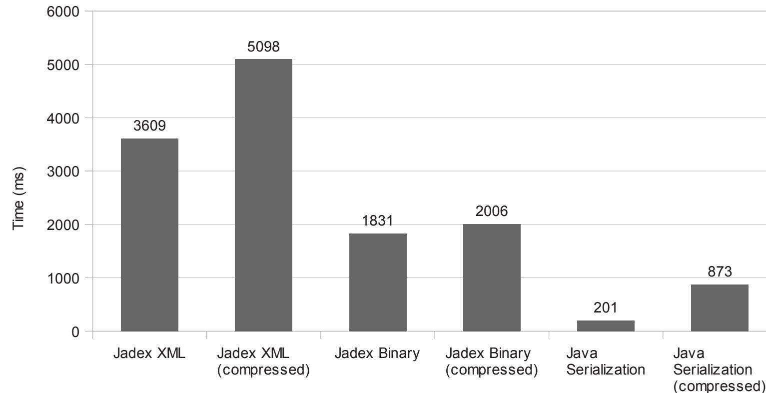

Fig. 12. An additional compression pass increases total encoding time

Since the compression pass substantially reduces the size of messages, especially for verbose formats, it may suggest that starting out with a compact format gives only a marginal advantage. However, data compression is not free in terms of computation time. While compression helps compactness, this issue has to be weighed against the performance message feature. Despite the DEFLATE algorithm being a comparably fast compression algorithm, Figure 12 shows that it adds a substantial amount to the total encoding time of the content. In fact, the additional time required seems to grow with the number of bytes in the uncompressed content, which is reasonable considering the algorithm must evaluate every byte of the uncompressed data at least once to produce a reversible output.

As a result, data compression does not appear to be generally beneficial when both performance and compactness are important; however, it is another useful tool to adjust the balance between the two language features. In the next section we will discuss further improvements, future work and provide a conclusion on the performance of Jadex Binary.

6 Future Work and Conclusion

The evaluation of Jadex Binary in Section 5 appears to provide sufficient evidence that Jadex Binary already has significant advantages in both compactness and performance. However, the performance results of the Java serialization shows that further performance improvements may be possible. One way of further

reducing the overhead of Jadex Binary is to reduce the use of the Java Reflection API to access complex objects. This could be accomplished by injecting bytecode-engineered delegate classes which use direct method calls to retrieve and set bean properties.

In addition, the encoder and decoder of Jadex Binary are largely independent of the Jadex platform. It would therefore be possible to include the message format in other agent platforms, thereby allowing them to offer an alternative compact message format for agent communications for certain types of applications.

Overall, Jadex Binary is both able to represent agent messages in a compact form and perform in a reasonably fast manner. Since these two features were the primary goal of Jadex Binary, it does so by sacrificing others like human readability. Nevertheless, if those features are important, other established languages already provide sufficient support. The addition of Jadex Binary allows a developer of a multi-agent system to pick the kind of format that provides the best match for the requirements of a specific application and switch the format depending on state of the application in the development cycle.

References

1. S. Willmott, J. Dale, B. Burg, P. Charlton, and P. O'Brien, "Agentcities: A Worldwide Open Agent Network," *Agentlink News*, vol. 8, November 2001.
2. T. Finin, J. Weber, G. Wiederhold, M. Genesereth, D. McKay, R. Fritzson, S. Shapiro, R. Pelavin, and J. McGuire, "Specification of the KQML agent-communication language – plus example agent policies and architectures," Tech. Rep. EIT TR 92-04, 1993.
3. *FIPA ACL Message Structure Specification*, Foundation for Intelligent Physical Agents (FIPA), Dec. 2002, document no. FIPA00061. [Online]. Available: http://www.fipa.org
4. *FIPA SL Content Language Specification*, Foundation for Intelligent Physical Agents (FIPA), Dec. 2002, document no. FIPA00008. [Online]. Available: http://www.fipa.org
5. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, "JADE - A Java Agent Development Framework," in *Multi-Agent Programming: Languages, Platforms and Applications*, R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds. Springer, 2005, pp. 125–147.
6. A. Pokahr and L. Braubach, "From a research to an industrial-strength agent platform: Jadex V2," in *Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009)*, H.-G. F. Hans Robert Hansen, Dimitris Karagiannis, Ed. Österreichische Computer Gesellschaft, 2 2009, pp. 769–778.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Third Second Edition*. Addison-Wesley, 2005.
8. C. Wiesner, S. Lhomme, and J. Cannon, "Extensible Binary Meta-Language (EBML)," Website, http://ebml.sourceforge.net/, 2012.
9. The Unicode Consortium, *The Unicode Standard*. Addison Wesley, 2006.

10. P. Hoffman and F. Yergeau, “UTF-16, an encoding of ISO 10646,” RFC 2781, Internet Engineering Task Force, 2 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2781.txt
11. L. P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” RFC 1951, Internet Engineering Task Force, 5 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt