MultiScience - XXX. microCAD International Multidisciplinary Scientific Conference
University of Miskolc, Hungary, 21-22 April 2016, ISBN 978-963-358-113-1

OPTIMIZING COMMUNICATION OF REST SERVICES ON THE .NET PLATFORM

Akos Nagy[1], Viktor Czegledi[2], Bence Kovari[3]
Ph.d. Student, M.Sc. Student, Associate Professor
Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

ABSTRACT

In this paper we discuss optimization possibilities in the communication of Representational State Transfer (REST) services. After the turn of the 20th century REST services became popular in the industry. This architectural style has a number of advantages, including its simplicity compared to traditional Web Services, its platform-independent nature and also the efficiency of communication. Though the proposed specification itself does not limit the data exchange format, JSON (Java Script Object Notation) is a widely agreed upon choice. This further improves efficiency and interoperability. While definitely more compact than XML, JSON is still quite verbose and time-consuming to process and parse, especially when platform-independency is not a strong criterion. This paper builds heavily on our previous results published in [1] as we aim to optimize even more the communication process of distributed services. In the rest of the paper we discuss the improvements we made to the original solution. To show the improvements, a case study was also implemented, which shows that a 58% improvement and a 43% percent improvement can be achieved in output length and speed, respectively.

INTRODUCTION

After the turn of the 20th century, the decreasing cost of bandwidth and the general availability of smart mobile devices led to a sudden increase in the number of distributed applications. While the traditional Web services architecture could provide the necessary technical tools the build this new generation of applications, a number of challenges did arise.

One of these was (and still is to a degree) the limited power supply capacity of the mobile devices. In order to spare this valuable resource an efficient data exchange format is needed, so the time of the communication becomes less [2]. Like XML, JSON also has the benefit of being platform-independent, but it is much more compact, making it an ideal choice.

Another challenge was (and still is) that applications need to be up and running in a very short period of time. As building the communication between client and server is a tedious task, if we can improve on this, we can decrease the time-to-market of the conceived product. While Web services build also heavily on the existing HTTP protocol stack, it still requires a number of customizations that take up a lot of the developers' time. The REST [3] architecture offers the same advantages as traditional Web services, but they can be implemented more easily and faster.

DOI: 10.26649/musci.2016.061

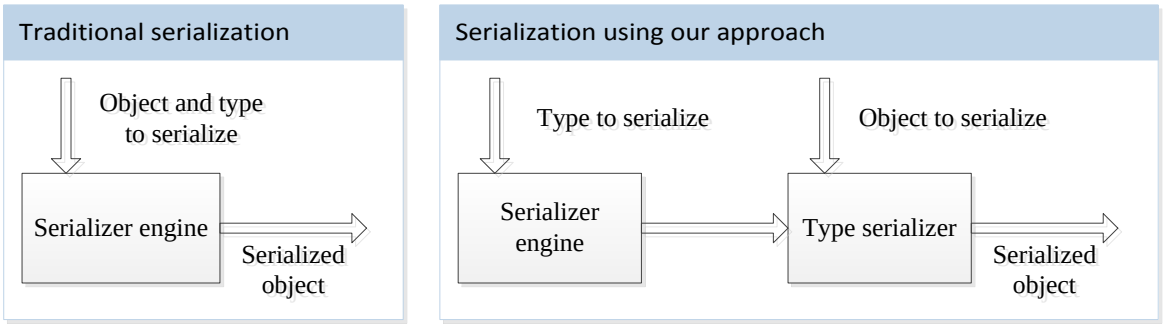
To further aid developers, implementing REST services is strongly supported in most of the popular programming frameworks by libraries like the Web API in the .NET framework. Our goal is to present an approach to optimize communication in REST services when all communicating parties run software built with this Web API, thus making platform-independency an unnecessary constraint.

RELATED WORK

The work presented in this paper builds heavily upon our previously designed serialization algorithm and module published in [1]

Our approach includes a new way of object serialization and a new, binary data exchange format coupled to this serialization method. Traditionally, serializer engines handle an object as a whole, meaning upon receiving an argument as input for serialization, they discover the structure of the type (its data members and their types) so they know how to serialize it and then write the contents of the object based on this information to the output stream. The solution proposed by us handles the type of the input object and the content as separate parts. First, the type of the object is discovered and then a new serializer module is generated on-the fly, which has these information built-in. Using this dynamically generated module, the contents of the object is written to the output stream. As this module incorporates all the information collected during the discovery phase, it has the benefit of being nearly as fast as a design-time written, type-specific serializer. The module itself is also cached, so no two types are discovered twice, saving even more time.

Figure 1
Differences of the traditional and new approaches



We also proposed an efficient binary protocol as the data exchange format. The originally proposed version in Extended Backus-Naur form is as follows (notice the addition of the object id in the case of complex types):

```
<serialized_stream>::=<types><serialized_graph>
<types>::=<typelist_length>{<type_name>}
<serialized_graph>::=<serialized_object>|{<serialized_object>}
<serialized_object::={<serialized_primitive>|<serialized_dict>|<serialized_list>|
                      <serialized_complex>}
<serialized_primitive>::=<type_id><object_value>
<serialized_dict>::=<type_id><dictlength>{<serialized_keyvaluepair>}
<serialized_keyvaluepair>::=<serialized_key><seralized_value>
<serialized_key>::=<serialized_object>
<serialized_value>::=<serialized_object>
<serialized_list>::=<type_id><listlength>{<serialized_object>}
<serialized_complex>::=<type_id>{<serialized_property>}{<serialized_field>}
<serialized_property>::=<serialized_object>
<serialized_field>::=<serialized_object>
```

Our original measurements showed that service time call could be improved by 23% and data can be compacted by 42%, but the algorithm and the implemented solution had a number of shortcomings. In the next section, we the improvements we made to overcome these.

CONTRIBUTIONS

A notable issue with the original solution was that it could only handle object trees. Object graphs that included an actual cycle caused an infinite loop never terminating the algorithm.

To solve this problem, we introduced the capability of serialization by reference. This means that when the object graph is traversed, every unique object is assigned a unique object identifier. When the object is encountered for the first time, this unique identifier is generated and it is written to output stream along with the contents. When the object is encountered again, instead of the contents only this object identifier is written to the output stream and the traversation moves on to the next object. To support this functionality we amended the *serializerd_complex* expression as follows:

```
<serialized_complex>::=<type_id><object_id>{<serialized_property>}{<serialized_field>}
```

In case of complex types, the object identifier is added to the serialization output. In the case of primitive types this is not needed (they are either value types of immutable types, neither of which can be referenced more than once in a graph). The semantics of the *type_id* field is changed. Originally it is supposed to show the identifier for the type of the object. This value is by our definition greater than zero, so negative values can be used to represent special cases. If the *type_id* in the stream equals to -2, this shows that the serialized object only contains the object id and the contents should be resolved from the already deserialized objects based on that.

Generating the object identifier is a subtask that must be handled within the serialization process. This can be done an arbitrary function *G(o)*, as long as

G(o1)=G(o2) is true if and only if *o1=o2*. After inspecting the reference source code of the .NET framework, we discovered that there is a component that is used by the built-in serializers that perform reference based object serialization. We opted to use this solution (the internal details of this module are beyond the scope of the paper, for more details see [4] and [5]).

Extending the purpose of the *type_id* field allows for a number of necessary improvements. Introducing the value of -1 as the type identifier shows that the object was, in fact, null in the graph, thus allowing for the handling of null values.

Another improvement was the introduction of variable length zig-zag encoding to handle 2, 4 and 8 byte long integer values. This (or a version of this) is widely used in industry standard serializer modules such as Apache Avro [6] or ProtoBuff [7]. These algorithms allow for the representation of these longer values using less bytes if the values are small. Our algorithm operates with a number of numeric identifiers. These are chosen by design as 4 byte integers so the algorithm can handle even large scale of data, but they do tend to be smaller (e.g. an object graph with even a just a hundred different types of objects is quite hard to handle from a software engineering point of view, so the type identifiers tend to be smaller). For this reason, the output is even more compact.

One final improvement was the introduction of by reference parameter handling. Traditionally, most programming languages and frameworks support the concepts of passing a parameter by value or by reference. When passing a parameter by value, the whole parameter is copied into a new instance that the callee uses. When passing a parameter by reference, only the memory address of the parameter is used, so no copying is involved, thus making the process faster. Also in managed frameworks this makes the memory cleanup process faster. In case of reference types, this is the working semantic by nature, but for value types it must be explicitly specified. As an improvement, we redesigned the dynamically generated code so that value type parameters are passed by reference also.

CASE STUDY

To measure the efficiency of the previous approach, a benchmark was created with the following setup.

A simple REST service was implemented using the .NET Web API to test end-to-end service call time and the length of the exchanged data. The service operation is passed a given number of objects and sends back an empty response. To compensate for initial inaccuracies, the 20^{th} service call time was measured. The objects sent through the network are complex objects containing strings, numeric values (integers, doubles) and collections (lists and dictionaries). To analyze the performance of the serializer, the time of the service call and the length of the output are measured at a number of objects ranging from 1 to 25000 and then a linear function of the object number is interpolated for the calculation of the output length and the call time. The slopes of these linear functions are the performance indicators that we used.

To test how the reference based serialization works, we designed to scenarios. In one of the cases the list of serialized objects contains only distinct objects. In the other case all the references in the list points to the same 10 instance.

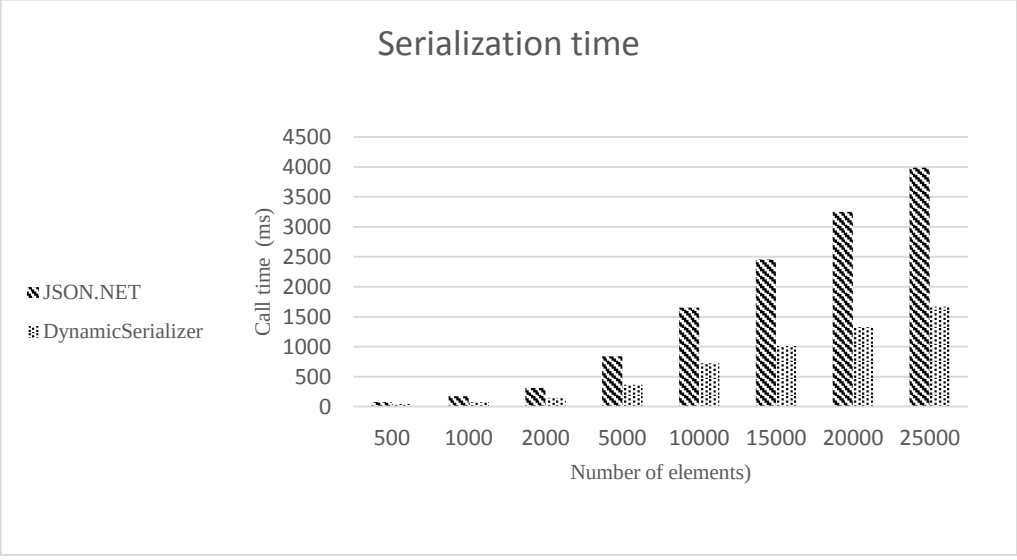
Table 1 shows these performance indicators calculated for the JSON.NET [8] – the default Web API serializer – and the DynamicSerializer, which is the implementation of the algorithm presented above for the first test scenario (with all different objects).

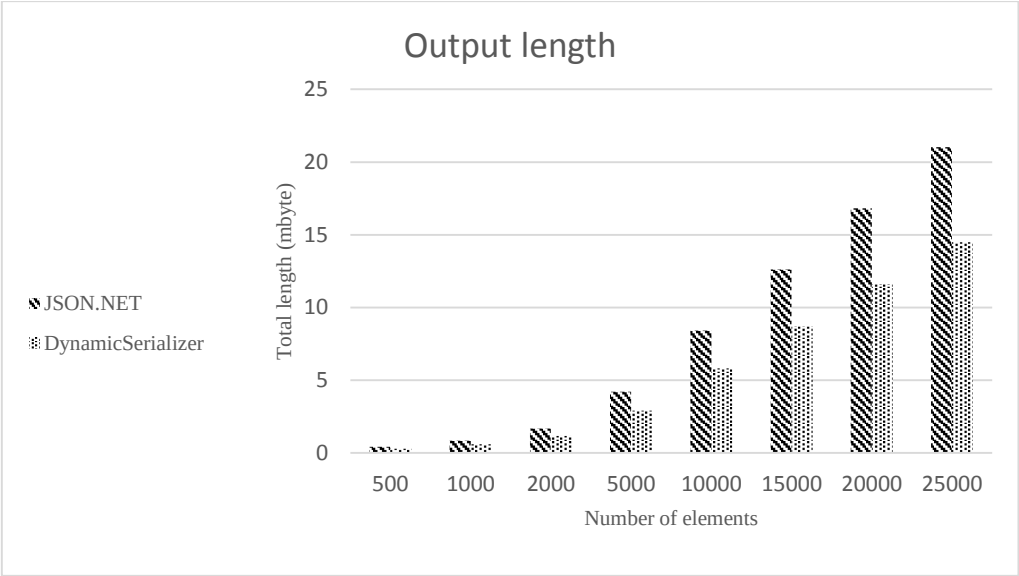
Table 1
Performance indicators of the serializers for different objects

	JSON.NET	DynamicSerializer
call time (ms/item)	0.159	0.066
output length (kbytes/item)	0.861	0.593

The results of Table 1 show that to serialize one instance of the type used, the JSON.NET serializer needs about 0.159 ms, while our proposed solution needs only 0.066, which is a 58% decrease. The output length shows that the JSON.NET generates 0.861 kbytes from one instance, while the DynamicSerializer uses only 0.593 kbytes, which is a 31% decrease. Figure 2 visualizes the differences of the call time the output length:

Figure 2:
Performance difference of the serializers for the all-different scenario





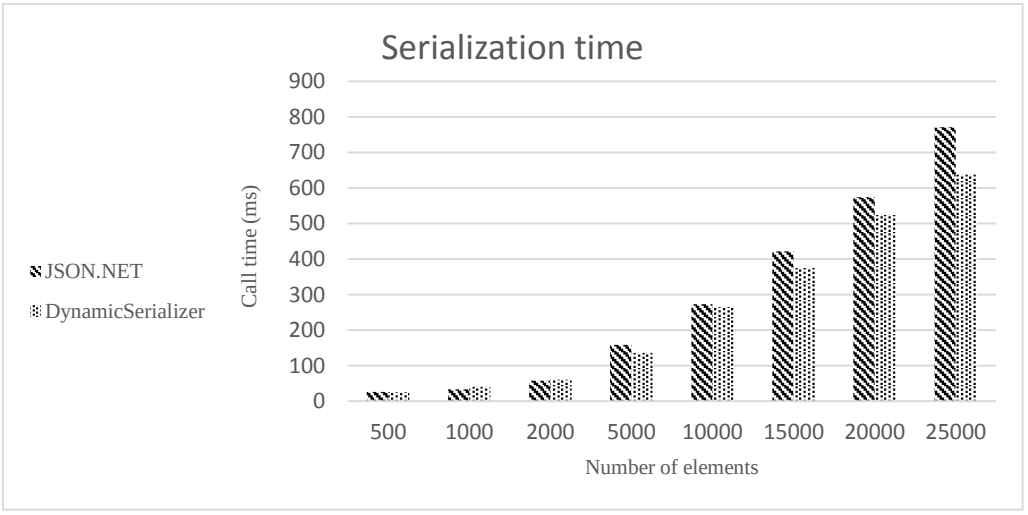
Since one of the key improvements of the algorithm is the by reference serialization capability, it was important to measure how the modules handles the duplicate references. Table 1 shows these performance indicators calculated for the JSON.NET – the default Web API serializer – and the DynamicSerializer, which is the implementation of the algorithm presented above for the first test scenario (with all different objects).

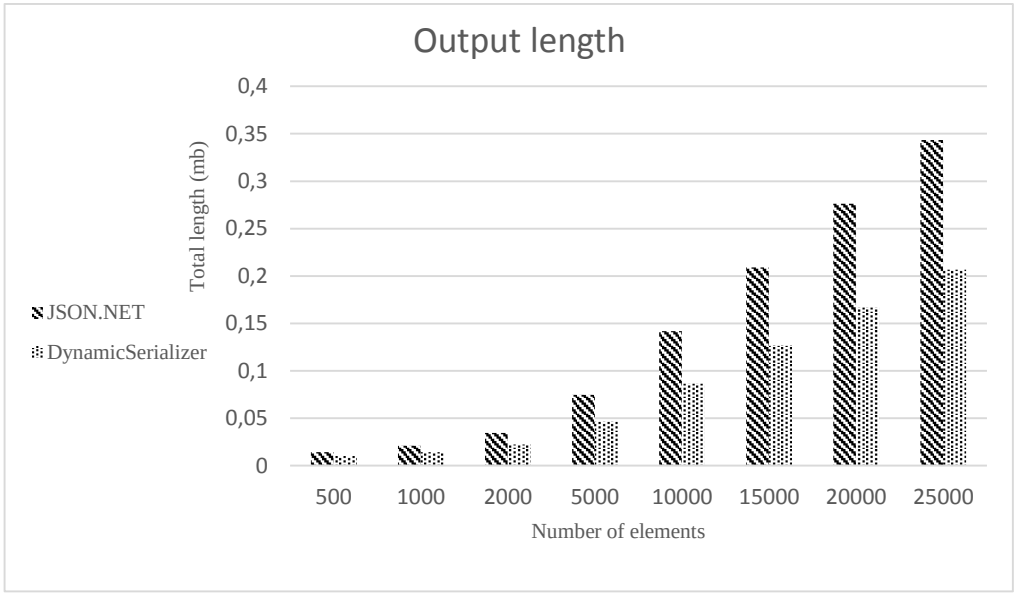
Table 2
Performance indicators of the serializers for duplicated object references

	JSON.NET	DynamicSerializer
call time (ms/item)	0.03	0.025
output length (kbytes/item)	0.014	0.008

The results of Table 1 show that to serialize one instance of the type used, the JSON.NET serializer needs about 0.03 ms, while our proposed solution needs only 0.013, which is a 17% decrease. The output length shows that the JSON.NET generates an average 0.014 kbytes from one instance, while the DynamicSerializer uses only 0.008 kbytes, which is a 43% decrease. Figure 3 visualizes the differences of the call time the output length:

Figure 3:
Performance difference of the serializers for the half-different scenario





CONCLUSIONS AND FUTURE WORK

The paper presented a novel approach to optimize the communication of SOAP-based web service applications. The approach contains an algorithm and a compact message format and an implementation of these using the Reflection.Emit .NET API. With a benchmark the efficiency of the serializer was demonstrated and concluded that the service call time can be improved by 52% and the output can be compacted by 31%.

Future work includes exploring the possibilities of preparing the serializer to process the object graph in a parallel fashion, thus improving further the efficiency.

Since serialization is a common task in applications, the proposed serializer module can be used in other scenarios. We plan to evaluate the efficiency under

different circumstances, for example compared to the persistence API provided in the Workflow Foundation library.

ACKNOWLEDGMENTS

This work was partially supported by the TÁMOP-4.2.1.D-15/1/KONV-2015-0008 project.

REFERENCES

[1] A. Nagy, B. Kovari, „Optimizing communcation of web services on the .NET platform," in *microCAD 2013: XXVII. International Scientific Conference*, Miskolc, 2013.

[2] K. Devaram, D. Andresen, „SOAP Optimization via Paramterized Client-Side Caching".

[3] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures," 2000.

[4] „.NET Reference Source - ObjectIdGenerator," [Online]. Available: http://referencesource.microsoft.com/#mscorlib/system/runtime/serialization/objectidgenerator.cs,b9dd7357a53b8a9c. [Accessed: 4 january 2016].

[5] „.NET Reference Source - BinaryFormatter," [Online]. Available: http://referencesource.microsoft.com/#mscorlib/system/runtime/serialization/formatters/binary/binaryformatter.cs,c19170bda56bc560. [Accessed: 4 january 2016].

[6] „Apache Avro Specification," [Online]. Available: https://avro.apache.org/docs/1.7.7/spec.html. [Accessed: 4 january 2016].

[7] „Protocol Buffers," [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding. [Accessed: 4 january 2016].

[8] „JSON.NET Home," [Online]. Available: http://www.newtonsoft.com/json. [Accessed: 4 january 2016].