

OPTIMIZING COMMUNICATION OF WEB SERVICES ON THE .NET PLATFORM

Ákos Nagy¹, Bence Kővári²

B.Sc. Student, Assistant Lecturer

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 process of web service applications built following the tenets of the Service Oriented Architecture (SOA), when both the client and the host run software built using the .NET platform. Web services use Simple Object Access Protocol (SOAP), an XML-based format to transmit data, which – though has a lot of benefits – suffers from being too verbose and time-consuming to process and parse, especially in the case when platform-independency is unnecessary. This paper first presents different approaches trying to optimize the communication and then discusses the new way developed by us, which has the potential to reduce communication time by 23% and the data produced by 42%. To show these improvements, a case study using a typical enterprise data structure is also implemented and analyzed.

INTRODUCTION

As the IT industry developed, the decreasing cost of bandwidth and computational capacity made the emergence of distributed architectures possible. The core premise of these architectures is very simple: multiple machines work separately to achieve a common goal and they exchange messages using a defined protocol to coordinate the workflow. One such architecture is the Service Oriented Architecture (SOA), which is a widely used architectural paradigm for building web service applications. Web services use the XML-based, platform-independent Simple Object Access Protocol (SOAP) message format. Web service applications must use this protocol and transform in-memory data into this format (known as serialization) before sending, and parse this format after receiving and create a data structure with the corresponding content (known as deserialization). These capabilities are usually implemented in a framework like the Windows Communication Foundation library, which is part of the .NET Framework.

Being an XML-based format, SOAP is verbose and parsing it is a complex task, causing a performance hit in applications. These disadvantages become even more apparent when all the nodes participating in the workflow run software on the same platform, thus making platform-independency unnecessary. The goal of this paper is to present an approach for optimizing communication in the case when all nodes run software developed using the .NET Framework, while respecting the tenets of SOA itself and keeping the SOAP format.

RELATED WORK

When designing SOAP, high performance was not a goal. Since its introduction in 1998 and after becoming a W3C recommendation in 2003, many studies have been carried out to analyze its performance and point out its disadvantages, such as [1] and [2].

Other related works include studies proposing methods to optimize the communication. One approach is proposed in the bSOAP framework in [3] and [4]. This implementation uses differential serialization and deserialization to improve performance, meaning that the framework keeps track of previous messages and when serializing an object only the differences have to be processed. The same goes for deserialization: the received messages can be compared to recorded ones and the previously instantiated objects can be reused when creating new ones. This can be especially efficient on the server side, when the host receives a lot of similar messages from different clients, such as in the case of a search engine, where only the search parameters differ. A similar approach is described in [5].

Differential serialization in effect is a method of caching. Another optimization proposal, described in [6] is also based on caching: on the client side, messages are recorded for later use. This is efficient when the client has a low number of definite requests, such as stock applications.

Besides caching, another approach for boosting SOAP-performance is compression. Compression-based optimization methods are discussed for example in [7], where significant improvements are achieved in the case of mobile devices.

The methods presented above operate directly with the messages. A more abstract approach is presented in [8], where the authors suggest WSDL-extensions into the architecture of web services, to help negotiate message exchange – this architecture is especially promising in the case of large documents.

While these methods try to exploit caching, compression or introduce architectural modifications, our method proposes a compact output format to be embedded into SOAP envelopes and also tries to leverage the possibilities lying in code generation to optimize the communication in the scenario, when all communicating nodes run .NET software.

CONTRIBUTIONS

In the following section we present a compact binary protocol, an algorithm to handle this protocol and an efficient implementation of this algorithm to optimize the communication in the proposed scenario.

Every piece of data that is sent is a .NET object, serialized and wrapped into a SOAP envelope, so the data can be represented as an object graph. Every complex object (i.e. an object that has members defined) is a node and every object of a simple type (i.e. a primitive type of the C# language with the exception of object and dynamic) is a leaf of this graph.

When sending a message, the data to be contained in the message has to be produced, which is the serialized version of the object graph representing the data. The algorithm proposed traverses the graph and examines each object. During

traversal, the types of the complex graph nodes are collected in a list. The index of a given type in this list serves as a unique identifier for a type for the current service operation call. In addition to collecting the types into a list, the algorithm also serializes the objects into a stream and marks each complex object with its identifier. In the proposed scenario both the client and the host can use the same assemblies containing the data classes used in the communication, rendering the need to write full type definition metadata into the stream unnecessary. In the case of simple types even the type identifier can be omitted, because these types are either value types or sealed reference types, so the design time and runtime types are equivalent without exception.

With these in mind, the output format of a serialized object graph is described with Extended Backus-Naur form as follows:

```

<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><serialized_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>

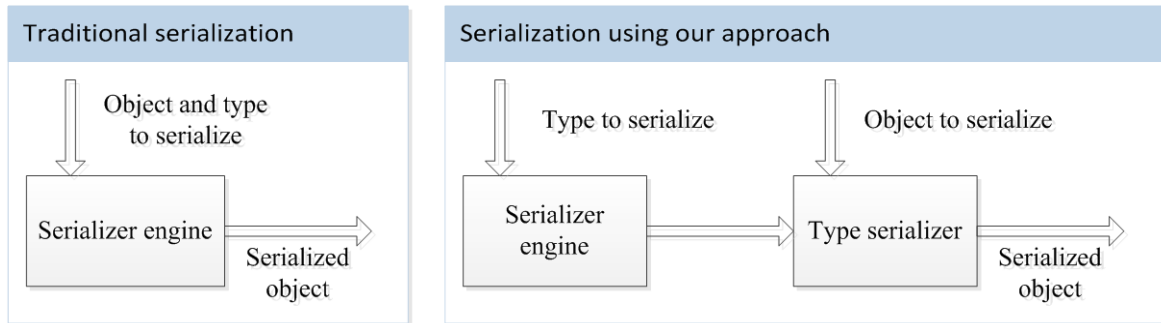
```

When a length is written into the stream, the following element is repeated exactly the number of times that length denotes. After an object graph is transformed into this format, it is encoded using Base64 format and embedded into a SOAP envelope.

To generate this format, the algorithm described above is implemented using the Reflection.Emit API. It allows programmers to build assemblies and types at runtime and incorporate them into their programs.

The task of serialization is broken down into two parts: the task of serializing a type and the task of serializing an object, a concrete instance of the type. First, when an algorithm encounters an object in the graph, it determines its type. If it has no knowledge of that type, it applies the algorithm presented above, but instead of writing that concrete value (item count or object value) into a binary stream, it emits code to an assembly to write that value into a binary stream. So at the end, the algorithm produces a fast and compact serializer which already has explicit knowledge about a given type and can serialize instances of this type efficiently. This process must be repeated for each complex type in the graph. Figure 1 shows the difference between the traditional approach of serialization and this method (the process of deserialization is similar):

Figure 1
Differences of the traditional and new approaches



The reason for the performance increase is two-fold: on one hand, after building the type specific serializer objects can be processed faster because the explicit knowledge of the type makes unnecessary to use reflection on the type each time. On the other hand, building the type specific serializer has to be done only once for each type in the graph.

CASE STUDY

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

A simple WCF service was used to test end-to-end service call time. The length of the output was measured separately, without the SOAP Envelope (this would only add a constant to all cases). The service operation is passed a given number of objects and sends back an empty response. To compensate for initial inaccuracies, the 20th 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 50000 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.

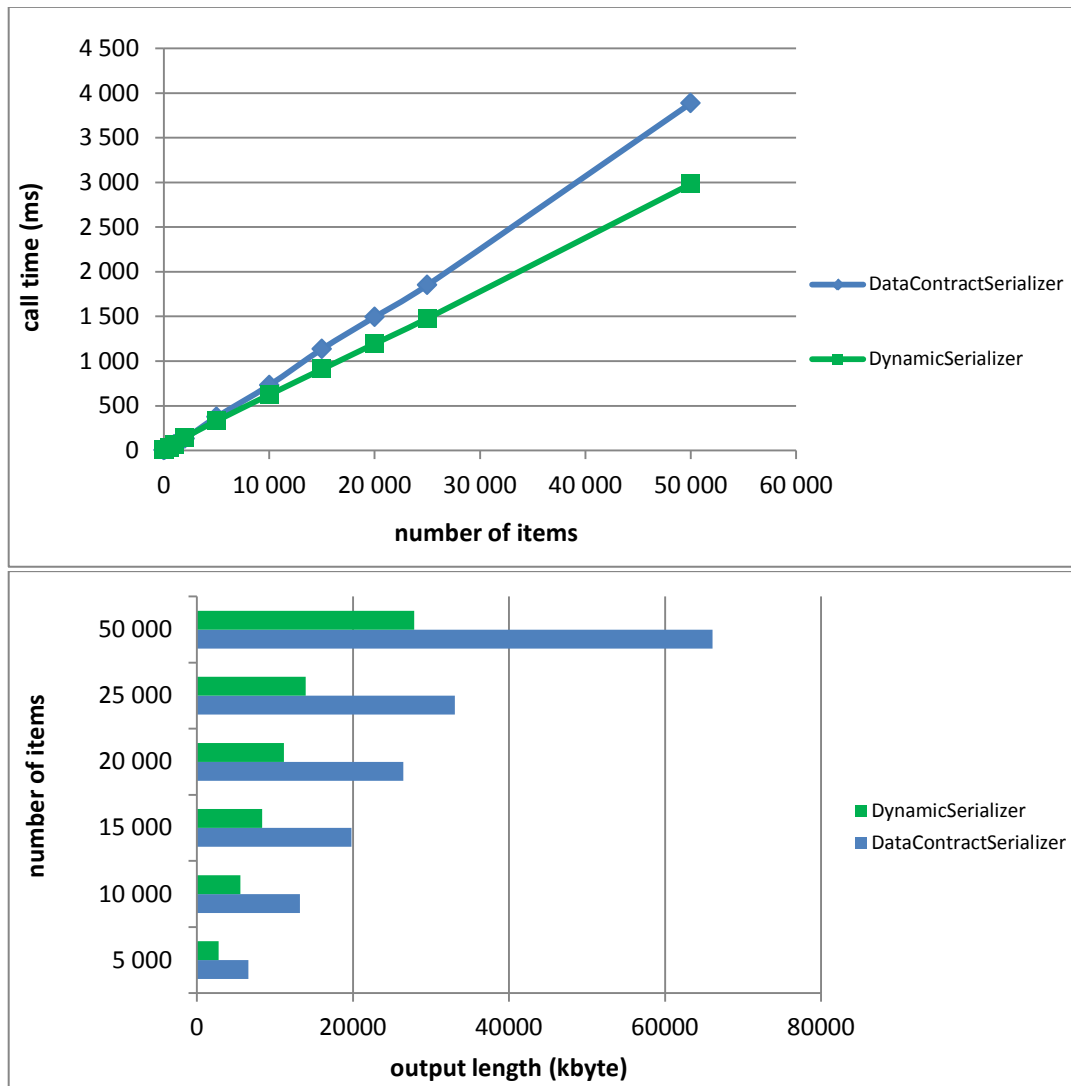
Table 1 shows these performance indicators calculated for the `DataContractSerializer` – the default WCF SOAP serializer – and the `DynamicSerializer`, which is the implementation of the algorithm presented above.

Table 1
Performance indicators of the serializers

	<code>DataContractSerializer</code>	<code>DynamicSerializer</code>
call time (ms/item)	0.076	0.059
output length (kbytes/item)	1.322	0.557

The results of Table 1 show that to serialize one instance of the type used, the DataContractSerializer needs about 0.076 ms, while the serializer implemented using the Reflection.Emit API needs only 0.059 ms, which is a 23% decrease. The output length shows that the DataContractSerializer generates 1.322 kbytes from one instance, while the DynamicSerializer uses only 0.557 kbytes, which is a 42% decrease. Figure 2 visualizes the differences of the call time the output length:

Figure 2:
Performance difference of the serializers



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 23% and the output can be compacted by 42%.

Future work includes exploring the possibilities of preparing the serializer to handle different versions of the same type (missing or newly introduced members), and handling circular references. The use of different compression methods on the message is also in plans.

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.

REFERENCES

- [1] R. Elfwing and U. Paulsson, "Performance of SOAP in Web Service," *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pp. 84-93, 2002.
- [2] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems," 2003.
- [3] N. Abu-Ghazaleh, M. J. Lewis and M. Govindaraju, "Differential serialization for optimized SOAP performance," *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pp. 55-64, 2004.
- [4] N. Abu-Ghazaleh and M. J. Lewis, "Differential Deserialization for Optimized SOAP Performance," *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, p. 21, 2005.
- [5] T. Suzumura, T. Takase és M. Tatsubori, „Optimizing Web Services Performance by Differential Deserialization,” *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pp. 185-192, 2005.
- [6] K. Devaram és D. Andresen, „SOAP Optimization via Paramterized Client-Side Caching”.
- [7] T. Min, T. Voigt, T. Naumowicz, H. Ritter and J. Schiller, "Performance considerations for mobile web services," *Computer Communications Journal*, vol. 27, no. 11, pp. 1097-1105, 2004.
- [8] S. Amer-Yahia and Y. Kotidis, "A Web-Services Architecture for Efficient XML Data Exchange," *Data Engineering, 2004. Proceedings. 20th International Conference on*, pp. 523-534, 2004.