Latencies of Service Invocation and Processing of the REST and SOAP Web Service Interfaces

Tommi Aihkisalo, Tuomas Paaso
VTT Technical Research Centre of Finland
Kaitovayla 1, Oulu FI-90571, Finland
{tommi.aihkisalo, tuomas.paaso}@vtt.fi

***Abstract*—This paper studies the latencies experienced by the Web service client invoking a proprietary multimedia messaging service with both REST and SOAP interfaces implemented. This study has been made utilising XML, JSON, MTOM/XOP and Google Protostuff message and message content encodings where available. We study the overall round trip time from the start of the request to the response returning to the client. Furthermore, we measure and analyse the phases the request travels through to the server and the return of an associated response. The results of the overall round trip time and distribution of the delays along the service invocation path is presented to ultimately aid and justify a selection of the better performing interface implementation and associated message encoding scheme for the solution. Both interfaces implement the same functionality of registering a client, sending and receiving a message and finally unregistering. Therefore, the results are highly comparable which show differences not only between REST and SOAP but also between XML and JSON and how it can be further improved.**

I. Introduction

This paper compares the Web service invocation latencies and their causes experienced by the client during the service request-response round trip. Comparison is made between the two distinctively different Web service interface styles and further with different kinds of request serialisation methods deemed suitable for each message content. The evidence presented here confirms the anticipated belief that REST is many times faster than SOAP on the tested setup, but it also shows how some SOAP binary content optimisation mechanisms are not as beneficial as previously thought. We also demonstrate how the performance of the REST can be improved significantly with an alternative message wire format choice. The overall latencies experienced with Web services culminate in three related factors. The first is the processing delays on the request and response marshalling and unmarshalling tasks on the client and server, the second is the latency of the message and marshalled request/response transfer over the network, and the last is the performance of the underlying service logic on the server. This paper shows that a bad choice of request/response marshalling scheme might ruin the service invocation throughput on otherwise positive solutions. The tests that were run for this study consisted of the exchange of the application messages containing a lengthy array structure and a message with a binary content element. The tested object marshalling schemes were standard XML, JSON and Google Protostuff, the encoding engine of the Protocol Buffers [1] and SOAP XOP/MTOM [2] and [3]. The tests were run with a proprietary multimedia domain specific messaging service solution providing REST and SOAP interfaces with Jersey and Axis2 Web service frameworks respectively on the Apache Tomcat.

In the field of Web services, which has a large collection of WS-* standards, the easiest way to implement Web services is REST [4]. REST defines an architectural style of the Web, and to which HTTP tightly adheres, introducing alternative simpler and more Web-friendly or Web-like ways of designing and implementing Web services. Where SOAP offers complex service contracts, tightly specified data structures and APIs manifested in a service specific WSDL files, REST relaxes it by simplifying API to a fixed set of operations consisting of the basic HTTP verbs including GET, PUT, POST, etc. SOAP builds several layers on the top of HTTP while REST directly uses its possibilities and functionalities. A lack of description constructs similar to a WSDL is often seen as a disadvantage. Therefore, WADL [5], more recently Unified Service Description Language, USDL [6], has been introduced mainly to be used as a resource representation format declaration. Claims have been made that a need for separate description mechanisms can be avoided by using the Web's inherent OPTIONS HTTP verb to discover the allowed operations for each resource following the HATEOAS principle, providing means of navigation in the resources. However, it must be noted that while REST is deemed a simpler, more lightweight solution and therefore better performing, REST might not be able to provide an adequate service solution for operationally and functionally very complex systems. The contributors towards the general performance and complexity of the service invocation are not only the architectural style but also the resulting request sizes resulting from the used message encoding scheme. A completely separate question is the paradigm change from the RPC-like thinking of SOAP to resource-oriented design and engineering. Converting legacy RPC-like systems to REST might prove difficult, but starting from scratch may be easier due to different kinds of messaging needs and communication patterns.

This study consists of a set of measurements and result comparisons of the round trip time and intermediate delays on the Web service request-response cycle. The service client and the messaging service has an intended use in the multimedia domain, especially for embedded and computing resource-constrained client mobile devices. This domain sets require-

978-0-7695-4756-5/12 $26.00 © 2012 IEEE
DOI 10.1109/SERVICES.2012.55

ments for the frequent use of the binary content in messages and to a consumption of the scarcely available mobile network bandwidth. The measurements were concluded using a proprietary messaging service and a service client, providing both dual-style service access interfaces, REST and SOAP-style access to the same service. The executed operations were identical, registering a client, sending and receiving a messages and finally unregistering. By using the same underlying service logic and implementation, we were able to clearly measure and make distinctions between these two service access styles in detail to analyse the service invocation complexities and performance penalties of each solution in detail. The measured details included service request building and invocation, request parsing, request and response transmission, and execution of the service logic. Collecting all the necessary data required instrumentation with the client and client libraries and the service and the server layers themselves.

[7] compares SOAP performance on the alternative transfer protocols and encodings including the standard HTTP, PTCP, various compressed protocols among others on various network connection types. Their findings are that SOAP is not suitable for slow network connections and heavy loads. [8] from the same main authors continues on the same track, stating the benefits of alternative message encoding schemes over the de-facto standard XML. It emphasises the benefits of the binary and compressed XML variants, particularly due to reduced encoding overhead. This same has been perceived in [9] with the notion of the worse-than-expected performance of the JSON format but underlining the benefits of the Google Protobuffer binary variant. Due to this understanding of the binary encoding's benefit, for this study we included an MTOM/XOP binary part encoding for the SOAP messages containing binary content. The MTOM/XOP basically locates the binary part to an MIME attachment section of the wire formatted SOAP-XML message. The MIME attachment is then transmitted in binary, avoiding the necessary encoding and decoding burdens and overhead on both ends as is the case when using Base64.

As with the solutions presented in [10] and [11], we implemented a dual interface for a specific service that is responsible for relaying messages on a proprietary multimedia service platform. We saw that this messaging service was such that implementing both style interfaces was simple and they both provided very similar and comparable operations in a single restricted set of operations. Unlike in [11], we did discover significant differences on performance not only between the SOAP and REST style operation but also the effect of the utilized message encoding pattern adhering also to the results of [9]. Our results did also indicated, unlike in [10], a significant difference between the practical performance of SOAP and REST and their heavy dependency on the utilised message serialisation scheme. To gain a better understanding of the complete service's invocation path and its delays, we carefully instrumented the client, REST and SOAP client libraries, Tomcat's web server part Coyote, Catalina container and finally the service itself. This instrumentation provided us with a better view of the delays on the complete chain of the service method invocation and result returning path in the form of the measured delays in each part on the service invocation path. The analyses of these measurements not only show the effect of the service invocation complexity on both architectural styles but also the effect of the utilised request object marshalling scheme.

The rest of this paper presents a review of the measurement parameters and arrangements and then further reviews the results and analyses them.

II. Tested system

The system utilised for the measurements consisted of a suitably instrumented client and a server stack on a server implemented for sending and receiving messages containing arbitrarily binary or text data content between the client and server. For the purpose of testing and comparing both style Web service interfaces, we had both REST and SOAP versions implemented of both the client and service. The modular design of each allowed switching the employed interface fluently on both, whilst the basic functionality and logic stayed the same. The REST and SOAP versions of the messaging service shared the very same core service logic with its internal message queues and filtering. Similarly, the client was implemented so that the basic logic was the same but where a switch of the client library REST or SOAP-style interface on the service could be invoked. The messaging service requires a registration of the message listener, allows the setting up of filters, sending a message to a designated receiver and furthermore retrieves all received messages directed to the appropriate client. The SOAP interface was very much RPC-like while the REST interface exposed its message queues reserved for the each registered client. Table I defines the messaging service's operational interface for the both interface style and the operations' respective abbreviations used later as a reference. This table does not illustrate the associated attributes of each function call. REST function calls are depicted with the required HTTP verb and the logical URI it is addressed to, whereas SOAP implementation here only uses POST. In the REST version, the clients and the message queues they own are identified with a unique queue ID, which the client receives after registering a URI value in an HTTP-created response's header data and onto which the further calls are directed. Messages directed for other clients are moved to their respective queues by the server. The request and response messages do contain identical payload data, but due to the nature of the SOAP it produces more typical SOAP protocol overhead. In the REST version, the registration is a simple HTTP POST operation which allows conveying all the required parameters as command attributes, including the client identification and access authentication data. Similarly, unregistration is rendered as a simple HTTP DELETE command to a queue specific URI. This would not be sufficient in terms of security in a real world scenario. According to [12] adding an SSL/TSL encryption to the HTTP requests alone would add 70% to the processing time. This

TABLE I
MESSAGING SERVICE'S SOAP AND REST INTERFACES

Service Operation (Abbrv)	SOAP Function	REST Function and URI
Register a client (REG)	RegisterClient()	POST URI:server/
Send a message (SEND)	SendMessage()	PUT URI:server/queue_id
Receive all messages (RCV)	ReceiveMessage()	GET URI:server/queue_id
Unregister a client (UREG)	UnregisterClient()	DELETE URI:server/queue_id

would impact on both SOAP and REST-based implementation identically in this case and is out of scope for this study.

In our tests we used two different kinds of message structures. Both had an identical application-specific message metadata section with the intended message recipient information, authentication information, etc. The payload part in the messages had either an array size of 100 elements or a binary of 128kB mimicking a small photo, audio or video content, while the intended use of this messaging service is in the multimedia domain. With the REST solution, we employed XML, JSON or Google Protostuff message encoding and used SOAP as a standard XML representation with Base64 or MTOM/XOP for the binary part encoding. This method attaches the binary part as XOP encoded element in the MIME attachment part. The utilised message object structures are illustrated in Figure 1. All message formats except Google Protostuff were supported by default by the service frameworks. To add the support for it required Protostuff implementation of the JAX-RS specific MessageWriter and MessageReader interfaces. These message objects that were sent and received and respectively marshalled to a wire format within the HTTP requests, produced the following sized messages as listed in Table II. The total HTTP request/response size and the actual payload are separately listed in it. The SOAP messages embed the application message further into the SOAP envelope, thus increasing the total overhead. While using REST, the serialised application message is the only part in the HTTP request that is transferred that represents the state of the message queue according to the RESTful design principles. This data was collected using the Wireshark network monitoring tool.

A. Instrumentation

We measured and calculated the time of the total round trip, starting from the service invocation and ending with the service method returning consumed. The total latency on the client's service invocation included forming the request data object, marshalling it to a JSON, XML or other utilised wire format, transmission over the network to the server and the server unmarshalling the request, directing it to the correct service and invoking the service method. This is followed by the server processing of the service request and further by the forming of the response object, its marshalling, transfer, unmarshalling on the client and returning the invocation call. Both the client and server were instrumented on several levels

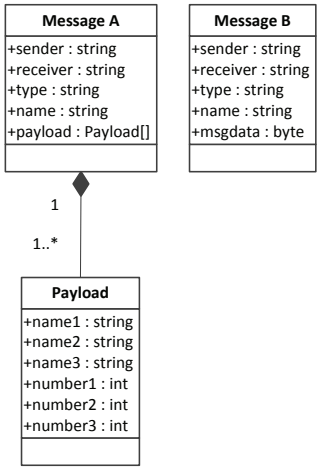

Fig. 1. Sent and received message object graph types. Message B contains a binary data part which encodes as an attachment in MTOM/XOP method or Base64 encoded portion in a standard SOAP and REST. Message A has an array of 100 Payload elements each including string fields of 150 bytes and 30 bytes of integer data.

to measure delays at each phase of the request and response processing by time stamping each step. With the multiple instrumentation points, we were able to determine the accurate delay of each step on each layer during the messaging service's operation invocation. The time stamping and measurements were conducted on the entry and exit points of the actual client, client library and HTTP server transfer, container processing and the actual service processing on the servlet through Axis2 or Jersey stacks. We conducted the measurements of each operation using the content types and message formats with SOAP and REST as listed in Table II. The measurements were conducted using DelayTool [13], enabling the collection of highly accurate timestamps at the nanosecond level for the calculation of the experienced delays. The measurement points and their designations are illustrated in Figure 2.

B. Calculations

The time stamping on the request-response path starts from the client's invocation, CM_{Invoke}, of the client library's service method from where the request object is passed and serialised to a wire format and sending started, stamped as CL_{Send}, to the recipient HTTP server stamped as $\mathrm{SH}_{Receive}$ marking reception of the message complete. From that point, the request is forwarded to the servlet container and stamped as SC_{Invoke} and further to the Axis2 or Jersey framework and the messaging service at SS_{Invoke}. After the service processing and container returning at SS_{Return} and SC_{Return} the response send is started from the HTTP server at SH_{Send}. The response is fully received at $\mathrm{CL}_{Receive}$ and parsed and directed to the client fully returning at CM_{Return}. With the instrumentation points providing time stamps during the request-

TABLE II
HTTP REQUEST/RESPONSE MESSAGE (HTTP) AND PAYLOAD (PL) SIZES IN BYTES FOR EACH MESSAGE SERVICE OPERATION

		REG		SEND		RCV		UREG	
		Req	Resp	Req	Resp	Req	Resp	Req	Resp
Array of 100 elements, payload 18000 bytes	REST XML	353	305	30945	301	161	30872	201	91
	REST JSON	353	305	25896	301	162	25822	201	91
	REST GPS	353	305	24523	301	168	24408	201	91
	SOAP XML	966	309	31931	309	911	31635	974	309
Binary content of 128kB	REST XML	353	305	176081	301	161	176047	201	91
	REST JSON	353	305	175909	301	162	175898	201	91
	REST GPS	353	305	131912	301	168	131797	201	91
	SOAP XML	966	309	176707	309	911	176301	974	309
	SOAP MTOM	966	782	133454	782	1371	133207	1434	782

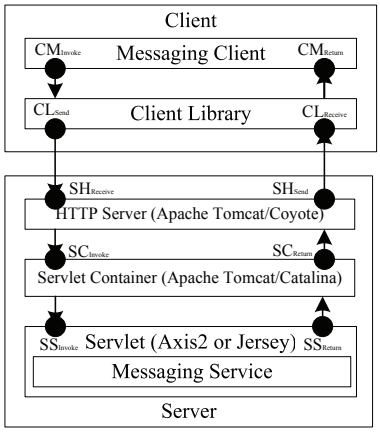

Fig. 2. Server and client layers with instrumentation points for the measurements and their corresponding designations.

response flow the calculations for the delays experienced on different parts can be made. The round trip time at the client RTT_{Client} can be calculated as in (1), which is a total time calculated from the first and last time stamp. It is calculated only for a cross reference and estimation of the possible error accumulation along the complete delay path measurements and calculations as presented later.

$$RTT_{Client} = CM_{Return} - CM_{Invoke} \tag{1}$$

As the total round trip consists of the sum of the delay chain on the request, service processing and response path, it can be also expressed as in (2) comprising all processing and transmission step delays.

$$\begin{aligned} RTT_{Client} = D_{ClientLib} + D_{Req} \\ +D_{ServerCont} + D_{ContService} + D_{ServiceProc} \\ +D_{ServiceCont} + D_{ContServer} + D_{Resp} \\ +D_{LibClient} \end{aligned} \tag{2}$$

where the delays between the layers on the request path are client invokes the client library $\text{D}_{ClientLib}$ as in (3), requests the transfer D_{Req} as in (4), HTTP server invoking servlet container $\text{D}_{ServerCont}$ as in (5) and container invoking the service $\text{D}_{ContService}$ as in (6). The service processing time, $\text{D}_{ServiceProc}$, is defined as in (7). Respectively on the response path, they are service returning to the container $\text{D}_{ServiceCont}$ as in (8), container invoking the HTTP server $\text{D}_{ContServer}$ as in (9), response transfer delay D_{Resp} as in (10) and the client library returning the client $\text{D}_{LibClient}$ as in (11). In the measured results the difference between the results of (1) and (2) proved to be negligible.

$$D_{ClientLib} = CL_{Send} - CM_{Invoke} \tag{3}$$

$$D_{Req} = SH_{Receive} - CL_{Send} \tag{4}$$

$$D_{ServerCont} = SC_{Invoke} - SH_{Receive} \tag{5}$$

$$D_{ContService} = SS_{Invoke} - SC_{Invoke} \tag{6}$$

$$D_{ServiceProc} = SS_{Return} - SS_{Invoke} \tag{7}$$

$$D_{ServiceCont} = SC_{Return} - SS_{Return} \tag{8}$$

$$D_{ContServer} = SH_{Send} - SC_{Return} \tag{9}$$

$$D_{Resp} = CL_{Receive} - SH_{Send} \tag{10}$$

$$D_{LibClient} = CM_{Return} - CL_{Receive} \tag{11}$$

C. Measurement setup

The measurements were carried out utilising one and the same messaging service, which provided REST and SOAP-style interfaces. The clients were produced with the least effort approach from producing the SOAP client from WSDL with a generator provided with Axis2 and the REST client using the POJO approach with JAXB conventions. Similarly, the server interfaces were produced from POJO service logic, trying to avoid manual implementation as much as possible. REST solutions did require, however, some additional work in terms of designing and integrating the interface component with the service logic. The RESTful implementation was tested using the wire formats of JSON, XML and Google Protostuff, while SOAP relied on SOAP-XML and SOAP XOP/MTOM for the binary content. The message service was run on an Apache Tomcat 6.0.35 servlet container employing its default Web server and other components. In its SOAP configuration the messaging service was implemented using Axis2 1.5.6 framework and for the REST interface, Jersey 1.11 implementing the current JAX-RS API was used. Their associated client libraries were employed at the client end. Equally, all parts of the system were instrumented with the DelayTool delay measurement tool. All the tests were run with a local loop networking on the same computer with Intel Core 2 Duo @ 2.54 GHz, 4GB of RAM, Ubuntu Linux with a standard series kernel version 2.6 and Java virtual machine SE 6.

Each combination of the Web service protocol, message serialisation format and content type was repeatedly measured 1,000 times to allow the results to stabilise in longer time series and therefore achieve more accurate averages. The first one or two time stamping cycles at the beginning of the set of 1,000 repeats gave over ten times higher delays between the measured phases. This was due to the startup phase of the Java Virtual Machine and they were excluded from the average results. In addition, it was apparent how Java garbage collection affected the measurements by clearly increasing delays at roughly every 100th measurement.

III. Results and analysis

A. Round trip time

After measuring and calculating RTT_{Client} from the collected data, we plotted it on the graphs presented in Figures 3 and 4. Figure 3 shows how even the slowest REST based solutions, almost equal REST-JSON and REST-XML, are still many times faster for the each service request operation invoked than SOAP. Sending and receiving the messages from the server using the SOAP solution had a delay of 3 and 4 times greater sending and receiving respectively than these REST solutions. However, these REST solutions experienced round trip times twice as high as REST-GBP. The explanation can be found in Table II which shows the message sizes achieved by the each associated encoding method, with SOAP having the highest overhead and a much more complex message processing path. These apply only to sending and receiving operations while the client registration and unregistration took almost an equal amount of time for the each REST solution. As indicated by the study of the encoding methods at [9], the REST-JSON does not perform much better than REST-XML here. The used JSON codec lacks marshalling and unmarshalling performance which affects the overall round trip time so that the JSON's advantage of shorter data representation and therefore faster network transfer is lost. Figure 4 plots the respective results achieved with binary

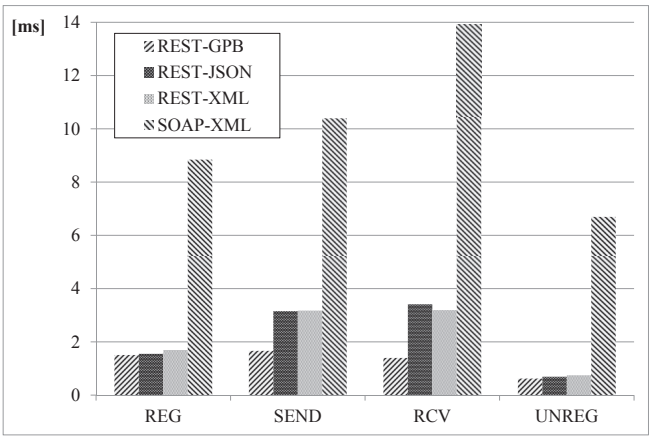

Fig. 3. Round trip time, RTT_{Client}, achieved at the client for each operation with an array content of 100 elements.

content. Here it is again apparent how SOAP-based solutions do not perform as well as REST solutions. The interesting detail among the SOAP solutions is how the MTOM/XOP variant that is optimised for the binary content carriage is no faster than the traditional SOAP-XML, but in fact the contrary. Roughly, sending causes a delay of 3 - 10 times longer and receiving 3 - 6 times longer. However, REST-GPB reaches outstanding results compared with any other producing delays of 1/3 - 1/2 of the other REST solutions. Registration and unregistration round trip time for each REST solution is roughly the same, as these do not require carriage of the encoded payload as the parameters are conveyed in HTTP headers.

B. Delays in the different processing steps

Figure 5 shows the share of the delay distributed between the different phases of the request-response processing path. Included are only the sending and receiving operations while the space is limited and the shares in registration and unregistration do not offer equally interesting information. The only interesting detail is in unregistration, in which the SOAP solutions spend significantly longer in the service processing phase. While the implementation in the background messaging service itself is the same as in the REST, one explanation is the initialisation and setup of the SOAP processing when the first SOAP message including the registration invocation is received. The results show how the biggest delay is created

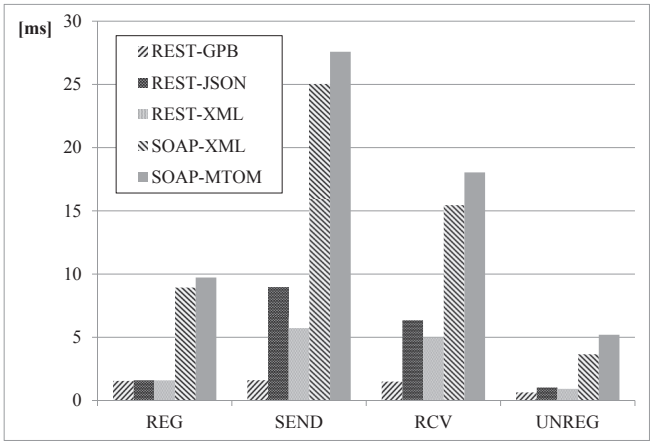

Fig. 4. Round trip time, RTT_{Client}, achieved at the client for each operation with a binary content of 128kB.

by D_{Req}Req in sending and D_{Resp}Resp in receiving as expected, while they contain the delay caused by the carriage of the actual payload for those operations. In the SOAP-XML solution, however, the shifted high share of the delay of the client library returning is caused by the behaviour of the WoodStox XML parser with delayed parsing. This parser parses the message object only when accessed as is the case here when the message is returned as an object to the client. The REST-JSON solution is slightly slower than the corresponding XML solution. The distribution of the delays shows how the JSON solution in sending operation consumes more time in the request writing phase, D_{Req}Req, than the XML solution. However, the overall times are almost equalled out when the XML solution spends more time invoking the service from the container, $\mathrm{D}_{ContService}$ContService, phase. The same phase consumes roughly half of the time on the solutions with JSON manifesting faster message object unmarshalling. Potentially, the JSON solution can achieve higher overall throughput due to somewhat smaller message object representation only in the real life networked applications. In overall results, the REST employing Google Protobuffer encoding proves to be contributing to the fastest results in this case. The graph in Figure 6 represents the same results with a message containing a single piece of binary data representing e.g. a photo transfer from a client to the server. Included are all the messaging formats able to carry binary content. REST-JSON and XML, as with the standard SOAP-XML, do contain the binary content as a Base64 encoded text string known for being slow and producing significant overhead in relation to the original content size. This can be seen from the table listing the message sizes but also on the total RTT and the share of the processing phases illustrated. Sending and receiving a message with Base64 encoded binary content shows as an additional share either in the actual request and response writing and transfer but also in the intermediate processing phases, depending on the actual message marshaller. For example, SOAP utilises the WoodStox XML serialising and parsing with Axis2 proprietary object model mechanism utilising deferred parsing where the content is parsed only when the fields are accessed; this is evidential in the values of $\mathrm{D}_{ContService}$ContService and $\mathrm{D}_{LibClient}$LibClient during the internal transitional request and response returning phases. Surprisingly in this light, the RTT total times are disadvantageous for SOAP-MTOM over the traditional SOAP-XML method in these tests where only the local loop networking was utilised. However, as we can see from the distribution of the results, the processing phases on the server are much more time consuming for the MTOM and XML variants in the sending phase and the opposite for the client receiving. The completed message size in the wire format is significantly smaller, suggesting faster network transmissions in real life cases with less available network capacity than what is available in the local loop. On the other hand, with the binary content the REST-JSON does not marshall to remarkably smaller messages than the REST-XML. Therefore, according to the results the slightly smaller message size offers only very modest benefits over the REST-XML here. REST-JSON accumulates a higher total RTT, including higher shares of the request and response writing and network transfer. This is due to the less than perfect throughput capability of the JSON marshalling mechanism as shown inčitetai2. The fastest overall time was reached by the REST with Google Protobuffer encoding again.

IV. Conclusion

The analysis of the results carried out for this study clearly show how in all cases in the light of the described scenario, REST is the fastest. The conventional SOAP-based solutions here tend to cause 4-5 fold delays compared with REST-XML and JSON on the request-response processing cycle. Furthermore, the difference between the SOAP solutions and the fastest REST-GBP was 5-10 times. It was not only the utilised XML encoding but also the intrinsic complexity of SOAP. As was demonstrated with REST-XML, the XML alone was not counterproductive.

The processing chain appears to be more heavily loaded for REST-JSON than REST-XML, but the only benefit that the JSON solution is able to achieve in real life applications is the possibility for the faster network transmission of the marshalled request-response message objects due to the tighter encoding with JSON. In this scenario with very high bandwidth local loop networking, this benefit did not emerge at all, rendering the benefits of JSON insignificant, with both reaching about the same delays, JSON being even a little bit slower. With real networking, the JSON solution would achieve a higher throughput due to the less verbose message object encoding overhead with the regular content. This depends on the network bandwidth of course, as the size difference for the benefit of the JSON encoding in the marshalled message object with an array content was 16%. However, that encoding size benefit is mostly lost with the Base64 encoded binary content in the message. The slightly higher latency in the JSON marshalling and almost similarly high overhead in

comparison to the XML with similar binary content caused higher latencies for REST-JSON. Similar circumstances apply to the SOAP-MTOM solution with the binary content in comparison to SOAP-XML. The best performing solution with both message content types, however, was REST-GPS. It did spend only half of the time in sending and receiving operations in comparison to the other REST solutions. The registration and unregistration was on the same levels with the other REST solutions, as they did not embody the transfer of lengthy message bodies in the HTTP payload unlike with SOAP.

Overall, the message size proved to be a very significant factor here among the ideological overall complexity difference between SOAP and REST. However, the processing in the overall request-response chain tends get heavier the more compact the messages are, contributing to the smaller network transfer delay due to the smaller size that compensates the overall result. This is the key when using a more realistic networking scenario to achieve a desired space and time relation, which the REST-GPB is able to achieve with little effort.

V. Future work

In this paper, we only estimated that the delay caused by the actual network transmission is going to be higher with more verbose XML solutions. Therefore, e.g. in comparison to the JSON solutions, the actual and measured combined total round trip time is unknown on real networking situations. However, as we stated, that the total time spent in forming and reading the JSON data is higher than in corresponding XML format, JSON contributes to a faster network transmission on a real network due to a shorter data representation. This may result a shorter round trip time for JSON solutions but the effects of this needs be tested on the more realistic networking environment. It requires a challenging and careful synchronization mechanism of the time stamping instrumentation clocks if using two separate networked computers.

Acknowledgment

The authors would like to thank the ITEA2-ACDC, ITEA2-SPY and Celtic-Motswan projects for offering the case study and funding for this work.

References

[1] "Developer guide - protocol buffers - Google code." [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/overview.html

[2] N. Mendelsohn, H. Ruellan, M. Gudgin, and M. Nottingham, "XML-binary optimized packaging," W3C, Tech. Rep., jan 2005, http://www.w3.org/TR/2005/REC-xop10-20050125/; W3C Recommendation.

[3] N. Mendelsohn, M. Gudgin, H. Ruellan, and M. Nottingham, "SOAP Message Transmission Optimization Mechanism," W3C,, Tech. Rep., jan 2005, http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/; W3C Recommendation.

[4] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," 2000. [Online]. Available: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

[5] M. Hadley, "Web application description language," W3C,, Tech. Rep., aug 2009, http://www.w3.org/Submission/wadl/; W3C Recommendation.

[6] K. Kadner, "Unified service description language xg final report," W3C,, Tech. Rep., oct 2011, http://www.w3.org/2005/Incubator/usdl/XGR-usdl-20111027/; W3C Incubator Group Report.

[7] J. Kangasharju, S. Tarkoma, and K. Raatikainen, "Comparing soap performance for various encodings, protocols, and connections," in *Personal Wireless Communications, volume 2775 of Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 397–406.

[8] J. Kangasharju, T. Lindholm, and S. Tarkoma, "Xml messaging for mobile devices: From requirements to implementation," *Comput. Netw.*, vol. 51, pp. 4634–4654, November 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1290550.1290816

[9] T. Aihkisalo and T. Paaso, "A performance comparison of web service object marshalling and unmarshalling solutions," in *Proceedings of the World Congress on Services, SERVICES 2011, Washington, DC, USA, July 4-9, 2011*. IEEE Computer Society, 2011, pp. 122–129.

[10] M. Pagni, J. Hau, and H. Stockinger, "A multi-protocol bioinformatics web service: Use soap, take a rest or go with html," may. 2008, pp. 728 –734.

[11] G. Mulligan and D. Gracanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," dec. 2009, pp. 1423 –1432.

[12] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan, "Anatomy and performance of ssl processing," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 197–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=1317536.1318411

[13] P. Paakkonen, J. Prokkola, and A. Lattunen, "Instrumentation-based tool for latency measurements," in *International Conference on Performance Engineering '11*. ACM, March 14-16, 2011 2011.

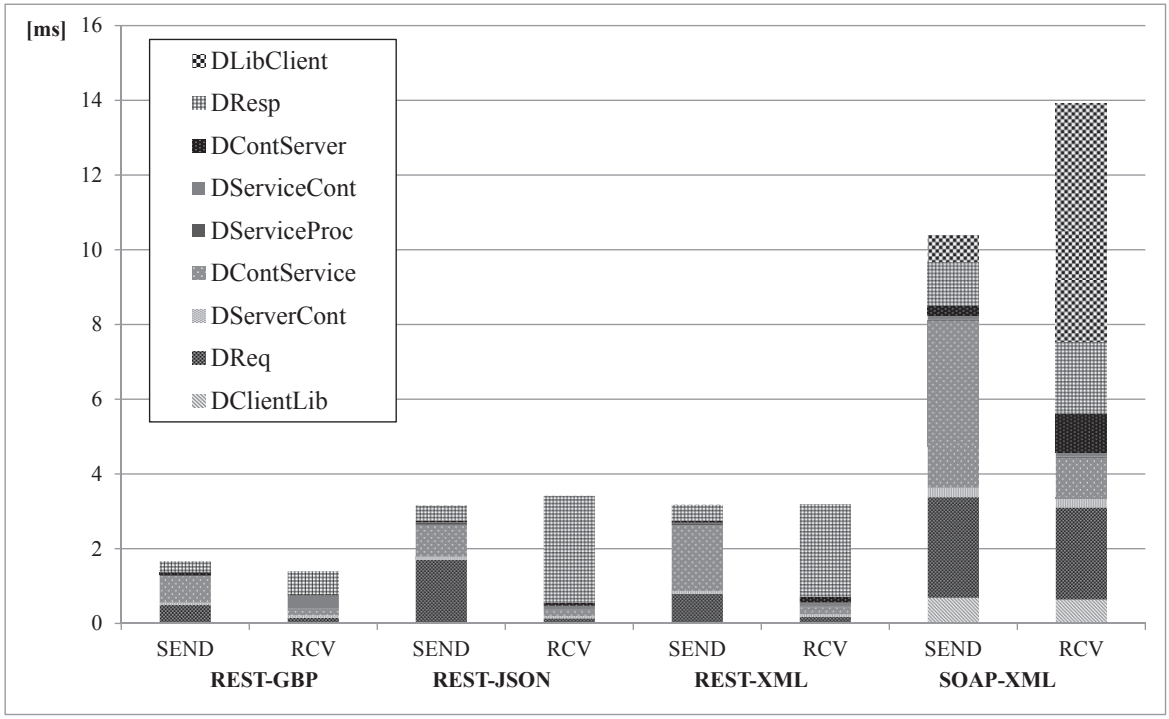

Fig. 5. Delays in the different phases of the client and server stack during SEND and RCV on the REST and SOAP interfaces with a messages containing an array of 100 elements.

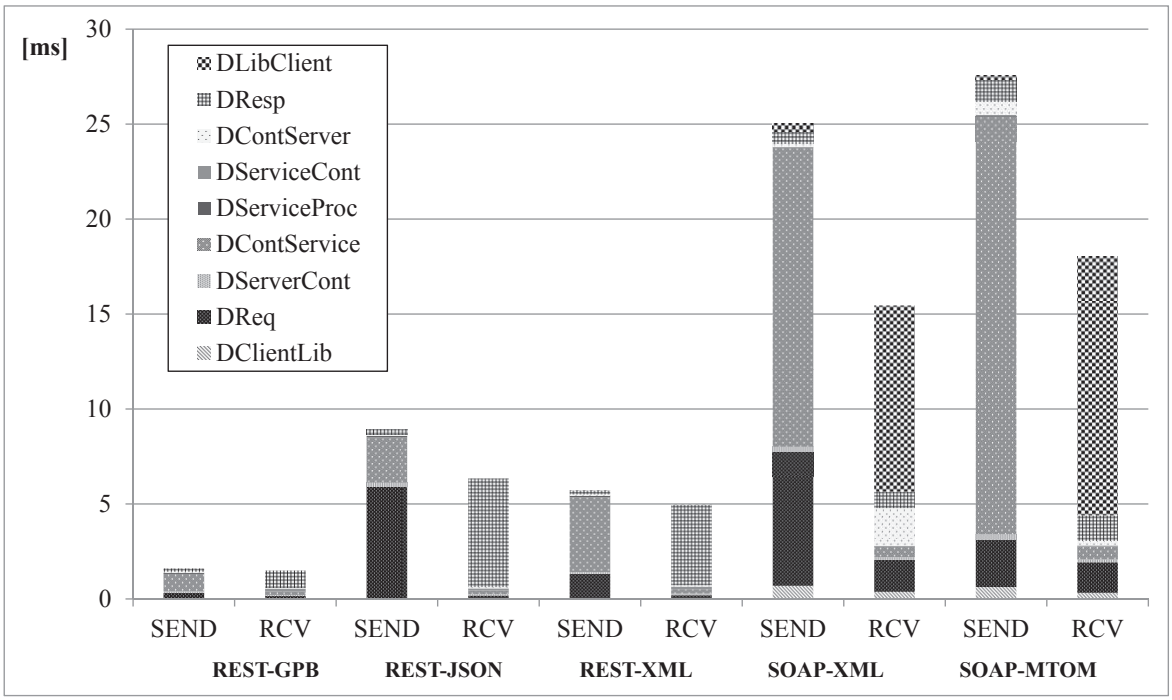

Fig. 6. Delays in the different phases of the client and server stack during SEND and RCV with REST and SOAP interfaces with a messages containing binary content of 128kB.