Employing Dynamic Object Offloading as a Design Breakthrough for SOA Adoption

Quirino Zagarese *, Gerardo Canfora, Eugenio Zimeo
{quirino.zagarese, gerardo.canfora, eugenio.zimeo}@unisannio.it

Department of Engineering, University of Sannio

Abstract. In several application contexts, Web Services adoption is limited due to performance issues. Design methods and migration strategies from legacy systems often propose the adoption of coarse-grained interfaces to reduce the number of interactions between clients and servers. This is an important design concern since marshaling and transferring small parts of complex business objects might entail sensible delays, especially in high latency networks. Nevertheless, transferring large data in coarse-grained interactions might bring useless data on the client side, whereas a small part of the transferred object is actually used.
This paper presents a novel approach to extend existing Web services run-time supports with dynamic offloading capabilities based on an adaptive strategy that allows servers to learn clients behaviors at runtime. By exploiting this approach, service based applications can improve their performances, as experimental results show, without any invasive change to existing Web services and clients.

Keywords: web-services, performance, xml serialization, service design, adaptation

1 Introduction

Web Services emerged as a breakthrough for cross-platform communication in heterogeneous environments thanks to the use of standards such as XML, SOAP [12] and WSDL[13]. However, performance issues slew down their adoption in environments where performance matters. This is mostly due to high latency and low bandwidth characterizing world-wide networks, something really tough to deal with when it comes to marshaling and transferring complex business objects. Meaningful examples of such objects can be found in the PLM (Product Life-cycle Management) area, where large data concerning structural and management aspects of products need to be shared among multiple systems [4], and in the telecommunication area, where providers sales and ordering systems expose interfaces to extract data on customers' orders [3]. In these complex and often multi-organisational environments, design an migration strategy to embrace the SOA paradigm can be very challenging.

* The work of this author is supported by Provincia di Benevento

The need to minimize the overhead due to network latency often leads to the design of coarse-grained service interfaces. However, this approach can cause unnecessary data transfers, since each service invocation implies the serialization of a large XML document, even if the client needs only a small portion of such document. Over the years, many solutions have been proposed to minimize client-server interaction time, by compressing XML documents or by incrementally loading XML structures. The XMill[1] tool, Cheney's SAX events encoding [6] and Rosu's incremental dictionary-based compression[7] represent important results concerning the former approach. Incremental loading has been proposed in [2], where web services calls are inserted in XML documents in place of their results, as well as in [8], where a placeholder of the result of an invocation is propagated among services and lazily loaded only when needed. The latter approach seems more promising since it allows for reducing the amount of data transferred on the network by delivering to the caller only the data used by the application, whereas XML compression could still be used to further reduce the size of exchanged messages. However, the idea of incremental loading is in its infancy and current Web services technology does not yet provide tools for smart lazy-serialization of object attributes.

This paper tries to fill this gap by proposing an architecture for a middleware aimed at optimizing network-based interactions for data-intensive services. Optimization is achieved by applying a technique that combines eager and lazy serialization of the attributes of objects resulting from services invocations. We call such technique *Dynamic Object Offloading*.

The remainder of the paper is organized as follows. Section 2 introduces and characterizes the problem. Section 3 describes the architecture of the proposed middleware. Section 4 presents some preliminary experimental results. Section 5 discusses the results, concludes the paper and outlines future work.

2 Problem characterization

Most systems that expose functions through web-services communicate with other systems through an inherently eager-loading approach: every time a new request is received, a result is computed, serialized to the network, and unserialized on the client side. A different approach consists of adopting a lazy-loading strategy. In such case, for each incoming request, a result is computed as well, but XML fragments get serialized to the client as soon as they are needed. The former approach may lead to unnecessarily transferred data, but the interaction is completed within a single request-response cycle. This means any network latency overhead contributes only once to the overall time needed to complete the client-server interaction. The latter approach ensures that only needed fragments get serialized to the client, but leads to multiple request-response cycles. Consider an object *Obj* containing n attributes whose sizes are defined in vector $S = \{s_1, s_2, \ldots, s_n\}$. If the web-services stack is based on an eager-loading strategy, the time needed by the client application in order to retrieve *Obj* can be computed as:

$$T_{retrieve} = L + \frac{\sum_{i=0}^{n} s_i}{Th}$$

where L represents SOAP transport latency and Th stands for SOAP transport throughput. In order to characterize $T_{retrieve}$ for a lazy-loading strategy, we introduce a binary vector $A = \{a_1, a_2, \ldots, a_n\}$ where the generic element a_i is defined as:

$$a_i = \begin{cases} 1 \text{ if } client \;\; uses \;\; n_i \\ 0 \; otherwise \end{cases}$$

As mentioned, for a lazy-loading strategy, multiple request-response cycles occur, since each access to a not-yet-loaded attribute triggers a new request. For such scenario, $T_{retrieve}$ can be defined as:

$$T_{retrieve} = L(1 + \sum_{i=0}^{n} a_i) + \frac{\sum_{i=0}^{n} a_i s_i}{Th}$$

To achieve a performance gain in terms of $T_{retrieve}$, the following must apply:

$$L \cdot \sum_{i=0}^{n} a_i + \frac{\sum_{i=0}^{n} a_i s_i}{Th} < \frac{\sum_{i=0}^{n} s_i}{Th} \tag{1}$$

If we assume that all attributes in *Obj* exhibit the same size S and define the number of accessed attributes as $N_{access} = \sum_{i=0}^{n} a_i$, then we can rewrite (1) as follows:

$$LN_{access} + \frac{N_{access} S}{Th} < \frac{NS}{Th}$$

where N is the total number of attributes in *Obj*. A performance gain is achieved if:

$$N_{access} < \frac{NS}{L \cdot Th + S}$$

So far, we have explored when a plain lazy-loading strategy performs better than a eager-loading one. One could combine the mentioned approaches in order to minimize both request-response cycles and downloaded fragments. In this case, N_{access} must be split into two terms: N_{eager} and N_{lazy}. The former represents the number of attributes downloaded during the first request-response cycle; the latter concerns the number of attributes downloaded on demand, by issuing further requests. The problem then consists of keeping the value of N_{lazy} as low as possible in order to minimize $T_{retrieve}$.

3 Middleware architecture

In this section, we introduce an architecture for a middleware that minimizes $T_{retrieve}$ by combining eager and lazy-loading. We split XML documents returned by Simple Object Access Protocol (SOAP) Web Services into two sets respectively having N_{eager} and N_{lazy} cardinalities: the former contains the eagerly loaded object attributes, the latter contains the lazy loaded ones. For each service request, the server decides which attributes are likely to be used by the client application, thus eagerly serializing them, and which are not, making them available for lazy access.

Choosing which attributes should be eagerly serialized cannot be evaluated in a static way, since clients behaviour is not likely to be known a priori. For this reason, we propose to monitor clients behaviour and characterize them by considering the following scenarios:

1. clients accessed an eagerly loaded property of the object
2. clients accessed a lazily loaded property of the object
3. clients did not access a lazily loaded property of the object
4. clients did not access an eagerly loaded property of the object.

The first and third scenarios describe desirable situations, as the system predicts the actual client behavior: no cache-misses are issued, nor any eager serialization is wasted. The second and forth scenarios concern negative situations; the former takes place when the server does not eagerly serialize an object attribute, but the client tries to use it; the latter happens when the server eagerly serializes an attribute, but the client does not use it.

Clients behaviors are stored in a *Knowledge Base* (KB) component which exposes two kinds of operations: "query", to decide if an object attribute should be eagerly serialized, and "update", to instruct it about clients behaviour.

Figure 1 describes a client-server interaction mediated by our proposed middleware. Gray components realize a standard architecture based on Web Services. *Service Interceptor* is a JAX-WS [5] compliant component, able to access service incoming and outgoing messages. It is responsible for analysing services invocations results and applying dynamic offloading, based on information obtained by querying KB. It implements the Interceptor architectural pattern that allows services to be added transparently to a framework and triggered automatically when certain events occur[11].

Data Access Layer (DAL) is a set of components that hide objects attributes loading strategy from the client-side and notify server about clients behaviour. *Lazy Data Access Service* (LDAS) enables lazy loading of objects attributes. It is responsible for updating the Knowledge Base, on the basis of notifications from DAL, and delivering offloaded attributes. It behaves like a dictionary where each entry is an *InvocationID-ObjectPropertyName* pair.

A sample invocation scenario can help better describing the whole architecture. When a service request is issued by a client (no. 1 on the diagram), Service Interceptor inspects the outgoing response message and queries the KB (2, 3) in

order to decide which attributes of the returned object should be eagerly serialized to the client, after assigning an invocation id to them (5), and which ones should be offloaded to the LDAS (4). Offloading is performed asynchronously, in order to minimize response time.

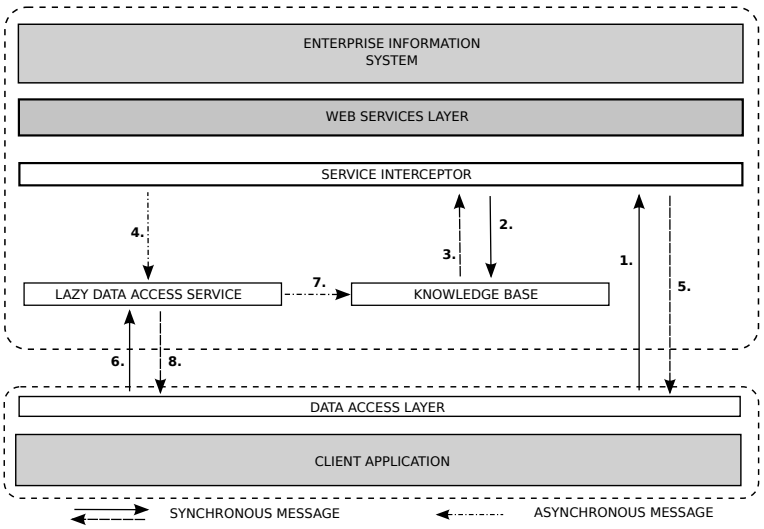

Fig. 1. Interaction phases in the proposed architecture

After the client receives the result of the invocation, DAL keeps track of accesses to object properties. When the client requests an eagerly loaded attribute, DAL simply saves such information and returns the attribute. When a request for an offloaded attribute is issued, DAL forwards a request (6), containing the id related to the invocation that generated the root object and accesses related data recorded since the last miss, to LDAS that, in turn, asynchronously updates KB (7) and serializes the requested attribute back to the client (8).

4 Preliminary evaluation

We have implemented the described architecture in a prototype system based on the Apache CXF framework [10]. CXF is a JAX-WS compliant implementation and offers a flexible API for message interception.

The *LDAS* component has been implemented as a separate CXF Web service. The *KB* implements a profiling strategy, where for each attribute in the business object returned by the target service, the following probability is computed and updated:

$$P_{access} = \frac{\#attribute\_accesses}{\#service\_invocations}$$

We defined a target service that allows the client to retrieve a sample business object containing 100 fixed-size attributes. The client is emulated through a component that actually invokes the target service, accesses the attributes of the returned objects and possibly raises requests to *LDAS*. The access pattern is specified through a binary input vector *Acc*, where $Acc[i]$ is defined as follows:

$$Acc[i] = \begin{cases} 1 \text{ if } client \;\; uses \;\; n_i \\ 0 \; otherwise \end{cases}$$

where n_i is the i^{th} attribute in the business object.

The evaluation process has been based on two cases. First, we compare eager loading, pure-lazy loading, random dynamic offloading and learning dynamic offloading for a client that never changes its behaviour. The eager loading approach makes the server always serialize the whole object; when pure lazy loading is used, the server always returns an empty object and the client can retrieve attributes by issuing new requests; with random dynamic offloading, the server randomly decides, for each attribute, whether it should be serialized; learning dynamic offloading implements a simple strategy where the probability to use an attribute is computed as the number of accesses to such attribute, divided by the number of invocations of the service, as outlined in section 3. In this phase, we make the client access a constant number of attributes; in the first round, the client constantly accesses 10 of the 100 attributes in the business object. Then we make the client access 20 of the 100 attributes and so on, until all of the attributes in the business object get accessed. Each round consists of 100 invocations of the target service. For each round we evaluate the average $T_{retrieve}$. The process has been repeated for three different attribute sizes: 1MB, 100KB, 10KB. These are realistic sizes for attributes in business objects characterizing the PLM area, such as object attributes encapsulating CAD files or images. By repeating measures for different attribute sizes, we are also able to outline how multiple request-response cycles impact $T_{retrieve}$, when network overhead dominates object size.

In the second case, we compare the mentioned strategies for a client that randomly decides to access or not a generic attribute. We evaluate the average $T_{retrieve}$ over 1000 invocations of the target service.

Experimental results are reported in table 1 where each column contains the average $T_{retrieve}$ values for a round of 100 invocations, given a specific N_{access} value. The right most column contains the average $T_{retrieve}$ values obtained in the second case. Results show that a learning strategy outperforms all the other strategies in terms of $T_{retrieve}$, if the client application keeps a static behaviour and if it does not access all the attributes of the business object.

We expected that a pure lazy-loading strategy would outperform eager-loading. Surprisingly, a random offloading strategy outperforms eager-loading, as long as the client does not access more than 90% of the attributes. The table shows that a pure eager-loading strategy is nearly always the worst choice, if the business object exhibits coarse-grained attributes. As object granularity decreases, lazy-loading and random offloading become less competitive. Learning offloading still outperforms other strategies regardless of object granularity.

Table 1. Experimental results with static and random client behavior

	N_{acc}	10	20	30	40	50	60	70	80	90	100	*Rand.*
	Size	1MB										
$T_{retrieve}[s]$	Eager	119,89	119,89	119,89	119,89	119,89	119,89	119,89	119,89	119,89	119,89	119,89
	Lazy	12,53	25,02	37,50	49,98	62,47	74,95	87,44	99,92	112,40	124,89	62,47
	Rand.	66,21	72,85	79,39	85,23	91,44	97,85	103,28	110,15	115,97	122,36	91,25
	Learn.	12,04	24,03	36,02	48,00	59,99	71,98	83,97	95,96	107,95	119,94	94,78
	Size	100KB										
$T_{retrieve}[s]$	Eager	11,75	11,75	11,75	11,75	11,75	11,75	11,75	11,75	11,75	11,75	11,75
	Lazy	1,72	3,39	5,06	6,73	8,40	10,07	11,74	13,41	15,08	16,75	8,40
	Rand.	6,70	7,56	8,42	9,30	10,05	10,94	11,79	12,57	13,41	14,25	10,08
	Learn.	1,23	2,40	3,58	4,75	5,93	7,10	8,28	9,45	10,63	11,80	10,29
	Size	10KB										
$T_{retrieve}[s]$	Eager	1,22	1,22	1,22	1,22	1,22	1,22	1,22	1,22	1,22	1,22	1,22
	Lazy	0,67	1,28	1,90	2,52	3,14	3,75	4,37	4,99	5,60	6,22	3,14
	Rand.	0,93	1,25	1,54	1,88	2,19	2,48	2,78	3,13	3,39	3,72	2,18
	Learn.	0,17	0,29	0,42	0,54	0,66	0,78	0,90	1,03	1,15	1,27	2,06

Results from the second case show that learning offloading is still a good compromise, also if the client exhibits an unpredictable behaviour. The results shown in the right most column of table 1, confirm again that whilst lazy-loading outperforms other strategies when the business object contains coarse-grained attributes, it becomes the worst choice as soon as granularity decreases. On the other hand, random and learning offloading are the best compromises.

5 Conclusion

Experimental results show that learning offloading can considerably reduce the amount of time needed by a client to retrieve a business object, or a useful subset of its attributes strictly needed to execute its business logic. Client predictability is a key factor: a client exhibiting a static behaviour leads to a remarkable reduction of $T_{retrieve}$. However, even when the client is characterized by a random behaviour, results show that learning offloading is still a good compromise between pure-eager and pure-lazy loading.

An important requirement to make correct decisions on eager or lazy serialization is to have a good idea of how a client is going to use the object returned by a service invocation. The decision should be based on a realistic expectation of the customer usage, which, in turn, depends upon the specific task to be performed in a given interaction. A strong basis of information on how the object attributes will be accessed will result in optimized decisions that will exhibit fewer failures in the field. In this context, usage profiles can be useful to actively gather information on how clients are actually using the objects they receive from services. We intend exploring usage profiles to characterize families of interaction between a set of services to perform a given task; usage profile

information will then be used to guide predictive decisions on eager and lazy serialization of sets of attributes based on the task to be performed.

The attributes of an object do not live in isolation and, for any given usage profile, certain subsets of attributes are more likely to be used than others. Thus, once the usage profile is known, access to an object attribute can suggest which attributes are likely to be accessed in the near future; this information can be used to implement predictive strategies for serialization. Of course, relationships among attributes are non-deterministic, and this calls for probabilistic models. Among these models, Bayesian networks are a promising tool to infer the probability that an attribute be accessed, given the accesses to other attributes, by modeling the conditional dependencies via a directed acyclic graph. We plan exploring learning procedures for Bayesian networks to learn the network that depicts the dependencies among attributes from usage profiles data.

6 Acknowledgements

The authors would like to thank C. Sementa, A. K. Hosseini and P. Cantiello for stimulating discussion on preliminary ideas behind this work [9].

References

1. H. Liefke, D. Sucio, XMill: an Efficient Compressor for XML Data, in Proc. of ACM SIGMOD international conference on Management of data, pp. 153-164, 2000.
2. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and Web services integration. In VLDB, 2002.
3. S. Amer-Yahia and Y Kotidis. A Web-service architecture for efficient XML. data ex-change. In ICDE, 2004.
4. Siemens PLM Software , Open product lifecycle data sharing using XML, 2011.
5. JSR-000224 Java API for XML-Based Web Services 2.0, http://jcp.org/aboutJava/communityprocess/final/jsr224
6. J. Cheney, Compressing XML with multiplexed hierarchical PPM models, in Data Compression Conference, 2001, pp. 163-173. [Online]. Available: http://citeseer.ist.psu.edu/cheney01compressing.html
7. M.C. Rosu, A-soap: Adaptive soap message processing and compres- sion, In Proceedings of the IEEE International Conference on Web Services. Salt Lake City, Utah, USA, pp. 200207, 2007.
8. Giancarlo Tretola, Eugenio Zimeo: Extending Web Services Semantics to Support Asynchronous Invocations and Continuation ICWS 2007: 208-215
9. Q. Zagarese, C. Sementa, A. K. Hosseini, P. Cantiello, Improving the performance of Web Services by Dynamic Object Offloading, in WIP Proceedings of PDP 2011, Ayia Napa(Cyprus) 2011
10. Apache CXF - http://cxf.apache.org/
11. Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2, John Wiley & Sons, pp. 101, 2000
12. SOAP Specifications - http://www.w3.org/TR/soap/
13. Web Service Definition Language - http://www.w3.org/TR/wsdl