Enabling Advanced Loading Strategies for Data Intensive Web Services

Quirino Zagarese, Gerardo Canfora, Eugenio Zimeo
Department of Engineering
University of Sannio
Benevento, ITALY
{quirino.zagarese, gerardo.canfora, eugenio.zimeo}@unisannio.it

Françoise Baude
INRIA Sophia Antipolis Méditerranée
CNRS, I3S, Université de Nice
Sophia Antipolis, FRANCE
francoise.baude@inria.fr

***Abstract*—Improving performance of Web services interactions is an important factor to burst the adoption of SOA in mission-critical applications, especially when they deal with large business objects whose transfer time is not negligible. Designing messages dynamic granularity (offloading) is a key challenge for achieving good performances. This requires the server being able to predict the pieces of data actually used by clients in order to send only such data. However, exact prediction is not easy, and consequently lazy interactions are needed to transfer additional data whenever the prediction fails. To preserve semantics, lazy accesses to the results of a Web service interaction need to work on a dedicated copy of the business object stored as application state. Thus, dynamic offloading can experience an overhead due to a prediction failure, which is the sum of round-trip and storage access delays, which could compromise the benefits of the technique.**

This paper improves our previous work enabling dynamic offloading for both IN and OUT parameters, and analyses how attributes copies impact on the technique, by comparing the overheads introduced by different storage technologies in a real implementation of a Web services framework that extends CXF. More specifically, we quantitatively characterize the execution contexts that make dynamic offloading effective, and the expected accuracy of the predictive strategy to have a gain in term of response time compared to plain services invocations. Finally, the paper introduces the Attribute Loading Delegation technique that enables optimized data-transfers for those applications where data-intensive multiple-interactions take place.

***Keywords*-web-services; performance; middleware; data-intensive services; adaptation**

I. Introduction

Whilst Web services are becoming the standard de facto for improving interoperability among software components distributed over different organizational entities, their performances still remain an issue that limits the adoption of SOA in mission-critical and data-intensive applications. In particular, data-intensive analysis and computation have been defined as the trunk of the "Fourth Paradigm of Research" [1], because of the increasing number of scientific areas where petabytes of data are produced every day and need to be processed. A key challenge in the design of data intensive services is the definition of the right operations grain for services interfaces to achieve the best compromise between the number of exchanged messages and the size of each message.

Until today, design heuristics have been adopted to reduce the number of interactions on the network by designing coarse grained interfaces. Current web-services specifications and frameworks implementations exploit an implicit eager loading technique, since both arguments passed to services operations and results returned back to clients are respectively packed inside a single request message and a response one, thus leading to serious performance issues when the complexity and the size of such parameters grow. Provided that useful data must be transferred in any case, the main problem with the normal approach is the risk of transferring useless data between Web services and clients.

To overcome this problem, a common approach consists of redesigning the structure of the involved entities and splitting the initial operation into a set of more specific and finer grained ones. This makes it possible to avoid very large messages, at the cost of an additional design effort. Moreover, the operations exposed in the service interface tend to be less meaningful, since they do not directly map to domain functions, and a higher number of messages is required to complete the same task.

A different solution consists of decoupling the static structure of the exchanged messages from the way such messages are dynamically transferred. As long as the web-service semantics are met, it is possible to keep meaningful interfaces and separately handle performance issues. In this direction, defining a dynamic message granularity is a key challenge for achieving good performance without any additional burden for the designer. This requires the server being able to predict the data used by clients in order to send only such data.

However, exact prediction is not easy to achieve in general-purpose environments, and consequently lazy interactions are needed to transfer additional data when prediction fails [2]. To preserve the correct semantics, lazy accesses to the results of a Web service interaction need to work on dedicated copies of business object attributes that are stored on the server [3].

Previous works focus on client-server interactions and do not address scenarios where several endpoints exchange both IN and OUT complex objects. Moreover, reliability issues, that may compromise the applicability of the approach, have

not been analysed.

This paper presents a novel architecture (conceptually extending the one from our previous works), which is symmetric and completely decouples service-invocation semantics from data-transfers optimization. Such architecture eases the design of data-intensive scenarios, where large datasets can be virtually moved among endpoints, without compromising the overall performances.

We also discuss about the need to provide attributes copies persistence, in order to cope with failures during the offloading process, and we analyse the impact of server-side copies on dynamic offloading by comparing the overheads introduced by different storage technologies in a real implementation of a Web services framework that extends CXF. The comparison provides important information about the execution contexts (in terms of client behavior, network technology and framework implementation) that make dynamic offloading effective. The analysis highlights the expected accuracy of the predictive strategy to have a gain in term of response time compared to plain web-services invocations. Finally, we introduce the *Attribute Loading Delegation* technique, that reduces the number of attribute data-transfers to the needed ones, when multiple endpoints interact.

The rest of the paper is organized as follows. Section II describes the middleware architecture of our framework supporting object offloading. Section III discusses the benefits of offloading when a delegation schema is adopted as a multi-parties interaction style. Section IV presents the main aspects of the framework implementation and discusses some different technologies to implement the repository for storing objects copies. Section V analyzes the critical path of a typical service invocation in the presence of offloading to identify the main overheads introduced by the implementation of the technique and, in particular, the impact of objects copies. Section VI analyzes the performance in terms of the number of accessed attributes that makes offloading effective, by varying the technologies adopted for implementing the repository, the size of business objects and the characteristics of the communication channel. Section VII, discusses some related work and technologies. Finally, section VIII concludes the paper and introduces future work.

II. Middleware architecture

To address the problem of offloading data that will be used by clients with low probability, in [2], we have proposed a middleware that extends existing frameworks for Web services interactions with features for supporting transparent access to offloaded data. In this paper, we extend the architectural model proposed in the previous papers to support offloading both for request parameters and return values. Figure 1 describes this architecture by using a structural view that highlights symmetry and independence between service semantics and loading strategies, to fully support *Dynamic Object Offloading*.

A service-invocation scenario can clarify the role of each architectural component. When the client application makes a service invocation, the request is handled by the *OutgoingRequestInterceptor* (ORI), that is responsible of modifying the outgoing message, so that the server will be able to recognize the middleware-enabled client and consistently enact the service invocation. The ORI explores the structure of a call argument, for each one creates a list containing the corresponding attribute names and uses this list to query the *LoadingStrategy* (LS) component (no. 1 and 2 on the diagram). The query is performed by submitting the list of the attributes names and the result will be the list of the attributes that can be made available for lazy access (offloading candidates). LS is an abstract component and the responsibility to define the actual attribute-selection criteria is left to its implementations.

A pure lazy strategy will always return the input list as it is; a pure eager one will return an empty list; a mixed approach could offload only those attributes exceeding a predetermined size. The LS also provides update operations, so that the remote endpoint may notify it about which attributes have been actually used and which have not. This information can be exploited to design a strategy that adapts itself to endpoints behaviour, as reported on in [3].

Once the list of the attributes to be offloaded has been obtained, the ORI serializes them to the *OffloadingRepository* (OR), which will make them available for lazy access (3, 4). The request is then forwarded to the server (5) and handled by the *IncomingRequestInterceptor* (IRI), that is responsible of unwrapping it and hiding the applied loading strategy (6, 7), by means of the *ProxyManager* (PM). More specifically, for each call argument a proxy is instantiated before the actual invocation gets started. During the execution of the operation, whenever a getter method declared by one of the actual parameters is invoked, the proxy will verify if the corresponding attribute has already been loaded and possibly retrieve it from the remote OR (8, 11).

To achieve a higher level of flexibility, we have adopted

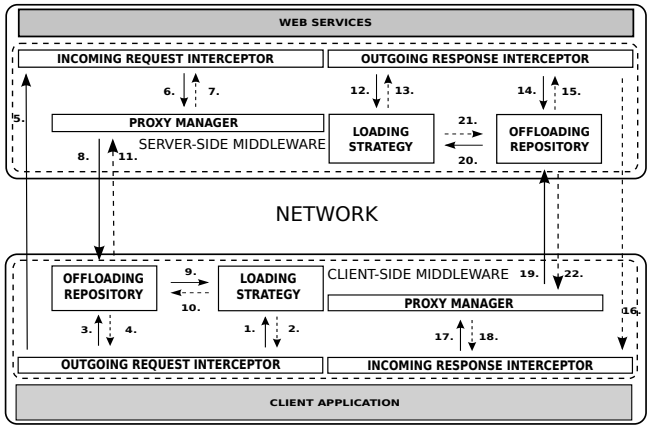


Figure 1. Components interations in an invocation scenario

an extensible *Dynamic Proxing System*, that enables the design of advanced attribute retrieval strategies. In addition to eager or lazy serialization, such system can be exploited to design retrieval strategies that asynchronously prefetch some attributes, in order to reduce delays during attributes retrievals. In this scenario, while the main thread executes the endpoint business logic, the retrieval strategy downloads the offloaded attributes. Once the main thread tries to access to one of such attributes, it may have to wait because the corresponding XML fragment has not been downloaded yet. Of course, the attributes transfer order will impact the perceived performances and the strategy could be designed in order to adapt itself to the attributes access order performed by the business logic. A specific proxy implementation may also insert additional data, concerning the actual usage of the attributes, inside the request, so that the client-side OR can update the LS, in an adaptation perspective (9, 10).

After the server-side method returns, the same process takes place to handle the response message: the *OutgoingResponseInterceptor* queries the local LS (12, 13), serializes the selected attributes to the OR (14, 15) and forwards the message to the client (16). Finally, the *IncomingResponseInterceptor* will instantiate a proxy for the result, by means of the client-side PM.

III. Attribute loading delegation

In this section, we explore the implication of offloading onto the delegation pattern, that becomes meaningful when the application scenario involves more than two endpoints, with the aim of implementing a multi-party transfer. Figure 2 depicts a simple scenario, where client *C* invokes *op1* from service *S1*, that in turn may invoke *op2* from service *S2*, in order to execute its own business logic. Both operations accept a sample object *O*, containing attributes *X*, *Y* and *Z*.

Let us assume that *S1* accesses attribute *X*, evaluates its value and possibly invokes *op2*. For the sake of clarity, we suppose each endpoint employs a pure-lazy loading strategy, when serializing arguments for services calls. When the first invocation takes place, the middleware, according to the loading strategy, saves *X*, *Y* and *Z* inside the client offloading repository. The execution of *op1* starts and eventually leads to an access to attribute *X*, thus generating a middleware request (interactions 1.1 and 1.2). After the *X* attribute has been retrieved, *S1* invokes *op2*; it is worth to note that the actual argument *O* is now an incomplete object, since attributes *Y* and *Z* are still stored on the client's offloading repository. Hence, the middleware should be able to distinguish between a plain object and one coming from an offloading process.

Before the invocation of *op2* starts, the middleware saves attribute *X* inside the *S1* corresponding offloading repository. Finally, let us suppose that *S2* tries to access to attributes *X*, *Y* and *Z*: in this case, the middleware will issue one request to the offloading repository connected to *S1* (2.1, 2.2),

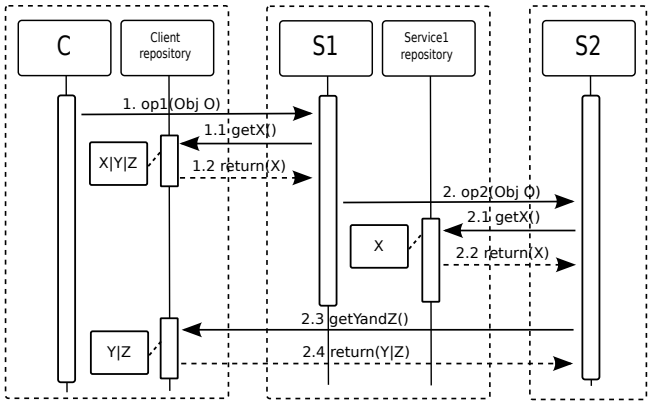


Figure 2. Attribute loading delegation scenario

and two more requests to the client's offloading repository (summarized as interactions 2.3 and 2.4).

One important consideration concerns the possibility to share the offloading repositories, among several endpoints. Consider the following example (Figure 3), where a set of services performs several computations over the same large dataset. This is what actually happens in the scientific workflows area [5]. In this contexts, the datasets can grow up to TBs or even PBs, thus they cannot be passed as web-services parameters, and fine-grained operations, to fetch small parts of them, must be provided. Also in the analysis of this case, we consider a pure-lazy loading strategy.

The *DatasetProvider* is the service supplying the access to the dataset, by means of the *getData* operation. The whole dataset structure is detailed inside the WSDL describing such service, according to the contract-first development of the Worker Services. This is a key point, since, in a standard context, each access to a dataset portion should have been designed as an explicit web-service call; in this case, each worker has a complete vision of the dataset structure and accesses to its portions are implicitly carried out by the middleware.

When *WorkerService1* (WS1) requests the dataset (interaction 1 in Figure 3), the provider-side middleware layer will offload it to the offloading repository (2, 3). Such repository can be easily shared with the worker services, since the middleware automatically exposes it as a service. The worker-side middleware instances need only to know

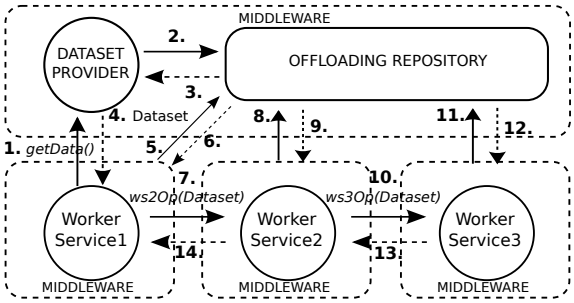


Figure 3. A shared offloading repository for data-intensive workflows

the location of such repository, in order to correctly wire it to the local components.

WS1 receives an empty Dataset instance (4) and its middleware layer downloads the needed attributes when necessary (5, 6). Once it terminates its task execution, *WS1* passes the dataset placeholder to *WorkerService2* (WS2), by invoking the *ws2Op* operation (7). Finally, the corresponding middleware instance retrieves the needed attributes and passes the placeholder on, down to the end of the workflow.

IV. Middleware implementation

We have implemented a Java-based prototype of the architecture presented in section II, which takes advantage of the message interception features offered by the Apache CXF framework [6]. We have added interceptors both to server and client sides, in order to enable the offloading process without the need to add any line of code during the service implementation.

The main responsibility of the *OutgoingMessagesInterceptors* is inserting middleware-specific SOAP headers that make the receiving endpoint aware of (a) which are the offloaded attributes and (b) how each of them can be retrieved. The former is needed to make the PM able to correctly instantiate a proxy for any parameter containing an offloaded attribute. Since such attribute is not present inside the received XML document, after the unmarshalling process, the corresponding property inside the generated object will be set to *null*. Hence, such information is needed in order to distinguish between a *null* value due to the offloading process and a real *null* value. The latter is needed to effectively retrieve each offloaded attribute since, as described in section III, in some scenarios, some offloaded attributes may not be stored on the endpoint that actually sent the SOAP message.

The *IncomingMessagesInterceptors* retrieve the information provided by the sending endpoints and replace the unmarshalled entities with the corresponding proxies. It is worth to note that the IRI will enact the offloading process only if it detects that the incoming request has been sent from a client that employs our middleware, thus keeping the service suitable for any kind of client.

We also mentioned that the middleware should be able to distinguish between a plain object and one coming from an offloading process. For this purpose, our solution prescribes a specific interface defining the following hook methods:

- *onPlainINParam(OffloadingContext oc)*
- *onProxyINParam(OffloadingContext oc)*
- *onPlainOUTParam(OffloadingContext oc)*
- *onProxyOUTParam(OffloadingContext oc)*

First and third methods should be implemented to respectively handle plain IN and OUT parameters; the second and forth ones should be implemented to handle IN and OUT parameters generated by an offloading process that took place on a remote host. In each case, the middleware delivers an *OffloadingContext* instance, containing a reference to the actual parameter, as well as entry points to main components like the *LoadingStrategy* and the *OffloadingRepository*. The *OffloadingContext* also provides all the meta-data from the SOAP header that is needed to correctly handle entities generated by offloading processes.

To create a proxy for each IN or OUT parameter, the PM exploits the dynamic code generation features provided by the *Javassist* library [7]. It dynamically creates a new class that extends the type of the parameter and overrides every getter method. The PM can be extended by implementing the *ProxyHandler* interface which offers three hook methods:

- *onConstruct(ProxyContext pc)*
- *onGet(ProxyContext pc)*
- *onDestroy(ProxyContext pc)*

The first is invoked right after a proxy is instantiated and should be implemented in order to setup an attribute retrieval strategy. The second gets called every time a getter method is invoked. Here the specific implementation may collect data about the actual attributes usage and send it to the remote endpoint. The last method is invoked when the garbage collection of a "proxied" parameter takes place, since each proxy also overrides the *java.lang.Object.notify()* method. A well designed strategy should take advantage of this extension point to inform the remote OR that it is safe to discard the corresponding offloaded attributes. When one of the mentioned events takes place, a *ProxyContext* instance gets passed to the *ProxyHandler* implementation, so that the strategy can easily inspect and possibly modify the state of both the proxy and proxied instances.

Finally, the OR plays a key role in the retrieval strategy, since its implementations can employ learning techniques, in order to achieve better performances. When a proxy sends a request for an attribute, the OR will respond by sending a name-value pairs list. An OR implementation may thus be designed in order to infer attributes co-usage relationships: in this perspective, if it is observed that a proxy requesting attribute *X*, in most of the executions has also requested attributes *Y* and *Z*, such strategy may decide to return all of them inside a single interaction.

The OR should provide attributes persistency for the middleware to be fault-tolerant. To this end, we have implemented both a MySQL-based OR and a MongoDB [8] one. MongoDB is a scalable, high-performance, open-source, NoSQL database that can store chunks of any size. It provides features like auto-sharding and asynchronous data replication that make it really fulfilling for our purposes.

V. Critical path analysis

The scenario described in section II outlines an execution path that extends a plain service invocation. To avoid a negative impact on service latency, the middleware layer should add as less overhead as possible. Moreover, reliability is a key concern, since a failure during the offloading process would lead to a violation of the service semantics with

reference to conventional interactions. Figure 4 shows an execution path from a server side perspective.

The whole critical path can be summarized into four main phases that identify contributes of the overall overhead. The first phase concerns the request message inspection and the proxy instantiation (interactions 5.2, 6, 7.1 and 7.2); the second phase is the actual invocation (7.3, 7.4); the third one concerns the selection of offloading candidates (12, 13). Here the strategy-designer should avoid the execution of complex algorithms that may slow down the whole process; strategies should work in background and take advantage of feedbacks from remote endpoints, to adapt the selection criterion. To achieve good performances, this phase should be implemented as a lookup operation that fetches the resulting state of a background running algorithm.

The fourth phase, the offloading enactment (14, 15), is the most critical from both the performance and reliability points of view. First, the ORI serializes each attribute, as it would happen for a plain service call; afterwards, it sets up a list containing the attribute name-value pairs of the offloading candidates and submits it to the OR. This component is responsible for making each pair univocally retrievable by assigning a unique id to it. One simple and well-performing way to make each pair available for later access, consists of keeping it in memory. However, as soon as the number of attributes grows, this approach becomes unfeasible, because of the huge amount of memory allocation. Moreover, if a failure occurs, all the offloaded attributes cannot be recovered and the service semantics cannot be preserved. To overcome this problem, server-side attributes persistence must be provided. Although this leads to additional overhead, it can be kept small by intervening at the architectural and at the technological levels. From the architectural perspective, the OR could assign an id to the pair, return the control to the ORI and asynchronously persist the data. Obviously, if the client tries to retrieve a pair (19.1-22.2), before it has been persisted, it must be blocked until the process completes. Key-value objects can be effectively persisted by employing NoSQL DBMSs, as discussed in [4]. For the considered execution path, the time needed to retrieve a business object is:

$$T_{ret} = T_{preCall} + T_{call} + T_{postCall} + T_{lazyAccess} \qquad (1)$$

The first term is defined as $T_{preCall} = T_{ori} + T_{iri} + L_s$, where the first two contributions represent the time needed by the *OutgoingRequestInterceptor* and the *IncomingRequestInterceptor* to process the SOAP message, while L_s stands for SOAP transport latency. T_{call} comprises the time needed to actually perform the operation call and serialize its result. $T_{postCall}$ is defined as follows:

$$T_{postCall} = T_{ls} + T_{orsi} + T_{irsi} + T_{proxy} + T_{off} + T_{et}$$

where T_{ls} is the time needed to query the *LearningStrategy*, T_{orsi} and T_{irsi} represent the message processing time

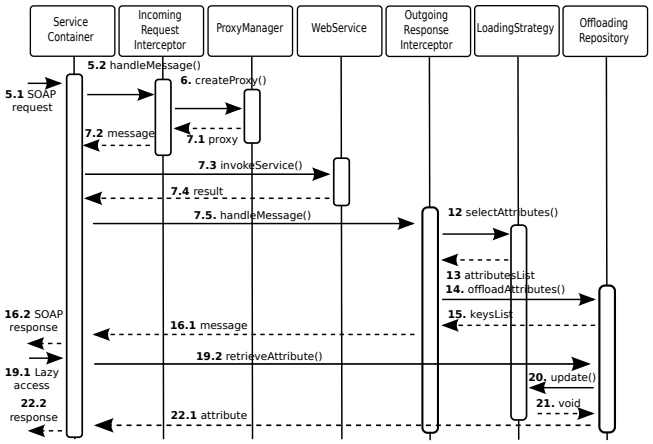


Figure 4. Server-side execution path for a middleware-enabled service invocation; interactions numbers are related to, and detail, the ones shown in Figure 1.

for the *OutgoingResponseInterceptor* and the *IncomingResponseInterceptor*, and T_{proxy} is the time needed to instantiate a proxy for the result on the client. To define T_{off} and T_{et}, we introduce a vector $S = \{s_1, s_2, \ldots, s_n\}$, where the generic element s_i represents the size of the corresponding attribute in the business object, and a binary vector O, where the generic element o_i is defined as follows:

$$o_i = \begin{cases} 1 & \text{if attribute } i \text{ is an} \\ & \text{offloading candidate} \\ 0 & \text{otherwise} \end{cases}$$

T_{off} is the time needed to save the offloading candidates in the *OffloadingRepository*, and is defined as:

$$T_{off} = \sum_{i=1}^{N} T_w(s_i) o_i$$

where $T_w(s_i)$ expresses the time needed by the *OffloadingRepository* to save an attribute with size s_i. Finally, T_{et} is the eager transfer time and is defined as follows:

$$T_{et} = \frac{\sum_{i=1}^{N} \overline{o_i} s_i}{Th_s}$$

where Th_s stands for SOAP transport throughput and $\overline{o_i}$ means that an offloading candidate is never transferred in eager mode. The last term in (1) is the overall time needed to serve lazy accesses to attributes:

$$T_{lazyAccess} = 2L_s \sum_{i=1}^{N} a_i o_i + \frac{\sum_{i=1}^{N} a_i o_i s_i}{Th_s} + \sum_{i=1}^{N} T_r(s_i) a_i o_i$$

where T_r is the time needed to load an attribute exhibiting size s_i from the *OffloadingRepository* and a_i is an element of the binary vector A, defined as follows:

$$a_i = \begin{cases} 1 & \text{if client uses attribute } i \\ 0 & \text{otherwise} \end{cases}$$

T_r and T_w are strictly related to both the technology and the loading approach adopted for the repository implementation. They represent key parameters for the evaluation of the middleware implementation. In a pure-lazy strategy,

the T_{off} contribution is maximized, since all the attributes are offloading candidates. Moreover, the $T_{lazyAccess}$ term is also maximized, since any attribute access leads to a request-response cycle. The pure-lazy approach can, thus, be considered as a worst-case from the T_{off} and $T_{lazyAccess}$ perspectives.

It is worth to evaluate which are the necessary conditions to achieve a performance gain, in terms of T_{ret}, when a pure-lazy strategy is employed, compared to a plain call that does not employ our middleware. If we assume that all the attributes in the business object have the same size S, $T_{ret(plain)}$ can be defined as follows:

$$T_{ret(plain)} = T_{call} + 2L_s + \frac{NS}{Th_s}$$

where N is the total number of attributes in the business object. In the pure-lazy case, T_{ret} is defined as:

$$T_{ret} = T_{call} + 2L_s(N_a + 1) + NT_w(S) + N_aT_r(S) + \frac{N_aS}{Th_s} + T_{mw}$$

where $N_a = \sum_{i=1}^{N} a_i$ and T_{mw} comprises the interceptors SOAP message processing time, T_{ls} and T_{proxy}. Finally, we can express the necessary conditions to achieve a performance gain in terms of N_a as follows:

$$N_a < \frac{NT_w(S) - T_{mw} + \frac{NS}{Th_s}}{2L_s + T_r(S) + \frac{S}{Th_s}}$$

It is worth to note that, for a generic strategy,

$$N_a = N_{eager} + N_{lazy}.$$

N_{eager} expresses the number of attributes, among those the client actually uses, that are transferred during the first request-response cycle; N_{lazy} represents those that are transferred on demand, by issuing further requests. For a pure-lazy strategy, $N_a = N_{lazy}$; for an ideal strategy, $N_a = N_{eager}$, since the endpoint will send only the useful attributes, inside the first request-response cycle; in the latter case, $T_{ret} = T_{ret(plain)} + T_{mw}$ and a performance gain is achieved if:

$$N_a < N - \frac{T_{mw}Th_s}{S}$$

For a real strategy, both N_{eager} and N_{lazy} will be non-null. The accuracy of the strategy is a key-point and can be evaluated by taking into account two aspects: (a) how many useful attributes were transferred during the first request-response cycle (N_{eager}/N_a), and (b) how many of the transferred attributes were actually used by the client ($N_a/N_{transferred}$).

VI. Performance evaluation

In [3], we have shown that dynamic object offloading can lead to a considerable performance gain, in terms of the time needed by the client to retrieve a business object, if such client exhibits a predictable behaviour. On the other hand, we have shown that a simple learning technique, as well as a random strategy, can still outperform pure-eager or pure-lazy loading, depending on object granularity, if the client randomly decides to access the generic attribute i. Here, we are interested to understand when dynamic offloading is useful, regardless of the learning strategy. For this purpose, we do not focus on the value of T_{ret} itself; we evaluate the highest value of N_a that guarantees a lower T_{ret}, compared to a plain service invocation that does not employ our middleware.

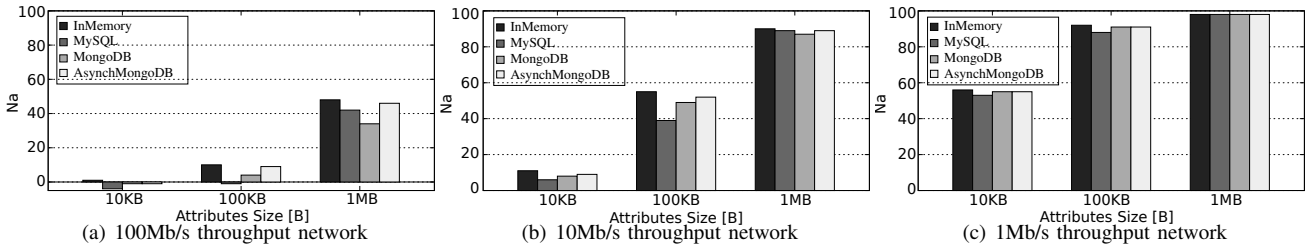
We consider a pure-lazy strategy, since it can be considered as a realistic worst case, as shown in section V. In the same section, we have mentioned that the time T_w, needed to store an attribute inside the repository and the time T_r, needed to retrieve it, are strictly related to the technology employed to realize the *OffloadingRepository*; for this reason, we have structured the evaluation process into four phases, describing the behaviour of four different repository implementations:

- InMemory repository
- R-DBMS managed persistent repository
- NoSQL DBMS managed persistent repository
- NoSQL DBMS managed persistent repository with asynchronous I/O

In the first phase, we have computed the average T_w and T_r, for an *OffloadingRepository* that saves and retrieves attributes from an in-memory hashtable. Such solution can lead to excellent performances, but it is neither scalable nor reliable. In the second phase, we introduce attributes persistency, by saving them inside a MySQL-managed database. Such database consists of a single table without any relationship to other entities, thus we do not need all the consistency related features provided by RDBMSs. Therefore, in the third phase we evaluate an *OffloadingRepository* implementation based on the NoSQL, documented-oriented, MongoDB DBMS. Finally, in the fourth phase, we try to minimize the impact of T_w, by submitting the attributes to be saved to a worker thread that uses a MongoDB store. In this case, the *OffloadingRepository* generates a unique id for each offloading candidate and submits the candidates to the worker thread that will asynchronously persist them, thus relieving the critical path of the I/O contribution.

Each T_w and T_r value has been computed as the average of 1000 invocations of a web-service that returns a business-object containing 100 fixed-size attributes. Each phase has been repeated for three different attributes sizes - we choose 10KB, 100KB and 1MB as meaningful sizes, as explained in [2] - in order to show how object granularity impacts on performances. Finally, all the tests have been run on a machine equipped with a Intel Core i7-2630QM (2.0Ghz, 6MB L3 cache), 4GB DDR3 Memory and a Toshiba 5400RPM, 8MB cache hard-drive.

Figure 5(a) shows the comparison between the *OffloadingRepository* implementations, for the considered object granularities, when transferring the attributes over a high throughput network (100 Mb/s). In this case, unless a 1MB granularity is considered, pure-lazy loading can nearly never lead to an improvement of performances. Anyway, it is interesting to observe that the AsynchMongoDB implementation



(a) 100Mb/s throughput network (b) 10Mb/s throughput network (c) 1Mb/s throughput network

Figure 5. N_a (maximum number of accessed attributes to keep an improvement of T_{ret}) values for different Offloading Repositories tested at different SOAP-level throughputs and fixed 30ms SOAP-level latency

can obtain results that are close to the InMemory one.

Figure 5(b) shows a more realistic Internet-based scenario. [1] We can see that the N_a value is still around 10%, if we consider a 10KB granularity, but things definitely change if we observe the 100KB one. The difference between the various implementations becomes wider: MySQL reaches 39.53%, while AsynchMongoDB gets very close to the InMemory implementation (respectively 52.37% and 55.32%). Despite these values seem still quite low, they are enough to achieve a not negligible improvement in areas like the scientific workflow context, we have introduced in section III. In such scenario, each service is likely to use only a small part of the entire dataset and dynamic offloading is an enabling approach to virtually move large datasets among services, while keeping strong typing of attributes and contract-first development. If we consider the 1MB granularity, we see the values reach the 90%, thus making offloading effective in most situations.

Finally, figure 5(c) shows how the considered implementations perform when data is moved over a low throughput network (1Mb/s). Such scenario may be perceived as unusual, but helps understanding how offloading can improve performances in presence of network bottlenecks, when dealing with world-wide distributed service-oriented systems. Again, the $size/throughput$ ratio matters and the N_a value increases along with the object granularity. In all of our tests, the AsynchMongoDB implementation outperformed both synchronous MySQL and MongoDB ones, but the gap becomes negligible in low-bandwidth networks, since transfer-time dominates storage overhead. Synchronous MongoDB outperforms MySQL in all tests characterized by an object granularity of 10KB and 100KB, while MySQL performs better in those characterized by a 1MB object granularity. This last result is due to a better T_r value for MySQL (-18%) as well as a better T_w (-51%). On the other hand, the NoSQL paradigm seems to be more effective when reading smaller chunks (-15% for 100KB chunks and -9% for 10KB ones) as well as for the writing phase (-59% for 100KB chunks and -83% for 10KB ones). The described results may suggest that offloading will not be useful as soon as the Internet average speed will be comparable to that characterizing current LAN networks. Such conclusion would be wrong, since the amount of data produced by modern industrial and scientific systems is growing much faster than the internet speed. Considered that the $size/throughput$ ratio heavily impacts on the overall performances, we believe that offloading will become even more effective for tomorrow systems. The comparison between the different persistence approaches suggests that a NoSQL store can lead to better performances, as long as a scalable hardware infrastructure is provided. This was not our case, hence performances degrade when the indexes do not fit in RAM, as for the 1MB attributes test. However, we believe that NoSQL DBMSs better fit this kind of application, since they are designed for cloud-based environments and are intrinsically scalable [10]. Finally, our tests outline a lower bound in terms of N_a due to the pure-lazy approach, but better performances can be easily achieved by employing a light-weight learning strategy, as shown in [3].

[1] the considered 10Mb/s throughput corresponds to the average Internet speed at the time we are realizing this work [9]

VII. Related work

There are several attempts aimed at improving web-services performances and they can be summarized in four main categories. The first category concerns message compression. In [11], a wide description of the main efforts concerning such approach is provided and a novel SOAP compression technique, based on the generation of a single custom pushdown automaton, obtained from the service WSDL, is proposed.

The second category concerns response caching. This technique has been extensively employed in the mobile computing area, as shown in [12], where a cooperative caching system is proposed, in [13], where automatic prefetching, by means of a sequence prediction algorithm, is provided, and in [14], where a perfomances improvement is obtained by exploiting differential caching. Meaningful results are also described in [15], where cryptographic hashing is employed in order to detect similarities between responses, as well as in [16], where a framework providing caching for mashups is presented.

The third category is focused on SOAP processing optimization. In [17], a lightweight checkpointing mechanism is proposed, in order to relieve the memory and CPU load, during SOAP deserialization. In [18], performances are

boosted by exploiting schema-specific parsing, while a useful technique based on the reuse of serialization information is described in [19].

The last category concerns advanced interaction patterns. In [20], an exchange pattern that enables demand-driven transmission of SOAP attachments, is presented. In [21], support to asynchronous invocations and continuations, by means of future variables, is provided. Habich *et al.* have foreseen the need to decouple service implementation and data transfers in [22], but they cannot achieve full transparency, since plain clients are prevented from invoking those services that employ their framework. Finally, two interesting approaches are described in [24] and [25], where the data-transfers in workflows are addressed. The former takes advantage of service-proxies to share data, without involving the workflow engine; the latter introduces the *state* concept in workflows, to share intermediate data among successive services. Both solutions have proven to lead to good performances, but do not provide attribute-grained transfers, thus preventing the possibility to employ more advanced techniques.

VIII. Conclusion and future work

In this paper, we have introduced a symmetric architecture that supports *Dynamic Object Offloading* for both IN and OUT parameters, and decouples service-invocation semantics from data-transfers optimization. We show how such decoupling can enable new service-oriented scenarios, where large datasets can be virtually moved among endpoints, by taking advantage of the delegation approach. Finally, we precisely describe when it is useful to employ offloading, by evaluating the actual overhead added by our framework and comparing different technological solutions for the *OffloadingRepository*, which is responsible of handling the attributes copies, in order to keep service semantics. Future work will be mainly aimed at exploring learning-based solutions, in order to precisely indicate which ones give the best predictions, while keeping good performances and system scalability. Moreover, we want to investigate about repository deployment solutions: we believe that is possible to obtain better performances by constructing an overlay network of repositories, upon services endpoints, thus exploiting network topology and taking advantage of the delegation mechanism. Finally, we are planning to experiment our framework inside the PLAY project [23]. In this context, our middleware will be employed to optimize the interactions between a web-application and a Web services API that exchange large datasets of RDF-encoded events.

References

[1] T. Hey, S. Tansley, and K. Tolle, Eds., The Fourth Paradigm: Data-Intensive Scientific Discovery. Redmond, Washington: Microsoft Research, 2009.

[2] Q. Zagarese, G. Canfora, and E. Zimeo, "Employing dynamic object offloading as a design breakthrough for soa adoption," in ICSOC, 2011, pp. 610–617.

[3] Q. Zagarese, G. Canfora, and E. Zimeo, "Dynamic object offloading in web services," in SOCA, 2011, pp. 58–65.

[4] N. Leavitt, "Will nosql databases live up to their promise?" Computer, vol. 43, pp. 12–14, February 2010.

[5] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," SIGMOD Rec., vol. 34, pp. 44–49, September 2005.

[6] (2012, Feb.) Apache cxf. [Online]. Available: http://cxf.apache.org/

[7] S. Chiba, "Javassist: Java bytecode engineering made simple." Java Developers Journal, vol. 9, January 2004.

[8] K. Banker, A Document Database for the Modern Web. Manning, 2011.

[9] (2012, Feb.) Net index by ookla. [Online]. Available: http://www.netindex.com/

[10] R. Cattell, "Scalable sql and nosql data stores," SIGMOD Rec., vol. 39, pp. 12–27, May 2011.

[11] C. Werner, C. Buschmann, Y. Brandt, and S. Fischer, "Compressing soap messages by using pushdown automata," in ICWS, 2006, pp. 19–28.

[12] H. Artail and H. Al-Asadi, "A cooperative and adaptive system for caching web service responses in manets," in ICWS, 2006, pp. 339–346.

[13] A. Göb, D. Schreiber, L. Hamdi, E. Aitenbichler and M. Mühlhäuser, "Reducing user perceived latency with a middleware for mobile soa access," in ICWS, 2009, pp. 366–373.

[14] M. S. Qaiser, P. Bodorik, and D. N. Jutla, "Differential caches for web services in mobile environments," in ICWS, 2011, pp. 644–651.

[15] W. Li, Z. Zhao, K. Qi, J. Fang, and W. Ding, "A consistency-preserving mechanism for web services response caching," in ICWS, 2008, pp. 683–690.

[16] O. Al-Haj Hassan, L. Ramaswamy and J. A. Miller, "Mace: A dynamic caching framework for mashups," in ICWS, 2009, pp. 75–82.

[17] N. Abu-Ghazaleh and M. J. Lewis, "Lightweight checkpointing for faster soap deserialization," in ICWS, 2006, pp. 11–18.

[18] W. Zhang and R. A. van Engelen, "High-performance xml parsing and validation with permutation phrase grammar parsers," in ICWS, 2008, pp. 286–294.

[19] F. Lelli, G. Maron, and S. Orlando, "Improving the performance of xml based technologies by caching and reusing information," in ICWS, 2006, pp. 689–700.

[20] S. Heinzl, M. Mathes, T. Friese, M. Smith, and B. Freisleben, "Flex-swa: Flexible exchange of binary data based on soap messages with attachments," in ICWS, 2006, pp. 3–10.

[21] G. Tretola and E. Zimeo, "Extending web services semantics to support asynchronous invocations and continuation," in ICWS, 2007, pp. 208–215.

[22] D. Habich, S. Preissler, W. Lehner, S. Richly, U. Aßmann, M. Grasselt, and A. Maier, "Data-grey-boxweb services in data-centric environments," in ICWS, 2007, pp. 976–983.

[23] (2012, Feb.) Play project. [Online]. Available: http://www.play-project.eu/

[24] A. Barker, J. Weissman, and J. van Hemert, "Reducing data transfer in service-oriented architectures: The circulate approach," Services Computing, IEEE Transactions on, vol. PP, no. 99, p. 1, 2011.

[25] D. Zhang, P. Coddington, and A. Wendelborn, "Web services workflow with result data forwarding as resources," Future Gener. Comput. Syst., vol. 27, no. 6, pp. 694–702, Jun. 2011.