Web Service Execution Streamlining

Costas Vassilakis[1], George Lepouras[2], Akrivi Katifori[3]

[1] Department of Computer Science and Technology, University of Peloponnese, Terma Karaiskaki 22100, Greece (costas@uop.gr)

[2] Department of Computer Science and Technology, University of Peloponnese, (gl@uop.gr)

[3] Department of Informatics and Telecommunications, University of Athens (vivi@di.uoa.gr)

ABSTRACT

Web services are functional, independent components that can be called over the web to perform a task. Besides being used individually to deliver some well-specified functionality, web services may be used as building blocks that can be combined to implement a more complex function. In such compositions, typically some web services produce results that are used as input for web services that will be subsequently invoked. In the execution schemes currently employed, web services producing intermediate results deliver them to some "coordinating entity", which arranges the forwarding of these intermediate results to web services that require them as input. In this paper we present an execution scheme that employs direct communication between producers and consumers of intermediate results. Besides performance improvement stemming from reduction of network communication, this scheme permits consumer web services to employ simpler authenticity and integrity verification algorithms on incoming parameters, when the producer web service is considered trustworthy.

Keywords: Web service synthesis, web service execution; optimization, streamlining

1. INTRODUCTION

According to the service oriented architecture paradigm, web services are offered by specific protocols and communicate over the internet, providing a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration [1]. Offering organizations advertise their services by enrolling them into publicly accessible registries, typically following the UDDI standard [2]. Clients locate the services of interest through these registries and invoke them, providing input parameters and obtaining the desired results.

In many cases, for the completion of a business transaction or for fully servicing a citizen's *life event* [3] a number of web services need to be combined in a fashion which demands results returned by a web service to be fed as input parameters to another. For example, in order to apply for a passport, the citizen must present a birth certificate; thus –at the information system level– the web service producing the birth certificate should be first executed and its output (the birth certificate) will be provided as an input parameter to the web service that records passport issuance applications.

Handling of such data flows between web services may be left to the client (Figure 1), who should invoke the first web service, collect the result and include it in the invocation of the second web service. In more complex cases, the client may need to coordinate an arbitrary number of web services, arranging for intermediate results to be forwarded to the appropriate consumer (e.g. a result returned from WS#1 may need to be forwarded to WS#3 and WS#4).

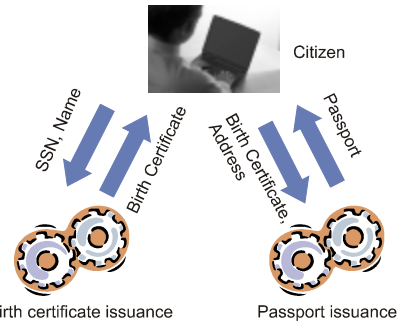

Figure 1. Client-Managed Service Composition

An alternative to client-managed data flow is the introduction of a *service aggregator* entity [4]. The service aggregator again publishes web services which are invoked by clients, but this time each such service corresponds to a "business transaction", not an individual operation. The service aggregator undertakes the tasks of (a) identification of the distinct services that must be invoked (in a static [5], dynamic [6], or semantics-based [7] fashion) and (b) their orchestration (arrangement of control flow and data flow between constituent services) [7], [8]. The service aggregator approach for the case of a citizen accessing a passport issuance service is illustrated in Figure 2.

While the web service aggregator approach reduces significantly the user's burden and the complexity in the client (through removing the need for web service identification, orchestration, and execution), the data flow between the individual web services still remains sub-optimal. More specifically, in both cases some

intermediate results (the birth certificate in the illustrated examples) are sent from the producing web service to the coordinating entity (the client or the web service aggregator), only to be subsequently forwarded to the consuming web services. If such results were directly forwarded from the producing to the consuming web service, an extra transmission would be avoided, improving thus performance.

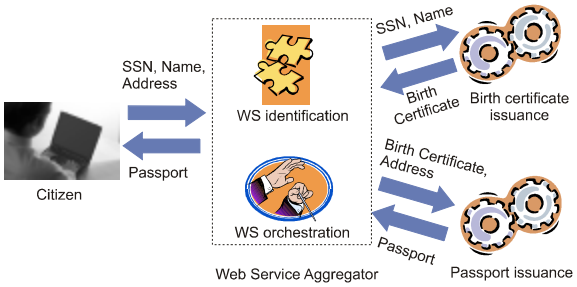

Figure 2. Web Service Aggregator Approach

An additional benefit from the direct communication is related to security and trust: consider the case that a consuming web service needs to verify the authenticity and integrity of an incoming parameter, such as the birth certificate. If the parameter were directly supplied by the producing web service and the organization offering the consuming service trusts the one offering the producing web service, then it suffices to verify the producer's authenticity, e.g. by verifying the source IP address. If, however, the parameter were supplied by the coordinating client or an aggregator entity, then such a check does not suffice, and document authenticity and integrity should be verified through more complex and computationally expensive methods, such as digital signatures and public key infrastructure [9]. Naturally, the provision of support for direct communication between producing and consuming web services should complement the normal invocation mode and not substitute it, since compatibility with clients not supporting the optimized scheme or simply wishing to retrieve the result should be retained.

The rest of the paper is organized as follows: section 2 presents related work on the areas of web service composition, orchestration and execution. Section 3 presents the proposed solution, illustrating the architectural elements and their interaction during the streamlined execution. Finally, section 4 concludes the paper and outlines future research directions.

2. RELATED WORK

For determining the web services that need to be invoked in the context of a business transaction or while servicing a life-event, three predominant approaches exist insofar. The eGov project [10] allows the composite service developer to draw simple web services from a pool of existing ones and define their execution flow and the data flow between services, creating thus a composite task; composite tasks can be themselves reused as building blocks for building other composite services. Technological frameworks, such as the web services composite application framework (WS-CAF [5]) also undertake such approaches. [6] proposes a less rigid approach, where the developers specify a schema for various aspects of the composite service, including structure of the flow, service definition, nodes for decision taking and event handling, processing of data and regions with transactional "all-or-nothing" semantics. The original schema may be altered at execution time, to tackle cases where constituent elements of the composite service have been modified since the definition of the composite service. In [7], a semantics-based service composition mechanism is presented, in which users of e-government services request the desired output (e.g. certificates, documents etc) and an ontology is employed to identify the web services that need to be executed to produce the requested result. The ontology includes full description of the input requirements and output types of each available web service, thus the execution order of the selected web can be derived on the basis of data flow requirements. A Web Services Composition Approach based on Software Agents and Context is finally described in [11].

Typically, web service composition methods focus on the creation of the service execution plan, and leave the actual execution to be performed by separate tools. For example, [5] employs the IONA Service Bus [12] for execution) while another approach would be to format the service execution plan according to the rules of a standard web service orchestration language, such as BPEL4WS ([13] or WSFL [14]) and delegate the execution responsibility to a web service orchestration engine [8]. A notable exception is [7], which includes a web service execution module, needed in this case to cater for selection between different concrete implementations (mainly for optimization purposes) and accommodation of the event-condition-action rules that are supported by the platform. The OntoGov project also includes an orchestration component, which again selects between different concrete implementations, taking into account jurisdiction issues in the context of e-government [15]. None of the orchestration engines made available by the industry or proposed by researchers addresses the issue of data streamlining between constituent web services of a composite service.

3. STREAMLINED WEB SERVICE EXECUTION

In order to accommodate streamlining of results, while maintaining the ability to invoke web services in the "traditional" fashion, an extra module is introduced in the service-oriented architecture, namely the *web service streamliner*. A separate instance of the web service streamliner should be installed by any organization wishing to include the extra functionality for the web services it offers, as illustrated in Figure 4.

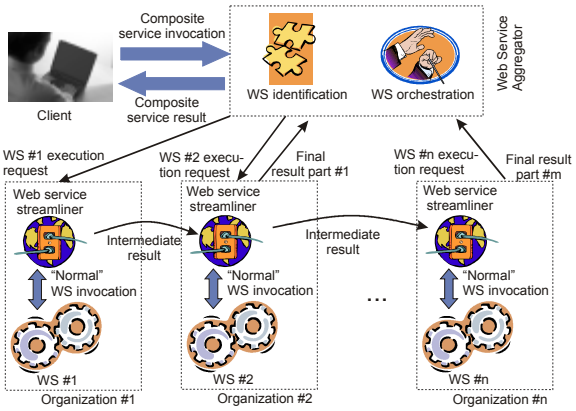

Figure 3. Web service streamlining

According to this enhanced architecture, the web service aggregator initially formulates the composite service execution plan, which involves the execution of web services WS_1, WS_2, …, WS_n, running at the premises of organizations O_1, O_2, …, O_n, respectively. Subsequently, for each web service that needs to be invoked, the web service aggregator sends a WS execution request to web service streamliner running at the corresponding installation. The web service streamliner undertakes the task of collecting parameter values that are yet unavailable (will be produced by other web services within the execution plan), invoking the local web service and forwarding its results to their intended consumers. Finally, the web service aggregator will collect the parts of the final result, assemble them into a reply message, and send it to the client that initiated the request. Note that clients and streamlining-unaware web service aggregators can still invoke individual web services according to the "traditional" paradigm, providing all input parameters and collecting the results, as illustrated in figures 1 and 2.

In the following paragraphs, the web service streamlining operation is described in more detail. Firstly, the web service execution request messages, sent by the web service aggregator to web service streamliners contain the following specifications:

1. the (local to the installation) web service to be executed.

2. for each input parameter (a *part* in the WSDL *input* message [2]), either a concrete value (which can be directly used) or a specification of the *value provider*, i.e. a particular web service execution that will provide the concrete value.

3. for each output parameter (a *part* in the WSDL *output* message [2]), a list of *value recipients* that this output parameter should be forwarded to. A value recipient is either a particular web service execution (being run in the context of the same composite web service) or the web service aggregator; the latter collects result parts that are part of the final reply, packs them into a web service response and sends it to the client that initiated the composite service execution. This scheme hides all streamlining complexities from the client, which interacts with the system using a single standard web service invocation.

The aggregator may transmit all requests as soon as the execution plan has been formulated, regardless of the data dependencies between constituent web services; the web service streamliners undertake the responsibility of deferring the execution of web services for which some input parameters are yet unavailable, as described in the following paragraphs.

When a web service streamliner receives such a request, it first checks if all input parameters for invoking the web service are available (i.e. specified as concrete values); in such a case, it proceeds with the execution of the specified web service (acting thus as a "normal" web service client) and collects its result. Subsequently, the result is analyzed into its constituent parts, and each part is forwarded to the appropriate recipients, as specified in the request message.

If, however, not all input parameters are available (i.e. for at least one parameter the message contains a *producer specification* rather than a concrete value), the web service streamliner stores the request into a *pending request queue*. The request will remain in the queue until the values for all input parameters have been collected from the designated producers, at which time the request is extracted from the queue and executed, as described above.

Note that there exists a possibility that some output parameter produced by a web service arrives at a web service streamliner *before* the request that will consume this parameter has been received. For instance, if the network link between the web service aggregator and organization #2 of Figure 3 is too slow or down, it is possible that request #1 is sent to organization #1 and processed, and its result is forwarded to organization #2 (through an alternative network link) before request #2 has arrived to the streamliner at organization #2. In such cases, the receiving streamliner stores the incoming value in an *unclaimed parameter repository*; each incoming request is always matched against the repository contents, and if any of the parameters therein were destined for the specific request, it is extracted and bound to the request. Entries in the unclaimed parameter repository may be evicted after a certain period of time, defined by the local administrator, to cater for cases that the composite service execution in the context of which the parameter has been produced has been cancelled, thus the parameter is no longer useful.

An issue that needs to be carefully addressed in this scheme is to enable web service streamliners to correctly correlate results of producing web services

with input parameters of consuming web services. Consider for example the case that, two citizens invoke "at the same time" the composite service depicted in Figure 4, with each citizen providing her own values for the SSN, name and Address parameters and expecting to receive her personal passport document. The pending request queue at organization #2 will contain two entries for execution of the passport issuance service, and the streamliner at organization #2 will receive two messages from the streamliner at organization #1 containing the respective parameter values; then the receiving streamliner should associate each incoming parameter value with the proper entry in the pending request queue. Similarly, the web service aggregator should be able to correlate each result returned by any streamliner with the corresponding composite service invocation, in order to return the result to the proper citizen.

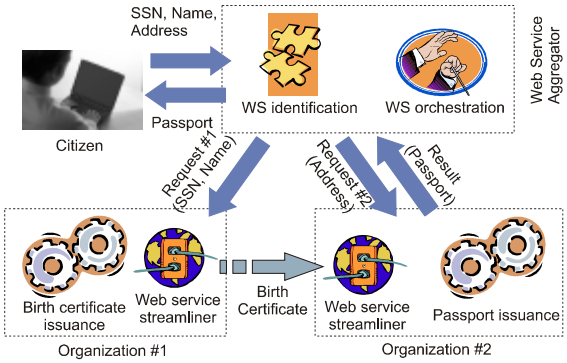

Figure 4. An Example Streamlined Composition

To enable web service streamliners to perform this correlation the following scheme has been adopted: for each distinct web service execution request, the web service aggregator generates a request identifier that is unique for the specific request; note that even in the presence of multiple web service aggregators sending execution requests to the same installations, the generation of unique identifiers is still possible, as described in [16]. The request identifier is included in each web service execution request. Furthermore, the *value provider* specification (used for input parameters that will be produced by other web services in the context of the composite web service execution) has the format *(provider_address, source_request_identifier)*, while the *value recipients* specification (used as a distribution list for output parameters) is a list of *(recipient_address, target_request_identifier, target_part_name)*. When the recipient is the aggregator, the *target_request_identifier* field in fact identifies the client request for which this result has been produced.

Under this scheme, when a web service streamliner receives a result from the invocation of the (local) web service, it firstly extracts the individual parts (output parameters) from the reply message. For each such part, the associated recipient list is traversed and, for each list node it constructs *parameter transfer* message *(providing_request_identifier, target_request_identifier, target_part_name, part_value)* which is sent to the web service streamliner running at the network address *recipient_address*. In this message, the *providing_request_identifier* is the identifier of the web service execution request that produced the result, *part_value* is concrete value of the output parameter, while the values for *recipient_address, target_request_identifier, target_part_name* are the corresponding elements from the recipient list entry.

Conversely, when a web service streamliner receives a *parameter transfer* message from a streamliner running at network address *sending_address* (*sending_address* is not included in the message body but the streamliner requests it from the network layer) it processes it as follows:

1. it determines if a web service execution request with identifier equal to *target_request_identifier* exists in the pending request queue. If not, the parameter transfer message is placed into the unclaimed parameter repository and the process terminates; otherwise, the service execution request record *SERR* is extracted and algorithm proceeds with the next step.

2. the input parameter list for SERR is scanned to locate the *target_part_name* designated in the parameter transfer message. If for this parameter a concrete value has been provided by the web service aggregator, or is value has already been provided by a previous parameter transfer message, then the current parameter transfer message is dropped.

3. If a value for the designated part is indeed waited for, then the *value provider* specification for this part is examined. More specifically, the *provider address* in this specification is matched against the *sending_address* (determined previously by querying the network layer) and the *source_request_identifier* in the specification is matched against the *providing_request_identifier* in the parameter transfer message. If either value is found to be different, then the parameter transfer message is rejected, and the algorithm terminates.

4. Finally, the particular input parameter of SERR is bound to *part_value*. If no more input parameters of SERR remain to be bound, then the web service specified therein may commence its execution.

The same algorithm is employed for matching parameters in the unclaimed parameter repository against incoming service execution requests.

Note that the third step in the previously described algorithm is not necessary for matching incoming parameters with requests, but is introduced for security purposes, serving as a safeguard against malicious entities that attempt to send counterfeit results to

streamliners. With this check present, a malicious entity trying to provide a bogus parameter value for a particular web service execution should be able to correctly determine (guess or eavesdrop) the request identifier of both involved web service execution requests (producer and consumer) *and* spoof [17] the network address of the legitimate producer. If request identifiers are carefully drawn from a large domain (e.g. 128 bits), similarly to the way HTTP session identifiers are selected [18], it will be infeasible for an attacker to correctly guess both request identifiers and thus deceive the receiving streamliner into accepting the counterfeit parameter. Anti-spoofing techniques [17] may also be employed in the network layer to further harden the defense against attacks of this type.

In order to clarify the execution procedure, consider the example illustrated in Figure 4, where the installation of organizations #1 and #2 run at addresses A1 and A2, respectively. The web service aggregator, after accepting the request from the citizen providing the values "1234567890", "Johnson John" and "Someplace 42" for the respective input parameters, formulates the following two requests, depicted in Table 1:

Table 1. Requests for the Passport Issuance Example

```
<req_id>10</req_id>
<serviceId name="Birth Certificate Issuance"/>
<input>
    <part name="ssn" concrete_val="1234567890"/ >
    <part name="name" concrete_val="Johnson John"/>
</input>
<output>
    <part name="BithCert">
        <recipients>
            <recipient addr="A2" reqId="35" part="cert"/>
        </recipients>
    </part>
</output>
```

```
<req_id>35</req_id>
<serviceId name="Passport Issuance"/>
<input>
    <part name="address" concrete_val="Someplace 42">
    <part name="cert" prov_addr="A1" prov_req_id="10"/>
</input>
<output>
    <part name="Passport">
        <recipients>
            <recipient addr="Aggregator" reqId="987654321"
                part="Passport"/>
        </recipients>
</output>
```

The first request (id = 10) is sent to the streamliner of organization #1, while the second one (id = 35) is sent to the streamliner of organization #2.

The request within organization #2 cannot be executed, since an input parameter value (cert) is missing, thus it is placed in the pending request queue. Request #1 however can be processed, thus the web service streamliner at A1 invokes the Birth Certificate Issuance service which returns a concrete birth certificate, which we will denote as *BirthCertificate_Value*. The streamliner at A1 should now forward this result to the streamliner at A2, as specified in the request; to this end, it formulates a message *(10, 35, cert, BirthCertificate_Value)*, which is sent to the target address *(streamliner at A2)*. Upon receiving this message, the streamliner at A2 locates the entry for the service request with id equal to 35 in the pending request queue, and verifies that a value for part *cert* is indeed expected; subsequently it checks that the parameter transfer message has indeed been received from network address A1 and that the providing request identifier in the message is actually equal to 10. Since both checks succeed, the value *BirthCertificate_Value* is bound to the input parameter *cert*. Now, all parameters for the invocation of the Passport Issuance web service are available, thus it is invoked and its result is collected and returned to the service aggregator, as specified in the recipient list for the *Passport* output part in the second request. The service aggregator collects the value and correlates it with the initial client request using the reqId value in the message; since the response to the client is now complete, the response is assembled and sent to the citizen, concluding the execution of the composite service.

CONCLUSIONS – FUTURE WORK

In this paper we have presented a method for streamlining the execution of web services that need to communicate in a "producer/consumer" fashion for the realization of a composite task. The proposed method eliminates unnecessary data transmissions, decreasing thus network load and improving performance; additionally it enables producers and consumers to communicate directly, facilitating the exploitation of trust relationships that may exist between them. Web service streamlining complements the "traditional" web service invocation paradigm, thus the involved web services remain available for streamlining-unaware consumers that wish to invoke them. Future work will include experimental quantification of the benefits through simulation, the full implementation and integration of the mechanism into operational systems and optimization of streamlined execution, both at service-composition time and at runtime.

REFERENCES

[1] Papazoglou M. P., Georgakopoulos, D, "Service-oriented computing", Communications of the ACM, Vol. 46, No. 10, pp. 25-28, 2003

[2] Newcomer E. "Understanding Web Services: XML, WSDL, SOAP, and UDDI", Addison Wesley Professional, 2002. ISBN: 0201750813

[3] Bercic B., Vintar M., "Ontologies, Web Services, and Intelligent Agents: Ideas for Further Development of Life-Events Portals", Proceedings of EGOV 2003, LNCS 2739, pp. 329-334, 2003

[4] Papazoglou, M.P., "Service -Oriented Computing:

Concepts, Characteristics and Directions", Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE'03), pp3-12, 2003

[5] Bunting D. et al., "Web Services Composite Application Framework (WS-CAF)", Ver1.0, 2003, http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf

[6] Casati, F., Ilnicki, S., Jin L.J., Krishnamoorthy V., Shan M.C. "Adaptive and Dynamic Service Composition in eFlow". Proceedings of Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000, pp. 13-31, Stockholm, Sweden, 2000

[7] Lepouras, G., Vassilakis, C., Sotiropoulou, A., Theotokis, D., Katifori, A., "An Active Ontology-based Blackboard Architecture for Web Service Interoperability", proceedings of the Second IEEE Conference on Service Systems and Service Management, pp. 573-578, 2005

[8] Peltz C., "Web Services Orchestration", Hewlett Packard, Co., January 2003, http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf

[9] Vacca, J.R., "Public Key Infrastructure: Building Trusted Applications and Web Services", AUERBACH, 2004, ISBN: 0849308224

[10] Tambouris E., "An Integrated platform for Realising Online One-Stop Government: The eGov Projet", Proceedings of the DEXA International Workshop "On the Way to Electronic Government", IEEE Computer Society Press, Los Alamitos, CA, p. 359-363, 2001

[11] Maamar Z., Kouadri S. M., Yahyaoui H., "A Web Services Composition Approach based on Software Agents and Context", Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 1619-1623, 2004

[12] Iona Technologies, "The Artix Enterprise Service Bus", 2005, http://www.iona.com/products/artix

[13] BEA, IBM, and Microsoft, "Business Process Execution Language for Web Services (BPEL4WS)", 2003, http://xml.coverpages.org/bpel4ws.html

[14] IBM Web Services Flow Language (WSFL), 2003, http://xml.coverpages.org/wsfl.html

[15] OntoGov project, "User Requirements and Specifications", Project Deliverable D4, available from http://www.ontogov.com

[16] Tanenbaum A.S., "Modern Operating Systems", Prentice Hall, Englewood Cliffs, N.J., 1992

[17] Heberlein, L., Bishop, M., "Attack Class: Address Spoofing", Proceedings of the 19th National Information Systems Security Conference, pp. 371-377, 1996

[18] Gutterman, Z., Malkhi, D., "Hold Your Sessions: an Attack on Java Servlet Session-id Generation", Proceedings of Cryptographers' Track, RSA Conference, pp44-57, 2005