Tuple Spaces for Self-Coordination of Web Services

Zakaria Maamar
Zayed University
United Arab Emirates

Djamil Benslimane
Chirine Ghedira
Claude Bernard Lyon 1 University
France

Qusay H. Mahmoud
Hamdi Yahyaoui
University of Guelph, Canada
Laval University, Canada

ABSTRACT
This paper presents an approach that supports the self-coordination of Web services. Coordination is a designer-driven operation where, for instance, the designer clearly indicates the actions that the Web services have to perform in case of conflicts. In this paper, a different approach is adopted. Web services are enhanced with mechanisms, which allow them to coordinate themselves during run-time. These mechanisms are encoded using control tuples that Web services post on tuple spaces. Web services consult a tuple space and consume the control tuples that are relevant to their coordination work.

Keywords
Web service, self-coordination, and control tuple.

1. INTRODUCTION
This paper presents an approach that aims at supporting the self-coordination of Web services. To achieve this support, the approach integrates control tuples [2] into its operation. By Web service (also called service in the rest of this paper), it is meant an accessible application that other applications and humans can discover and trigger to satisfy multiple needs [3]. Furthermore by coordination, it is meant the definition of the execution order of a set of Web services that collaborate towards achieving the same objective [9]. Besides the execution order, the definition of coordination includes the data and temporal dependencies that exist between the Web services, the corrective actions that the Web services execute in case of exceptions, and the resolution strategies that the Web services adopt in case of conflicts. Designers are responsible for specifying the coordination of Web services during design-time. In this paper, it is aimed at allowing Web services to coordinate themselves during run-time. Once a designer completes the definition of coordination, self-coordinated Web services are not authorized to seek any extra input from the designer. These Web services are expected to be autonomous in their action selection and execution. To this purpose, the self-coordination is built upon control tuples. Control tuples enable Web services to engage in asynchronous interactions with their peers so that they can agree on how their execution order can be adjusted, data and temporal dependencies can be reviewed, etc.

A major feature supporting the adoption of Web services is their capacity to be combined into composite services. The participation of Web services in various and simultaneous compositions has promoted the Web services instantiation principle of [8]. This instantiation principle consists of creating Web services of type instance from Web services of type root. The creation of a Web service instance occurs each time a Web service root receives and accepts an invitation of participation in a composition. Invitations originate from composite services, which seek the participation of Web service roots. One of the advantages of the instantiation principle is the possibility of organizing a Web service root along three categories (Fig. 1): Web service instances already deployed, Web service instances currently deployed, and Web service instances to be potentially deployed upon invitation acceptance. Invitation acceptance, which results in service instantiation, is subject to the verification of many factors denoted by constraints in Fig. 1 and detailed in [8]. A Web service root can always reject an invitation of participation in a composition due to multiple reasons, e.g., unavailability.

![Figure 1: Organization of a Web service root](image)

Web services in general and their instances in particular need computing resources on which they can be run. A Web service root requires resources for the following operations: (i) self-assessment of current status before invitation acceptance/rejection, (ii) service instantiation and attachment to composite service, and (iii) monitoring of service instances following their deployment. A Web service instance requires resources for the following operations: (i) processing of input arguments so that users’ requests are satisfied, (ii) interaction with respective Web service roots for update needs, and (iii) data exchange with pre- and post-service instances. The resources have first, to be identified (this is outside this paper’s scope) and then, scheduled because of their limited computing capabilities. Several Web service roots would like...
simultaneously allocating their respective service instances to the same resources, which might not be doable.

Before submitting Web service instances for execution, they engage themselves in a self-coordination process for the following reasons: (i) optimize the use of resources by taking advantage of available results of other peers, and (ii) recommend to service roots reviewing certain of their resource selection strategies based on past-execution experiences. With regard to (i), it was reported that the rationale use of resources becomes critical in Web services-based transaction management. It is not acceptable to lock resources for long periods of time as the number of Web services that are announced to users continues to grow, so that the use of resources will become intensive. Excessive periods of resource locking result in delay and scalability problems. Therefore reusing available results could to a certain extent constitute an alternative approach to resource blocking. With regard to (ii), the dynamic nature of the Internet requires a continuous monitoring and adaptability of the strategies that Web services take on, which allows a better handling of the occurring exceptions and computing features of resources.

Web services composition is a very active area of R&D [9]. However, very little has been done so far regarding the self-coordination of Web services. In particular, several obstacles still exist such as (i) lack of techniques for identifying, modelling, and specifying the requirements of self-coordinated Web services, (ii) existing approaches for Web services composition (e.g., WSFL) typically facilitate choreography only among component services, while ignoring the coordination that might occur among the service instances of the same component service, and (iii) available coordination approaches (e.g., WS-Coordination) do not accommodate neither the features of the Web service instantiation principle nor the temporal organization of a Web service as depicted in Fig. 1. In this paper we aim at presenting an approach for the self-coordination of Web services using control tuples. Moreover it will be shown in the paper that the self-coordination has a pre-requisite that consists of assessing the similarity between Web service instances. This similarity is tackled using Case-Based Reasoning (CBR) technique [1].

Section 2 motivates the value-added of control tuples to the self-coordination of Web services. Section 3 presents the architecture of this self-coordination. Prior to any coordination, the similarity of Web services is assessed in Section 4. Implementation status of self-coordinated Web services is discussed in Section 5. Section 6 summarizes some of the related works. Section 7 concludes the paper.

2. RATIONALE OF CONTROL TUPLES

The Linda-based tuple-space model is well recognized as a model for managing asynchronous interactions between separate components [2]. Three reasons motivate our integration of control tuples into the self-coordination of Web services. First, because Web service instances are created at different periods of time, the asynchronous mode of interaction in a tuple space is suitable for supporting the exchange of information between service instances that are already deployed or under deployment. Service instances do not have to exist during the same period of time to be able to engage in an information exchange. Second, the tuple space has storing and searching facilities for control tuples so that service instances can post and collect control tuples when needed. Finally, a tuple space supports notifications to service instances, which indicate using templates the kind of data they are interested in. This is important as data relevant to service instances are retrieved according to their content and not to a specific location or specific data-sender.

3. SELF-COORDINATION APPROACH

Discussions in Section 1 have shown the existence of three levels of abstraction: instance, root, and composite. Based on these three levels we advocate that the self-coordination process occurs among the Web service instances that originate from the same Web service root. This self-coordination is driven by two elements (i) the resources that are affected to service instances for their execution and (ii) the commonalities that might exist between the service instances. These commonalities are associated with users’ preferences and identify execution time parameter: a specific period of time to execute a service instance (e.g., at 2pm).

Fig. 2 shows the participants in the self-coordination of Web services. Each time a Web service root is instantiated upon invitation acceptance, the newly-obtained service instance is added to a pool of service instances, which originate all from this Web service root. A service instance (e.g., service instance21) that resides in a pool is already assigned to a composite service as a component. A pool of service instances has a dynamic content because of the service instances that a Web service root adds after creation and possibly withdraws after successful execution.

Once a service instance is created and then put in a pool, the service instance posts its execution-time parameter on the tuple space of the pool using write primitive. The posting of the execution parameters is associated with a control tuple, which is announced to the existing Web service instances. These ones might have an interest in the execution time of the new service instance. The existing Web service instances are kept in the pool waiting for their deployment turn on resources. After posting its execution parameter the service instance consults the available postings (i.e., existing control tuples) of other service instances using read primitive. The aim of the consultation is to discover any communality between the service instance and other service instances at the level of execution time. If there is communality, the involved service instances engage in interactions. The goal is to agree on how to join their efforts, although these service instances belong each to a separate composite service. For illustration if two service instances are going to be triggered at the same time and on the same resource, only one service instance has to be considered. The performance outcome is shared among these service instances when this outcome is posted on the tuple space. Since a service instance will not be executed, the Web service root notifies the appropriate resource so that this resource can be released; this means the possibility of accommodating extra execution requests. The notions here of same service instances and same resource are extremely important; two similar service instances can be assigned to different resources based on multiple resource selection strategies [10].

Another use of the tuple space of the architecture for self-coordinated Web services is to store the various postings of the service instances that completed their execution. The storing lasts for a certain period of time (Fig. 1). These postings are on different matters including the service instances’ past-execution experiences with certain resources in terms of reliability, QoS, etc. Information on execution
4. SIMILARITY AND WEB SERVICES

4.1 Similarity definition

One of the challenges that the self-coordination of Web services has to address is how to define the similarity of Web services at the instance instead of root level. Claiming the similarity of service instances just because they originate from the same service root is by far not sufficient. To this end, we suggest two requirements that service instances have to satisfy. The first one is related to the specification of composite services. The second one is related to the values of the input arguments of the service instances.

- **Requirement 1**: Web service instances should have in common the same specification of the composite services to which they respectively belong. For example if two service instances, which come from the same Web service root, participate in two separate composite services having each a different specification, one about travel planning and one about paper review, then these service instances will be declared not similar. To affirm the compliance of service instances with **requirement 1**, their composite services should have a common specification, e.g., travel planning.

- **Requirement 2**: Web service instances should have in common the values that are assigned to their respective input arguments, so that they could share the results of their respective performances among them. This requirement is complex to satisfy since the values of input arguments are dependent on the needs of users and types of application domain. For example, **city of destination** argument of **flight itinerary** Web service could receive *Paris* as input value for user_1 and *London* as input value for user_2. To address the compliance of service instances with **requirement 2** we identify the following cases:

1. **Full match values**: all values assigned to the input arguments of the service instances are identical. Referring back to flight-itinerary example this means that user_1 and user_2 want both to travel to the same city (e.g., *Paris*).

2. **Partial match values**: only some values assigned to the input arguments of the service instances are identical. Referring back to flight-itinerary example this means that user_1 and user_2 want both to travel to the same city (e.g., *Paris*) but on different dates (e.g., 12/12/04 for user_1 and 13/12/04 for user_2).

3. **No match values**: all values assigned to the input arguments of the service instances are different. Referring back to flight-itinerary example this means that user_1 and user_2 want both to travel to different cities on different dates.

Web service instances, which satisfy **requirement 1** and **requirement 2/full match values**, are declared similar. Therefore they can share their execution outcome, which permits optimizing the use of resources (i.e., only one service instance will be executed). For the Web service instances, which satisfy **requirement 1** and **requirement 2/partial match values**, they cannot at this stage be declared similar. Additional processing is needed using case-based reasoning.

Figure 2: Architecture supporting the self-coordination of Web services

A resource that did not meet its response time QoS over multiple execution sessions of Web service instances can be discarded by a Web service root from the list of resources to interact with in the future. An additional use of the stored postings is to have notes about the obstacles that the execution of a composition specification has faced. Web service roots originate from multiple providers. Hence the semantic heterogeneity of the data that their service instances exchange can easily arise. These obstacles have to be addressed for the benefit of the forthcoming compositions.

An interaction of type monitoring exists in Fig. 2. This interaction connects Web service roots to their respective tuple spaces. The monitoring that a service root executes is important because of the impacts that the self-coordination of service instances might have on the service root. This impact is illustrated with accepting/rejecting further invitations of participation in composite services. A Web service root has to be aware of the progress of its service instances, the agreements between them as discussed previously, their expected completion time, and the exceptions that each service instance might have raised. Basically the operations of a Web service root are concerned with various critical management elements such as (i) ability to monitor the operations of its service instances, (ii) ability to guide its service instances towards the actions that smooth and boost the self-coordination, and (iii) scalability of the pool of service instances in case of unforeseen circumstances.
The main idea in CBR is that if two problems look similar then the solutions to both problems will possibly be similar [5]. Finally for the Web service instances, which satisfy requirement 1 and requirement 2/no match values, they are not declared similar. This requires an independent execution thread per Web service instance.

4.2 Similarity evaluation

Case-based reasoning is not solely built on the knowledge of a problem domain or associations along generalized relationships between problem descriptors and conclusions. Instead CBR uses the specific knowledge of previously experienced, concrete problem situations (cases). A new problem is solved by finding a similar past case, and reusing this past case while taking into account the specificities of the new problem situation.

In CBR terminology, a case usually denotes a problem situation. A previously experienced situation, which has been captured and learned in a way that it can be used for solving future problems, is referred to as past case. In this paper, past cases illustrate two situations featuring Web service instances. The first situation is about the Web service instances that have completed their deployment (Fig. 1, line labelled with past), so their performance result is immediately made available for reuse. The second situation is about the service instances that are currently waiting for their deployment turn (Fig. 1, line labelled with present), so their performance result can be reused later on.

CBR uses a similarity measure function to evaluate the match between a given situation and cases stored in the case base. To carry out the evaluation, a mechanism for structuring the cases is initially required. We structure a Web service instance (in fact a Web service root) along the following arguments: (i) identifier, (ii) resource label on which the service instance is performed, (iii) execution time, and (iv) a set of extra arguments that depend on the application domain in which the Web service instance is used (e.g., city of destination, date of departure, date of return arguments in flight itinerary application-domain).

Furthermore the structure of a Web service instance decomposes an argument into two types: core and optional. The difference resides in the chronology of comparing the arguments during similarity evaluation. The chronology starts with the core arguments and continues next with the optional arguments. Users indicate which arguments are either core or optional. This excludes the first three arguments that are core by default. The categorization of arguments helps reduce the number of irrelevant cases of service instances to consider during similarity evaluation. For example the service instances that are not assigned to the same resource like the new service instance are dropped during the similarity evaluation. Web service roots are in charge of assigning values to identifier and resource label arguments of a Web service instance. Users are in charge of assigning values to the rest of arguments of a Web service instance. Last but not least each argument has a user-given weight \( w \), which varies depending on its type whether core or optional. The weighting prioritizes the arguments during the similarity assessment.

To illustrate the previous paragraphs about similarity evaluation, we consider the situation of a new Web-service instance and a set of Web service instances that are waiting for their deployment turn. As a first step the similarity function compares the values of resource label and execution time arguments of the new service instance to the values of the same arguments of the rest of service instances. If a similarity is identified according to requirement 2/full match values between these arguments, the similarity operation will proceed with comparing the rest of the arguments starting with the core and then the optional arguments. Otherwise, the similarity evaluation is terminated. The new service instance will have its own execution thread.

In what follows we overview the similarity function that evaluates the similarity between a given service instance \( I \) and a case \( C \) in the case base\(^{1}\). The case base is illustrated with the pool of services connected to a tuple space (Fig. 2). The similarity is based on the following set of rules:

1) The similarity measure function \( \text{Sim}(I,C) \) between a service instance \( I \) and a case \( C \) is computed as

\[
\text{Sim}(I,C) = \frac{\sum_{i=1}^{n} \text{Sim}(I[arg_i,C[arg_i]) \times w_i}{\sum_{i=1}^{n} w_i}
\]

Equation 1, which is an implementation of the nearest neighbor algorithm, shows that the similarity between \( I \) and \( C \) is the sum of every similarity measure of an input argument of \( I \) and an input argument of \( C \), divided by the sum of the weights of all input arguments. In this equation, \( n \) is the number of input arguments of the structure of a Web service instance; \( I[arg_i] \) is the value of the input argument \( i \) of the service instance \( I \); \( C[arg_i] \) is the value of the input argument \( i \) of the case \( C \); And, \( w_i \) is the importance weighting that is assigned to an input argument according to its type (core or optional).

2) If the value of the input argument of the service instance \( I \) is exactly the same as the value of the input argument of the case \( C \), then the similarity between both input arguments will receive the highest value, which is 1 (\( \text{Sim}(I[arg_i,C[arg_i]) = 1 \)). All input arguments of type core have to comply with this rule. Otherwise (i.e., there is no exact match between these arguments), the similarity evaluation is stopped.

3) If an input argument of the service instance \( I \) is close to the input argument of the case \( C \), and both input arguments can be computed in ordinal numbers (e.g., ticket fare in Euros), then the comparison between both input arguments will be carried out based on the meaning and purpose of the input arguments. For instance the similarity between the prices of a product was calculated in [6] using statistic functions (e.g., standard deviation) and averages. Referring back to flight-itinerary service, number of days is an argument that falls within the scope of rule 3.

4) If the value of an input argument of the service instance \( I \) does not match the value of the input argument of the case \( C \), then the similarity between both input arguments will receive the lowest value, which is 0.

The outcome of the similarity evaluation is a set of a numbers, where each number represents how a new Web service instance is close to the set of Web service instances of a pool. The highest is the number, the closest is the similarity between service instances.

5. IMPLEMENTATION STATUS

A proof-of-concept implementation of the architecture for self-coordinated Web services of Fig. 2 is under construction.

\(^{1}\)The similarity function is subject to further research.
using primarily Sun Microsystems’ J2EE 1.4 to create Web services and their respective Web service instances.

In addition to J2EE, WSDL and UDDI are used for service specification and discovery. Since WSDL only focuses on how to invoke a service, some of the attributes, which deal with Web services personalization based on users’ preferences of type execution time, are not supported by WSDL. To overcome this limitation such attributes are specified as tModels. Another feature of the prototype is the use of Sun Microsystems’ JavaSpaces to implement the tuple spaces. JavaSpaces is an implementation of Linda and provides a simple, fast, and unified mechanism for sharing, coordinating, and communicating distributed resources, services, and objects across the network. In the prototype, if a Web service instance is interested in a specific event, then it will be notified about this event occurrence through JavaSpaces.

6. RELATED WORK

The multiple research projects on Web services have neither promoted the Web services instantiation principle nor aimed at coordinating Web services at the instance instead of root level. We present some of the projects that have inspired shape the approach for self-coordinated Web services.

Lucchi and Zavattaro have noted that there is a renewed interest in Linda-like coordination languages (tuple space is a Linda-based model) from both theoretical and implementation perspectives [7]. In particular three main aspects have attracted the attention of the coordination community: (i) security in coordination, (ii) integration of Linda into Web technologies such as XML, and (iii) coordination of Web services. The work presented in this paper is inline with the third aspect. The use of tuple spaces is well received by the Web services community as the work of Fontoura et al. illustrates [4]. The authors have developed a set of tuple space-based tools to help develop and manage Web services. The objective of these tools is to simplify the creation, deployment, configuration, and invocation of services. Compared to Fontoura et al., our use of tuple spaces is one step forward. Indeed tuple spaces are not only used for managing Web services, but also for coordinating their respective instances with regard to their execution parameters and resource allocation.

The application of CBR to Web services composition is discussed in [5] for service discovery. The purpose is to allocate the pre-assembled closest services depending on users’ requirements. The service composition model of [5] integrates both types of service composition namely proactive and reactive. The deployment of case-based reasoning for service discovery has required developing a service case-base. It stores the multiple collections of service cases where a service case is a pre-assemble composite service that is constructed upon a set of relationships and constraints among the component services. In this paper the service cases correspond to Web service instances, whereas the service case-base corresponds to the tuple space. However the similarity evaluation of [5] is different from the similarity evaluation of this paper. Indeed the similarity here concerns Web service instances issued from the same Web service root, whereas the similarity of [5] concerns different Web service roots.

7. CONCLUSION

We presented our approach for self-coordinated Web services. We mainly focused on the coordination involving Web service instances, which in fact are the components of a composite service. The coordination was built upon two well known techniques including tuple spaces and case-based reasoning. Prior to any coordination, the Web service instances assess their similarity for a better reuse of the results obtained from previous executions.

The self-coordination of Web service instances constitutes the trigger of a major coordination process, which spreads over the remaining levels of the architecture of Fig. 2 namely root and composite. Three degrees of coordination are considered, which enables the presence of various types of configuration ranging from distributed coordination (because coordination occurs in several pools of service instances) to centralized coordination (because coordination is localized at the level of a Web service root).

8. REFERENCES