Web Services for E-commerce: guaranteeing security access and quality of service

Mario Bravetti Roberto Lucchi Gianluigi Zavattaro Roberto Gorrieri

Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna - Italy

{bravetti,lucchi,zavattar,gorrieri}@cs.unibo.it

ABSTRACT

Being E-commerce one of the most critical Internet application, it is fundamental to employ technologies which guarantee not only secure transactions but also an adequate quality of service. In this paper we present a solution to this problem based on an extension of the emerging Web Service technology. In particular we introduce a new Web Service discovery protocol that extends standard UDDI capability by adding: (*i*) the discovery of Web Services at run-time supporting environment re-configurations, (*ii*) security access control to Web Services and (*iii*) a mechanism for distributing service invocations among several Web Services implementing (at different efficiency levels) the same task. Discovery at run-time is realized by dynamically resorting to the discovery protocol (a Web Service itself) every time a service is invoked, access control is obtained by employing symmetric and asymmetric keys, while the different efficiency levels of service implementations are represented via weights. The proposal is formalized by employing a coordination platform, which is a probabilistic extension of the existing `WSSecSpaces`. Such a platform is based on a data space, where data can be not only protected via access control mechanisms, like in `WSSecSpaces`, but also accessed probabilistically.

Categories and Subject Descriptors

J.8 [**Internet Applications**]: Electronic commerce; C.2 [**Communication/Networking Technology**]: Miscellaneous; D.3 [**Programming Languages**]: Formal Definitions and Theory

Keywords

Coordination Languages, Linda, Probability, Security, Web Services

SAC '04, March 14-17, 2004, Nicosia, Cyprus.

1. INTRODUCTION

Technologies for implementing E-commerce must be carefully developed so to provide critical guarantees to the consumers and to the suppliers of the services. In particular it is fundamental that such technologies ensure not only the security of transactions, but also an adequate quality of service, i.e. the availability of the service should be guaranteed to all users. This paper presents a solution to this problem in the context of the emerging technology of Web Services. Web Services provide a mean to express complex business-to-business processes, in terms of finer-grained subtasks. Clearly, also E-commerce applications can benefit of Web Service technology; indeed, since the subtasks that compose E-commerce applications are often dedicated to different categories of users, e.g. service suppliers and consumers, defining access control policies and supporting quality of service at the level of single Web Services turns out to be a very convenient approach.

The Web Service technology provides standard mechanisms for describing the interface and the services available on the Web, as well as protocols for locating such services and invoking them. In particular each Web Service has an associated WSDL (Web Services Description Language) document which describes how it works and how to invoke it (via its "physical" address). Such document is registered at an UDDI (Universal Description, Discovery and Integration) server that provides a discovery service for the WSDL descriptions.

Moreover it is possible to specify complex services out of simpler ones via the so called *Web Services choreography.* Several proposals have been already set up: BPML [1] by BPMI.org, XLANG [16] and BizTalk [8] (a visual specification environment for XLANG) by Microsoft, WSFL [11] by IBM, BPEL4WS [9] by a consortium grouping BEA, IBM, Microsoft and others, etc. These are languages permitting the specification of the interdependencies among collaborating Web Services via the indication of the flow of their relative invocations.

Unfortunately, the current implementations of the Web Service technology are based on a very primitive discovery protocol (the UDDI) that assumes that: (*i*) Web Services are located once and for all (at compile/design time) before being invoked from other Web Services, and (*ii*) there is no abstract/semantical concept of *task* performed by a certain Web Service: several Web Services may provide the same task by means of different implementations (e.g. by using different languages/algorithms) or replicas of the same implementation.

As a matter of fact this prevents an effective use of the current Web Service technology for complex applications like E-commerce: since a run-time discovery of Web Services is not allowed, it is not possible to take into account reconfigurations of the environment (i.e. Web Services that appear and disappear after compile/design) and to manage distribution of the workload among several Web Services implementing the same task. In other words UDDI supports discovery operations such as "provide me with a link to a specific kind of resource" while it is not expressive enough for supporting queries such as "provide me with a link to a generic kind of resource performing a specific task in such a way that the overall resource usage is balanced".

In this paper we present a solution that provides discovery of Web Services at run-time and introduces the notion of abstract task in Web Service registries. On this basis, we also provide the new discovery protocol with security access control mechanisms and a mechanism for distributing service invocations among several Web Services implementing (at different efficiency levels) the same task.

The basic idea for providing run-time discovery of Web Services is to use a discovery protocol which is itself a (statically located) Web Service. In particular we resort to a particular kind of technology which, together with Web Service choreography, is currently one of the hottest topic of research in the Web Service community: Web Services exploited at run-time by other Web Services as *coordination spaces*. These services can be used in order to store and retrieve the information needed for managing the collaboration among Web Services that are willing to cooperate (in our case registry information: WSDL descriptions of Web Services associated with tasks). Some proposals have been already defined, see e.g. Ruple [15], `WSSecSpaces` [13], and the proposal by Álvarez et al [3].

All these proposals are based on the notion of loosely coupled interaction obtained via *generative communication* [10]: the coordination among the Web Services does not occur via a direct communication, but it is mediated by the coordination space that is a shared repository of data. A sender communicates with one or more receivers *writing* data in the repository (e.g. the registration of a new Web Service in our case); a receiver can *read* or *consume* these data from the space indicating with a pattern the kind of datum he is interested in (e.g. the discovery or the deregistration of a Web Service implementing a given task in our case). If several data are available that match the pattern, only one of them is returned and it is chosen non-deterministically. This form of communication is referred to as generative communication because when a datum is produced, it has an existence which is independent of its producer, and it is equally accessible to all components.

Moreover, some coordination spaces [15, 13] also support security access control to the data inside the shared repository. In particular `WSSecSpaces` [13], which has a finer grained access permission mechanism w.r.t. [15], makes it possible to discriminate among write, read and consume access permission at the level of the single datum, hence to authenticate/identify the producer of an datum or its reader/consumer. This is obtained by decorating data in the coordination space with symmetric and asymmetric access keys associated with single access operations. In the following we use the term *agent* to denote an entity having access to the coordination space characterized by a knowledge (e.g., partitions and asymmetric partitions); in our context, it could be a Web Service. In the case of the registry service these mechanisms allow us to model the fact that only allowed agents, can register or deregister their entries from the registry. On the contrary the retrieval (read) operation can be set to have a different (more free) access modality. As far as the read operation is concerned, we can also model security access control to registered Web Services: they may be reserved to different classes of users (e.g. staff of a factory providing services as opposed to the customers).

In this paper we introduce a new feature in coordination spaces: *probabilistic* retrieval of data in the repository. We use this new feature to model a discovery protocol which guarantees a rather concrete property related to the quality of service: a balancing in the workload among different Web Services executing the same task (possibly at different levels of efficiency). Note that, in this respect, it is crucial for the discovery protocol to be based on a probabilistic mechanism. Web Services register themselves at the registry service by introducing data in the shared repository, while clients discover the availability of Web Services implementing tasks by performing data retrieval operations. According to this pattern of interaction, in the case several implementations for a given task are available, one must be chosen. If the choice is just non-deterministic (as in the coordination models of [15, 13, 3]), there is no guarantee that the invocations are uniformly distributed among the available servers.

Technically, the new technology that we propose in the pure area of cooperation via coordination spaces is based on an extension of `WSSecSpaces`. With respect to the usual non-deterministic data retrieval operation, we propose a more sophisticated one that *probabilistically* selects the datum to be returned according to a *weight* which is associated to each datum in the repository. The higher the weight of a datum is, the higher is the probability for this datum to be returned as a result of the data retrieval operation. Given this new probabilistic access mechanism, it is possible to obtain a registry service that behaves in a convenient way. The efficiency level of a Web Service is somehow quantified and an appropriate weight is associated to the corresponding entry in the registry service. When the clients perform the discovery of tasks executing data retrieval operations, the underlying probabilistic access mechanism guarantees a balanced distribution of the returned data (Web Services implementing the task).

The outline of the paper is as follows. In Section 2 we formalize the new probabilistic coordination model by introducing the probabilistic extension of `WSSecSpaces` described above. In Section 3 we prove that the combination of the access control mechanisms already provided by `WSSecSpaces`, and the new probabilistic access control mechanism, permits the definition of a dynamic Web Services discovery protocol that supports controlled as well as balanced Web Service interaction. Finally, Section 4 describes an example of application that uses the proposed Web Service registry (a software factory distributing its products to clients with different privileges), while in Section 5 we draw some conclusive remarks.

2. ADDING PROBABILISTIC DATA-RETRIEVAL TO WSSECSPACES

WSSecSpaces is a Web Service that implements coordination model of SecSpaces [4] and, similarly to other XML-based tuple spaces [17, 6], exploits XML technologies to improve the expressiveness of the matching rule. More precisely, XML-Schema [7] documents are used in templates fields to define typed wildcards; these define the type of an entry field that can match with that wildcard. In Section 2.1 we present the primitives of WSSecSpaces, while in Section 2.2 we propose an extension of the model that supports probabilistic accesses to the entries stored into the data space.

2.1 WSSecSpaces

WSSecSpaces supports secure data-driven coordination in open environments. It extends Linda [10] by permitting to control the access to the entries stored in the coordination space and to authenticate/identify the producer of an entry or its reader/consumer.

The coordination primitives of WSSecSpaces are the classical ones of Linda: $out(e)$, $in(t)$ and $rd(t)$. The output operation $out(e)$ inserts an entry e into the data space (in short, DS). Primitive $in(t)$ is the blocking input operation: when an occurrence of an entry e matching with the template t is found in the space, it is removed and its content is returned. The read primitive $rd(t)$ is similar to $in(t)$, but in this case the entry e is not removed from the space.

In order to express security, WSSecSpaces implements access control mechanisms by extending Linda tuples with special control fields, namely *partition* and *asymmetric partition* fields. The former ones are used to logically partition the space (where a partition can be accessed only by owning the associated partition identifier), while the latter ones, as it will be clear in the following, allow us to discriminate between write and read/remove access permissions on each entry.

Let $Mess$, ranged over by m, n, ..., be an infinite set of messages and $TMess$, ranged over by tm, tn, ..., be an infinite set of messages used by templates. We assume that $Mess \subseteq TMess$, whose meaning is that data fields of templates can be set to any message used by entries (to find exact matching) or to typed wildcards. Let $Partition$, ranged over by c, c_t, ..., be the set of partition identifiers. Partition identifiers logically partition the space and the access to a partition is restricted to only those agents that know the partition identifier. Indeed, in order to perform an operation on a partition, agents must know the partition field identifying that specific partition. In order to allow all the agents to interact with each other via WSSecSpaces primitives, we also assume that $Partition$ contains a special default value, say #, known by all the agents. Let $APartition$, ranged over by k, k', k_t, ..., be the set of asymmetric partition identifiers. Informally, an entry with asymmetric partition field set to k can be accessed only by providing a template having a co-key $\overline{k}$ as asymmetric partition field. We also assume that $APartition$ contains a special default value ?, known by any agent, used to allow any agent to interact with each other. Let "$\overline{\cdot} : APartition \to APartition$" be a function such that $\overline{?} = ?$ and if $\overline{k} = k'$ then $\overline{k'} = k$. We also assume that, given $k, k' \in APartition$, it is possible to test if $\overline{k} = k'$ and that, given k, there is no way to guess $\overline{k}$ (we also use the term co-key of k to refer to $\overline{k}$).[1]

Access control mechanisms are based on control fields. In order to discriminate between rd and in access permission, entries have two occurrences of control fields: one associated to in operations and another to rd operations. A template has only one occurrence of control fields that is dynamically associated to the operation the agent is performing using it (i.e., a rd or a in operation).

The set $Entry$ of entries, ranged over by e, e', ..., is defined as follows:

$$e = < \vec{d} >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$$

where $c, c' \in Partition$, $k, k' \in APartition$ and the tuple of data $\vec{d}$ is a term of the following grammar:

$\vec{d} ::= d \mid d; \vec{d}$
$d ::= m \mid c \mid k.$

A *data field* d can be a message, a partition identifier or an asymmetric partition identifier. We define $\tilde{\cdot}$ as the function that, given an entry e, returns its tuple of data, i.e., if $e = < \vec{d} >^{[c]_{rd}[c']_{in}}_{[r]_{rd}[r']_{in}}$ then $\tilde{e} = \vec{d}$.

The set $Template$ of templates, ranged over by t, t', ..., is defined as follows:

$$t = < \vec{dt} >^{[c_t]}_{[k_t]}$$

where $c_t \in Partition$, $k_t \in APartition$ and $\vec{dt}$ is a term of the following grammar:

$\vec{dt} ::= dt \mid dt; \vec{dt}$
$dt ::= tm \mid c \mid k \mid null.$

With respect to entries, data fields used by templates can also be set to an additional value (*null*) that denotes the wildcard: the wildcard is used to match with all field values.

In order to describe the refined version of the matching rule (exploiting XML technologies) used by WSSecSpaces, we consider data fields (of entries and templates) as XML documents. Hence, we informally denote by $\triangleleft_{XML}$ the rule used by WSSecSpaces as matching relation between entries and templates fields (that exploits XML-Schema to define typed wildcards).

Definition 2.1. ***Matching rule and return value*** – *Let* $e = < d_1; d_2; \ldots; d_n >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$ *be an entry,* $t = < dt_1; dt_2; \ldots; dt_m >^{[c_t]}_{[k_t]}$ *be a template and* $op \in \{rd, in\}$ *be an operation. Let* c_e *and* k_e *be the control fields of* e *associated to* op. *We say that* e *matches*$_{op}$ t *if the following conditions hold:*

1. $m = n$
2. $dt_i \triangleleft_{XML} d_i$ *or* $dt_i = null$, $1 \leq i \leq n$
3. $c_e = c_t$
4. $\overline{k_e} = k_t$.

If a rd *or an* in *operation with template* t *is performed on a matching entry* e, *only the data fields are returned, i.e., the returned value is* $\tilde{e}$.

Conditions 1. and 2. rephrase the classical Linda matching rule, that is, they check whether e and t have the same arity and whether each data field of e either matches with the corresponding field of t or if the latter one is set to a wildcard. The third condition controls that the partition field

[1] The asymmetric partition fields can be implemented by exploiting asymmetric (or public key) cryptography (see [4]) and, usually, this property holds in public key crypto systems where, given a public key, it is not possible to guess the associated private key, and vice versa (for more details, see [14]).

of the entry, associated to the operation *op*, is equal to that of the template. Finally, the fourth condition checks that the asymmetric partition field of the template corresponds to the co-key of the asymmetric partition field of the entry associated to the operation *op*.

As pointed out by the matching rule, partition fields are a special kind of data fields that do not accept wildcard in the matching evaluation. Differently from partition identifiers, the asymmetric partition fields make it possible to discriminate between the permission of write and of read/remove of an entry. For instance, in order to read an entry with asymmetric partition field set to k the agent must set to $\overline{k}$ the asymmetric partition field of the template. This value can be an unknown value to the producer of the entry (because in order to perform a write operation it needs only k). Therefore, following the same idea of partitions, by properly distributing these values we can assign to agents the permission to perform a subset of possible operations on that entry.

Finally, the returned value of read/input operations does not include the control fields; it contains only the data stored inside the tuple of data fields. Therefore, new access permissions can be acquired only by performing read/input operations of entries containing partition or asymmetric partition values inside the tuple of data.

2.2 Adding probability to WSSecSpaces

In this section we discuss an extension of the model obtained by adding probabilistic accesses to the entries stored in the shared space. More technically, we insert an additional attribute to the entries, passed as a parameter to the output operations, that represents the weight of the written entry. Informally, the weight associated to each entry in the DS represents its *appealing degree*: among the entries in the DS that can be read/removed by an agent, the entry with greatest weight has the highest probability to be read/removed by the agent. It is worth noting that the Linda model accepts multiple instances of the same entry, therefore the probability to access a specific entry depends also on the weights associated with the several instances of each matching entry.

As we will show in Section 3, these weight attributes will be exploited to model the distribution of the service invocations among different Web Services taking into account their performance factor.

Entries stored in the DS are now represented by pairs of the form (e, w), that represent an entry e having weight w. Let $Weight$, ranged over by w, w', ..., be the set of weights, i.e. positive (non-zero) real numbers. The extended version of the `WSSecSpaces` primitives has the following syntax and meaning:

- *out* (e, w), where $e \in Entry$ and $w \in Weight$ is the output operation; given as parameters an entry e and a weight w it writes into the DS a pair (e, w).

- $in(t)$, where $t \in Template$ is the (unchanged) syntax of input operations; if an entry e matching the template t is available in the DS, i.e. $(e, w) \in DS$ for some $w \in Weight$, the execution of *in* causes the removal of a pair (e, w) from the space and returns $\tilde{e}$. The probability of removing a particular pair $(e, w) \in DS$ with e that matches t is the ratio of w to the sum of the weights w' in the pairs (e', w') in the DS such that e' matches with t (taking into account multiple occurrences of pairs).

- $rd(t)$, where $t \in Template$ is the (unchanged) syntax of read operation; if an entry e matching the template t is available in the DS, then the read is performed and the returned value is $\tilde{e}$. The probability of reading a particular pair $(e, w) \in DS$ with e that matches t is evaluated as in the input case. Note that, in particular, this means that the probability of reading a particular matching entry e contained in the DS is the ratio of the sum of weights w associated with the several instances of the entry e contained in the DS, to the sum of the weights of the entries e' in the DS matching with t.

It is worth noting that the probabilistic access to the entries is at the level of subspace the agent can access using a specific template. More precisely, the probability distribution depends on weights of all matching entries stored in the DS. Since all matching entries are contained in the partition of DS identified by the partition field of the template, also distributed implementations of the shared space can address the search of matching entries to specific locations by exploiting control fields.

Differently from control fields, we consider the weight as an attribute of the entry instead of as a possible additional data field. This is because weights are used to express the probability of the entries to be accessed and do not concern the content of the entries. Besides, we have not included in read and removal operations the possibility of indicating a specific weight of the matching entry, because the goal is to properly distribute the accesses to the matching entries according to the associated weights. For example, when the shared space is used to implement the Web Service registry mechanism the aim is to distribute the workload among the different implementations of the same task. Let us remark that this decision does not limit the expressive power because by using control fields it is possible to provide some users with better access to the discovery of Web Services. More precisely, the access to some Web Services can be reserved to a set of privileged users, augmenting (with respect to general users) their probability to find a Web Service with an high quality of the service.

3. A WEB SERVICE REGISTRY GUARANTEEING SECURITY AND QUALITY OF SERVICE

In this section we propose an implementation of a Web Service registry that provides clients with the following features: i) to discover Web Services; ii) to register new Web Services in the registry; iii) to remove a Web Service from the registry. We take advantage of the extended version of `WSSecSpaces` presented in Section 2.2. More precisely, the three phases described above can be mapped into the three operations provided by `WSSecSpaces`: i) *rd* operations are used to discover services; ii) *out* operations to register services, and iii) *in* operations to deregister a service or to update its weight.

In Section 3.1 we present the Web Services discovery protocol that exploits weights to properly balance the workload of Web Service invocations, while in Section 3.1.1 we describe how control fields can be exploited to restrict the access to Web Services. Finally, Section 3.2 describes how to register, deregister or update the information about Web Services. Furthermore, we show how, by discriminating between the *rd* and the *in* access permissions, we make it possible to guarantee that only allowed agents (e.g., the Web

Services themselves) can register, remove or update entries from the registry.

3.1 Web Service discovery

The UDDI [2] protocol supports registration and Web Service discovery. More precisely, each agent can interact with a UDDI server that, given the name (and optionally a set of attributes) of a service returns the WSDL document associated to the requested Web Service. The WSDL [5] document contains all information about a Web Service, such as the URL of the port that agents can use to access the service, the interaction modalities between requestor and Web Service (e.g., send-response) and the type of the argument passed to and received from the Web Service (usually based on XML-Schema [7]).

In the protocol that we are going to describe we assume that, similarly as in the standard UDDI protocol, each service is identified by a unique name. More precisely, we assume that the service indicates the kind of task performed; therefore, in general, many Web Services supply the same task (e.g., the sort of a list of elements). Formally, let $Task$, ranged over by s, s', ..., be the set of tasks Web Services can implement, $WSDL$, ranged over by ws, ws', ..., be the set of Web Services. A registered Web Service is represented by an entry in the DS; entries must have the tuples of data structured as follows (the range of values or the structure of control fields is left unchanged):

$$\vec{d} = < s, ws >,\ s \in Task \text{ and } ws \in WSDL,$$

where s indicates the task the Web Service described by the document ws supplies.

The set of registered Web Services is represented by the set of entries (with an associated weight) stored in the DS. To discover a Web Service (i.e., to obtain its WSDL), an agent must perform a rd operation. For instance, an agent that is willing to discover a Web Service supplying task s performs $rd(< s, null >^{[\#]}_{[?]})$; the return value will contain, in the second field, the WSDL associated to a Web Service supplying task s. In this case the discovery is performed in the default space (default field identifiers # and ? are used), but, as it will be discussed in the following Section, the discovery can be addressed also in reserved subspaces of the DS.

It is worth noting that, as previously discussed, we exploit XML technologies to define entries (and templates) as well as to implement the matching evaluation between data fields. Therefore, our model does not preclude the discovery of a Web Service satisfying a particular kind of data in the WSDL field of the tuple.

Finally, in our setting, the weights associated to each entry in the DS represent the quality level of the Web Service associated to that entry, e.g., the performance degree of the Web Service. In this paper we do not discuss the mechanism/principle used to assign weights to Web Services (e.g., based on the frequency of provided services): we consider the study of the methodology to establish the weights an orthogonal problem that can be tackled separately. We just assume that this values are determined by an external entity and that they can also change during the evolution of the system by invoking update operations (that we will define in the following) on the registry. The use of weights in the Web Service discover enables a workload distribution of the Web Service invocations that ensures some kind of quality of service requirement (depending on the meaning of the weights attributes).

3.1.1 Limiting the access to Web Services

In several E-commerce applications it is fundamental to allow the execution of specific tasks only to a subset of privileged users (see also Section 4), or to reserve a Web Service to a specific set of users. Control fields, that provide a manner to implement access control mechanisms for read and removal operations, can be used to satisfy these goals.

Let us consider, as an example, a simple case in which there is a service (identified by task s) that manages the booking of airline flights that is used by Internet users and by booking agencies. In order to allow a more efficient service to agencies, one or more Web Services supplying task s are reserved for them; this can be done by publishing the entry representing those Web Services in a partition identified by c_s, known only by the booking agencies. Observed that, from the numerical viewpoint, it is reasonable to presume the booking agencies to be fewer than users, the idea is to reserve a subset of the Web Services to the agencies only: in this way we should ensure that the efficiency of the reserved service is better than the one experienced by the Internet users.

3.2 Registration and deregistration of Web Services

The Web Service discovery phase commented in the previous Section assumes a specific structure of the entries. In order to be coherent with this assumption, the registration phase of a Web Service corresponds to an output of the entry containing the information about the service. More precisely, a Web Service, say WS_A, described by a WSDL document $ws \in WSDL$ and providing task $s \in Task$ that is willing to register, performs the following operation:

$$out(< s, ws >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}, w),$$

where $c, c' \in Partition$, $k, k' \in APartition$ and $w \in Weight$.

Clearly, as previously discussed, the choice of control field values affects the visibility of the described Web Service. Furthermore, the use of control fields indirectly provides also a trust measure of the registered Web Services. Indeed, any agent can write into the public space of the DS and, therefore, any agent can register malicious or inexistent Web Services in that space. Control fields can be exploited to register Web Services in reserved spaces where it is assumed that agents having access are trusted. Other aspects related to the use of control fields will be discussed in the following.

The deregistration of a Web Service can be done by simply removing from the DS the entry representing that Web Service. For example, to deregister the service WS_A a in operation is performed, more precisely $in(< s, ws >^{[c']}_{[k']})$.

The update of a Web Service in the registry consists of a removal operation followed by the output of an entry (with an associated weight) containing the updated information. For example, to update the weight of the Web Service WS_A to a new value, say w', the $in(< s, ws >^{[c']}_{[k']})$ followed by $out(< s, ws >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}, w')$ must be performed. Finally, following the same idea used to update the weight, we can define a similar procedure to move a Web Service from one subspace of DS to another one, e.g., to improve the performance of the Web Services having access in the latter.

In order to guarantee the consistency of the published Web Services, i.e. to allow only to the Web Service maintainer (e.g., the Web Service itself) the deregistration or the update of a registered Web Service, we present a possible manner to limit the access to the entries that exploits the access permission for in operations. As explained in detail in [4], if the Web Service keeps secret the access keys (partition and asymmetric partition identifiers) for in operations, it is the only Web Service that can remove (and then update) the service from the registry. Let us consider, as an example, a Web Service registered by performing $out(< s, ws >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}, w)$. If $\overline{k'}$ is known only by the Web Service itself, it is the only one allowed to perform the update of the weight, because the knowledge of $\overline{k'}$ is needed; indeed, the procedure to be performed for updating the weight to the value $w' \in Weight$ is an $in(< s, ws >^{[c']}_{[\overline{k'}]})$ followed by $out(< s, ws >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}, w')$.

4. AN EXAMPLE OF AN APPLICATION: A SOFTWARE FACTORY

The effectiveness of the proposed solution is evaluated by exploiting it in a real application. Let us consider the case of an application that offers distinct set of services for distinct categories of users. For instance, a software factory offering products to different users, e.g. partners (or privileged consumers) and generic consumers. Consumers can access the download service of the distributed products releases, while partners can also access the products releases not yet distributed (because, e.g., subject to testing phase). Moreover, in order to guarantee faster downloads, some Web Services (distributed, e.g., over the Internet) are reserved to the partners. In this case, we have to support both separate allocation of Web Services to different categories and a limited access to some Web Services (i.e. those providing the download of unstable software releases).

Let s (resp. s') be the task offering the download of distributed releases (resp. unstable releases), $SU = \{ws_1, \ldots, ws_n \mid ws_i \in WSDL, 1 \leq i \leq n\}$ and $SP = \{pws_1, \ldots, pws_m \mid psw_i \in WSDL, 1 \leq i \leq m\}$ be the set of Web Services providing task s for generic consumers and partners, respectively, and $SP' = \{pws'_1, \ldots, pws'_l \mid pws_i \in WSDL, 1 \leq i \leq l\}$ be the set of Web Services supplying task s'. We also define $weight$ be a function that, given a WSDL identifying a Web Service, returns its weight. Let $c, c' \in Partition$ be the partition value identifying the space reserved for consumers and for partners, respectively. We also assume that c is known only by the consumers (that can acquire its value, e.g., after a registration phase), while the knowledge of c' is limited to the partners.

The registration procedure of Web Services composing the application follows:

- $out(< s, ws >^{[c]_{rd}[c]_{in}}_{[?]_{rd}[?]_{in}}, weight(ws))$, for any $ws \in SU$ (registration of Web Services for consumers).
- $out(< s, ws >^{[c']_{rd}[c']_{in}}_{[?]_{rd}[?]_{in}}, weight(ws))$, for any $ws \in SP$ (registration of Web Services supplying task s for partners).
- $out(< s', ws >^{[c']_{rd}[c']_{in}}_{[?]_{rd}[?]_{in}}, weight(ws))$, for any $ws \in SP'$ (registration of Web Services supplying task s' for partners).

To discover these three class of Web Services a rd operation is to be performed, more precisely:

- $rd(< s, null >^{[c]}_{[?]})$, to discover a Web Service supplying task s for consumers;
- $rd(< s, null >^{[c']}_{[?]})$, to discover a Web Service supplying task s for partners;
- $rd(< s', null >^{[c']}_{[?]})$, to discover a Web Service supplying task s' for partners.

It is worth noting that the balanced access to the download services is guaranteed by the weights $weight(ws)$ associated to each Web Service ws. Only partners can perform the latter two operations because the knowledge of c' is needed, satisfying so the goals described above.

Finally, asymmetric control fields (above left to default values) can be exploited to refine access permissions. For example, it can be reasonable to guarantee users that registered services are indeed provided by the software factory. This can be done by exploiting asymmetric partition fields associated to the rd access permission. The fact that the entity that wrote the entry in the tuple space owned the key k stored in the rd asymmetric partition field authenticates its identity: users (that own the "public" key $\overline{k}$) know that the only entity possessing the "private" co-key k is the software factory.

5. RELATED WORK AND CONCLUSION

In this paper we have presented an extension of a coordination model that supports probabilistic accesses to the entries stored inside the DS. This new feature has been exploited to present a possible Web Services registry providing a discovery protocol supporting: i) the discovery of Web Services at run-time that takes into account the modifications of the system; ii) quality of Service (by exploiting weights on entries); iii) a mechanism to limit the access to Web Services (by exploiting control fields).

It is worth to remark that the proposed solution does not compete with the standard UDDI protocol but, rather, aims at covering the most critical requirements by integrating it with a coordination infrastructure.

The coordination model we have exploited to implement the proposed Web Service registry extends `WSSecSpaces` by introducing probabilistic accesses to the entries. Other proposals providing a coordination platform in the context of Web Services are [3] and Ruple [15]. The former is an implementation of a Web Service that allows for a location-based coordination of Web Services (i.e. different locations support different coordination spaces); security is not supported. Ruple is an implementation of a coordination platform for Web Services and supports security; access control mechanism on entries are based on digital certificates that do not discriminate between the rd and the in operations that, as pointed out in the examples of Section 3, it is a feature improving the level of trust about the information of registered Web Services.

As future work we plan to define a complete process algebra for the specification of E-Commerce systems based on our coordination infrastructure. To pursue this goal, we intend to follow the approach reported in [12], where a process algebra is proposed as a framework to be used for the formal modeling and verification of e-barter systems (multi-agent systems based on agents exchanging goods). Moreover, we intend to consider an extending setting where we associate with entries in the registry service more than one weight attribute in order to describe several indexes of quality of the service. In this way, the discovery phase can be parameterized on more than one degree of quality, allowing clients to address the discovery of Web Services with specific peculiarities. Finally, we intend to extend the proposed coordination model in order to allow an agent to access more than one partition at a time; this can be achieved by associating to each data retrieval operation a list of partition identifiers (instead of a single one) that is taken into account while selecting the search space.

6. REFERENCES

[1] Business Process Modeling Language (BPML). [www.bpmi.org].

[2] Universal Description, Discovery and Integration for Web Services (UDDI) V3 Specification. http://uddi.org/pubs/uddiv3.htm.

[3] P. Álvarez, J.A. Bañares, P.R. Muro-Medrano, F.J. Nogueras, and F.J.Zarazaga. A Java Coordination Tool for Web-service Architectures: The Location-Based Service Context. In *Scientific Engineering for Distributed Java Applications*, volume 2604 of *LNCS*, pages 1–14. Springer-Verlag, 2003.

[4] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Secspaces: a data-driven coordination model for environments open to untrusted agents. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68.3 of *ENTCS*, 2002.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL 1.1). [www.w3.org/TR/wsdl], W3C, Note 15, 2001.

[6] P. Ciancarini, R. Tolksdorf, and F. Zambonelli. Coordination Middleware for XML-centric Applications. In *Proc. ACM/SIGAPP Symp. on Applied Computing (SAC)*. ACM Press, 2002.

[7] World Wide Web Consortium. XML Schema. W3C Recommendation, 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[8] Microsoft Corporation. Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp].

[9] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS 1.0), 2002. [http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/].

[10] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.

[12] Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio. A Multi-Agent System for E-Barter Including Transaction and Shipping Costs. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 587–594. ACM Press, 2003.

[13] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications. In *Proc. of ACM Symposium on Applied Computing (SAC'04)*. ACM Press, 2004.

[14] B. Schneier. *Applied Cryptography.* Wiley, 1996.

[15] Rogue Wave Software. Ruple. http://www.roguewave.com/developer/tac/ruple/.

[16] S. Thatte. XLANG: Web Services for Business Process Design. Microsoft Corporation, 2001.

[17] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for Coordination in Web-based Systems. In *Proc. of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises WET ICE.* IEEE Computer Society, Press, 2001.