A Dependable Infrastructure for Cooperative Web Services Coordination

Eduardo A. P. Alchieri[1] Alysson Neves Bessani[2] Joni da Silva Fraga[1]
[1]DAS, Federal University of Santa Catarina, Florianópolis, Brazil
[2]LaSIGE, University of Lisbon, Faculty of Sciences, Lisbon, Portugal

Abstract

A current trend in the web services community is to define coordination mechanisms to execute collaborative tasks involving multiple organizations. Following this tendency, this work presents a dependable (i.e., intrusion-tolerant) infrastructure for cooperative web services coordination that is based on the tuple space coordination model. This infrastructure provides decoupled communication and implements several security mechanisms that allow reliable coordination even in presence of malicious components. This work also investigates the costs related to the use of this infrastructure and possible web service applications that can benefit from it.

1 Introduction

The Web Services technology, an instantiation of the service oriented computing paradigm [6], is becoming a *de facto* standard for the development of distributed systems on the Internet. The attractiveness of web services is its interoperability and simplicity, based on technologies widely used in the web, like HTTP and XML. Its attributes have been the motivation for many industrial and academic efforts for developing concepts and models for distributed applications based on the service-oriented paradigm.

Also, many efforts have appeared to define adequate forms of services composition in order to execute complex tasks. These compositions aim the cooperation in the execution of tasks involving multiple organizations [6, 25]. Specifications like *WS-Orchestration* [1] and *WS-Choreography* [8] address exactly this problem, specifying mechanisms for definition and execution of tasks that involve several web services.

Current approaches, as the two cited above, aim the web services integration through the specification of message flows among the services (control-driven coordination [24]). Another approach, which complements the first one, is the web services coordination through the use of a shared data repository (data-driven coordination [24]). In this approach, the cooperating services communication is done through a shared data repository, that can be used as a data storage or as a coordination mediator, providing decoupled communication. *Tuple spaces* [16] are a popular data-driven coordination model.

In the tuple space model, processes interact through a shared memory abstraction, the tuple space, where generic data structures, the tuples, are stored and retried. The basic operations supported by the tuple space are insertion, reading and removal of tuples. The main attractiveness of this model is its support for communication that is decoupled both in time (communicating processes do not need to be active at the same time) and space (communicating processes do not need to know each others locations or addresses) [10]. Moreover, this model provides some synchronization power through special operations provided by the tuple space that allow applications to solve important problems like leader election, consensus and mutual exclusion.

There has been some research about the integration of web services using the tuple space model (e.g., [7, 19, 20, 3]). The main advantage of this approach is the decoupling among the cooperating services: not even the services interfaces needs to be completely known. This work follows in the same line and proposes a cooperation infrastructure for web services that provides the inherent benefits of the tuple spaces model but that is also dependable.

The infrastructure proposed in this paper, called WS-DEPENDABLESPACE (WSDS), extends previous works of the authors on tuple space dependability [4, 5], incorporating new components that provide the integration of a dependable tuple space on the web services world. The WSDS architecture relies on stateless gateways that execute operations on a dependable tuple space, forwarding client requests from web service clients to it and vice-versa. Several mechanisms have been introduced in this architecture in order to tolerate accidental faults (like crashes and software bugs) as well as malicious attempts to disable the operation of system components (like attacks and intrusions) [28]. Moreover, WSDS maintains all the dependability properties of a dependable tuple space [4] *without further extensions to the web services core specifications*.

The main contributions of this work are: the design and implementation of the WSDS infrastructure, which is the first dependable web services data-centered coordination service; evaluation of the cost to access this type of infrastructure through an analysis of the operations latency and its

causes; and an analysis of some real applications where this type of infrastructure can be used, as well as its relation with the main standards for cooperation among web services.

2 Tuple Spaces

A *tuple space* [16] can be seen as a shared memory object that provides operations for storing and retrieving ordered data sets called *tuples*. A tuple t in which all fields have a defined value is called an *entry*. A tuple with one or more undefined fields is called a *template* (denoted by a bar, e.g., $\bar{t}$). An undefined field is represented by a *wild-card* ('*'). Templates are used to allow content-addressable access to tuples in the tuple space. An entry t and a template $\bar{t}$ *match* if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. For example, template $\langle 1, 2, * \rangle$ matches any tuple with three fields in which 1 and 2 are the values of the first and second fields, respectively. A tuple t can be inserted in the tuple space using the $out(t)$ operation. The operation $rd(\bar{t})$ is used to read tuples from the space, and returns any tuple of the space that *matches* the template $\bar{t}$. A tuple can be read *and* removed from the space using the $in(\bar{t})$ operation. The in and rd operations are blocking. Non-blocking versions, inp and rdp, are also usually provided [16].

To increase the synchronization power of the tuple space, we consider also the $cas(\bar{t}, t)$ operation (conditional atomic swap) [27], that works like an indivisible execution of the code: **if** $\neg rdp(\bar{t})$ **then** $out(t)$ ($\bar{t}$ is a template and t an entry). The cas operation is important mainly because a tuple space that supports it is capable of solving the consensus problem [27, 5], which is a building block for solving many important distributed synchronization problems like atomic commit, total order multicast and leader election.

3 DEPSPACE: A Dependable Tuple Space

A tuple space is dependable if it satisfies the *dependability attributes* [2]. The relevant attributes in this case are: *reliability* (operations on the tuple space have to behave according to their specification), *availability* (the tuple space has to be ready to execute the requested operations), *integrity* (no improper alteration of the tuple space can occurs), and *confidentiality* (the content of tuple fields can not be disclosed to unauthorized parties). DEPSPACE [4] is an implementation of a dependable tuple space that satisfies these properties using a combination of some mechanisms, which are described below[1].

Intrusion-tolerant replication. The most basic technique used in DEPSPACE is *replication*, i.e., the tuple space is maintained by a set of n servers in such a way that failure of up to f of them does not impair the reliability, availability and integrity of the system. The idea is that if some servers fail, the tuple space is still ready (availability) and operations work correctly (reliability and integrity) because correct replicas mask the misbehavior of the faulty ones. A simple approach for replication is *state machine replication* (SMR) [26]. SMR guarantees *linearizability* [17], which is a strong form of consistency in which all replicas appear to take the same sequence of states. SMR delivery properties are guaranteed by the BYZANTINE PAXOS protocol [12].

Access control. This mechanism allows only authorized clients to perform operations in the tuple space. Access control is fundamental to preserve integrity and confidentiality properties of a dependable tuple space. DEPSPACE provides two types of access control:

- **Credential-based:** For each tuple inserted in DEPSPACE, it is possible to define the credentials required for having access to this tuple (i.e., read or remove). These credentials are provided by the process that inserts the tuple in the space. There is also another level of access control: it is possible to define what are the credentials required to insert tuples in the space.

- **Fine-grained security policies:** DEPSPACE supports the enforcement of fine-grained security policies for access control [5]. This kind of policies takes into account three types of parameters to decide if an operation can be executed or not: the invoker *id*; the operation and its arguments; and the tuples currently in the space. An example is the policy: *"an operation $out(\langle CLIENT, id, x \rangle)$ can only be executed if there is no tuple on the space that matches $\langle CLIENT, id, * \rangle$"*. This policy does not allow the insertion of two tuples representing clients, with the same value on the second field (client identifier).

Credentials required for tuple insertion and security policies are always defined during the tuple space creation.

Finally, another important feature of DEPSPACE is its support for multiple logical tuple spaces: the system has administrative interfaces that allow (logical) tuple space creation. Logical tuple spaces have no relationship with other (logical) tuple spaces.

4 WS-DEPENDABLESPACE

This section describes the WSDS infrastructure. First, the system model is presented. After, the architecture and the system functionality principles are discussed and, in the end, specific mechanisms integrated in the architecture are shown.

4.1 System Model

The processes of the system are divided in three sets: n_r DEPSPACE servers $U = \{s_1, ..., s_{n_r}\}$, n_g access *gateways* $G = \{g_1, ..., g_{n_g}\}$ and an unknown set of clients

[1]The DEPSPACE also uses cryptography to satisfy the confidentiality property [4]. Due to space limitations, this functionality is not described here, neither its integration with the WSDS infrastructure.

$\Pi = \{c_1, c_2, ...\}$. The gateways are the only processes of the system that export WSDL interfaces, i.e., they are web services. The communication among clients and gateways is done using SOAP messages, and among gateways and servers (and among servers) using authenticated reliable point-to-point channels (these channels can be implemented using SSL/TLS [13]).

We assume an eventually synchronous system model [14]: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time), so that every message sent by a correct process to another correct process at instant $u >$ GST is received before $u + \Delta$. Δ and GST are unknown. The intuition behind this model is that the system can work asynchronously (respecting no delay bounds) most of the time but there are stable periods in which the communication delay is bounded. Notice that this model reflects the behavior of the Internet: the communication latency is stable most of the time, however a limit for this value does not exist. We also assume that all local computations require negligible time.

System processes are subject to Byzantine failures [18], i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Processes that do not follow their algorithm are said to be faulty. A process that is not faulty is said to be correct. We assume fault independence for servers, i.e., the probability of a server being faulty is independent of another server being faulty. This assumption can be substantiated in practice through the use of diversity [22]. WSDS works correctly while a bound of up $f_r \leq \lfloor \frac{n_r+1}{3} \rfloor$ DEPSPACE servers (the best resilience for this type of replication [12]), $f_g \leq n_g - 1$ *gateways* (at least one correct) and an arbitrary number of client fails.

Our last assumption is that each process handles a pair of keys (public and private from an asymmetric cryptosystem), used for generation and verification of digital signatures. Private keys are known only to each owner, however all processes know all public keys (through certificates).

4.2 WSDS Architecture

Figure 1 presents the WSDS architecture. In this figure it can be seen that the gateways connect clients and DEPSPACE servers. To do this, the gateways publish their WSDL interface in a UDDI repository. Thus, the clients are able to access them. As already mentioned, the clients and servers handle a key pair (public and private keys). All public keys are available through certificates.

The main component introduced in this architecture is the *gateway* web service, that works as a bridge among clients (of the coordination service) and the dependable tuple space. The gateway receives SOAP messages sent by clients and transforms them in DEPSPACE requests.

Thus, before accessing the tuple space, a client must find one or more gateway address on a UDDI service[2]. Thereafter, the client sends its requests to one of the gateways, which, in turn, forwards it to the DEPSPACE using a total order multicast protocol [12, 4]. DEPSPACE servers process the request and send the response to the gateway, which collects $n_r - f_r$ responses from different servers and forward this set of responses to the client. The client determines the response of its request by verifying which response was replied by at least $f_r + 1$ servers (at least one correct server).

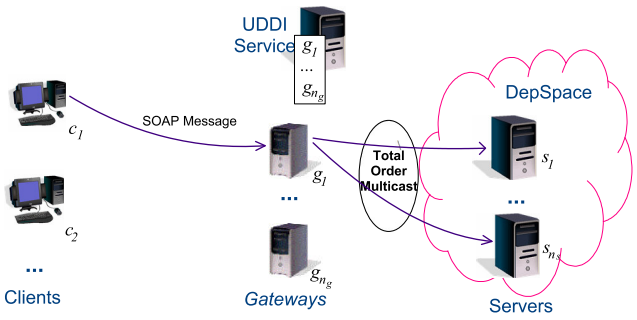

Figure 1. WSDS architecture.

The gateway does not execute any processing on the content of requests or responses, only the transformations between the two "worlds" (SOAP and DEPSPACE), previously described. Moreover, this service is stateless, i.e., it does not have state and thus there is no need of any synchronization among the n_g gateways of the system.

4.3 Dealing with Faulty Gateways

Although the apparent simplicity of the architecture depicted in Figure 1, the use of access gateways to make DEPSPACE accessible from the web services world makes the coordination system susceptible to several security problems. These problems, together with the solutions used in WSDS, are described in this section.

4.3.1 Authenticity and Integrity of Messages

The first problem to be solved in the WSDS architecture (Figure 1) is that a faulty gateway can request the execution of false requests (that were not sent by any client). Moreover, it would be able to modify the requests, modifying some of their data (ex., parameters and/or operation types). In the same way, it would be able to forge or modify replies of servers. The main point here is to guarantee that only requests (unchanged) made by clients and that only responses (unchanged) produced by servers are processed, avoiding all kinds of *man in the middle* attacks.

Moreover, the client needs to send its credentials together with the requests, then the servers will be able to

[2]To improve the system availability, the gateways can be registered in more than one UDDI service.

verify the client access permission, i.e., if the client can access the space and/or the tuple. These credentials are sent to the servers through a digital certificate, which is also used to prove the request authenticity (i.e., the client's digital signature). Thus, each client signs its requests before sending them. Servers only execute an operation if the signature is valid, in accordance with the corresponding certificate.

To prove the authenticity of the replies, the same approach is used. Each reply must be signed by a server with its private key. Thus, clients will be able to verify the authenticity of the replies, using the public keys of the servers, also contained in certificates (sent together with the replies). These signatures guarantee authenticity and integrity in end-the-end communications, i.e., nothing can be modified by gateways without being detected by DEPSPACE servers or clients. The certificates sent together with requests and responses are verified by standards procedures (e.g., X.509 or SPKI public key infrastructures).

4.3.2 Incomplete Execution of Requests

Notice that nothing guarantees that a client will obtain the response of its request if it is accessing a faulty gateway. Also, in the execution of removal operations, a malicious gateway can remove tuples from DEPSPACE and not send them to the client. Thus, these tuples will "disappear" from the space without being returned by a requesting client.

To solve this problem, a timer is associated with each request sent by the client. If a timeout occurs before that the response is obtained, the client sends the request to other f_g gateways (one at time), ensuring that eventually it has accessed at least one correct gateway. In this way, the execution of the request ends when the client receives the first set of valid responses (from some gateway) and determines the response of the request. This mechanism is activated whenever a client can not determine a response, which is caused either by timeout occurrence or by problems in the responses signatures.

In blocking operations (in and rd) it is not possible to determine why the client does not have received the replies, i.e., if the gateway accessed is faulty or if there is no tuple in the space that matches the template. Thus, it is not possible to use timeouts in these operations. For solving this problem, the client needs to send these requests to $f_g + 1$ different gateways and determine the response as previously described (when a timeout occurs).

4.3.3 Duplicated Requests

Since some requests can be sent to more than one gateway due to the mechanism described in the previous section, it is possible that a request is sent more than once to the DEPSPACE servers. Moreover, a malicious gateway can request the execution of an operation already executed by the DEPSPACE (*replay attacks*). In these cases, duplicated requests need to be discarded in order to maintain the consistency of the tuple space. To make this in a safe way, two mechanisms were implemented:

1. Each request has an unique identifier, composed by the client identity plus a sequence number. This number is inspected by DEPSPACE servers before the request execution, to verify if the request was already executed;
2. Each server has a buffer that stores replies to the previous request of each client. Thus, for each client, a request is executed by a server only if its sequence number is one unit greater than the last request executed (the reply is stored in the buffer). If the request has the same identification of the last executed, only the reply stored in the buffer is sent to the gateway. In all other cases, the request is discarded.

Using these two mechanisms, a client can not send a request without completing the previous one[3]. Besides preventing that a request is executed more than once, this strategy also solves the tuple disappearance problem (caused by incomplete execution of requests). In fact, when the client requests the re-execution of an operation through another gateway, the same reply (with the same tuple) is in the servers buffers ready to be sent to the client.

4.4 Implementation

The prototype was developed in Java and uses the DEPSPACE implementation [4]. Cryptographic primitives were provided by the default provider of JCE (Java Cryptography Extensions) – version 1.5. Signatures were implemented by the algorithms *SHA-1* and *RSA*, for hashes and asymmetric cryptography, respectively. Reliable authenticated point-to-point channels were implemented by TCP sockets and session keys based on *HmacSHA-1* algorithm.

The gateways were implemented using Axis 1.3 (`http://ws.apache.org/axis/`), an open-source SOAP protocol implementation, and deployed on *Tomcat* 5.5.20 J2EE container (`http://tomcat.apache.org/`).

All additional mechanisms required in our architecture (e.g., signature verifications and message identifiers) were integrated in DEPSPACE through *interceptors*. The implementation of interceptors on DEPSPACE was inspired by CORBA portable interceptors [23], and was integrated to the system architecture (see [4]) as an additional layer between replication and access control layers. Additional data, necessary to execute a request (e.g., signatures), is sent through an abstraction (context) that is part of the messages.

5 Performance Analysis

In this section, we analyze the WSDS performance. Considering the latency of each operations which is determined by the following equation: $L_{wsds} = L^{c}_{sign} +$

[3]This limitation can be relaxed for at most k requests if the servers store the last k replies to each client.

$L^{c\rightarrow g}_{comm} + L^{g\rightarrow s}_{tom} + L^{s}_{ver} + L^{s}_{op} + L^{s}_{sign} + L^{s\rightarrow g}_{comm} + L^{g\rightarrow c}_{comm} + (f+1)L^{c}_{ver}$.

The sources of latency represented in this equation are: L^{c}_{sign} - time to request signature; $L^{c\rightarrow g}_{comm}$ - latency for sending the request to the gateway; $L^{g\rightarrow s}_{tom}$ - time for forwarding the request to servers and its ordination; L^{s}_{ver} - time for verifying request signature; L^{s}_{op} - operation execution; L^{s}_{sign} - time to sign the reply; $L^{s\rightarrow g}_{comm}$ - time to send the reply to the gateway; $L^{g\rightarrow c}_{comm}$ - latency for forwarding replies to the client; and finally, L^{c}_{ver} - time to verify the reply integrity.

Considering that clients and servers spend similar amount of time in signature and verification operations, that the cost of local operations in tuple space is negligible (i.e., $L^{s}_{op} = 0$), and knowing that $L^{c\leftrightarrow g}_{comm} = L^{c\rightarrow g}_{comm} + L^{g\rightarrow c}_{comm}$ represents the communication between client and gateway, the latency of WSDS can be expressed as: $L_{wsds} = 2L_{sign} + L^{c\leftrightarrow g}_{comm} + L_{tom} + L^{s\rightarrow g}_{comm} + (f+2)L_{ver}$.

Following this, the latency in DEPSPACE access is $L_{ds} = L_{tom} + L^{s\rightarrow g}_{comm}$. Thus, the additional cost related to WSDS consists in: request and responses signatures; extra communication step to access the gateway; verification of the request and $f+1$ responses authenticity.

In order to quantify this latency, some experiments were conducted in a local area network. The execution environment was composed by four Athlon 2.4GHz PCs with 512M of memory and running Linux (kernel 2.6.12). They were connected by a 1Gbps switched Ethernet network. The Java runtime environment used was Sun's JDK 1.5.0_06 and the just-in-time compiler was always turned on.

The system was configured with four DEPSPACE servers (one server per machine), one gateway and one client, executed in different machines (together with one DEPSPACE server). We executed each operation 1000 times and obtained the mean time discarding the 5% values with greater variance.

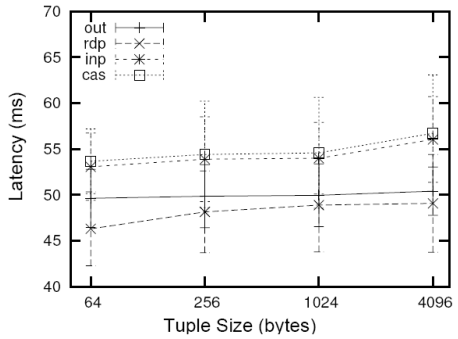

Figure 2. Latency of the WSDS.

Figure 2 presents the latency observed in the execution of each one of the four operations supported by the WSDS (varying the tuple size). The figure shows that, even with large tuples, the decrease is reasonably small in the WSDS performance, e.g., increasing the tuple size 64 times (64 to 4096 bytes) causes a decrease of about 7% in the system performance. Moreover, the variance presented ($2-6ms$) is appropriate for this class of system (web services).

Figure 3 presents the costs of each phase of the protocol (considering a tuple of 64 bytes). The table shows also the percentage of the cost of each phase in the total latency observed by clients, in accordance with the latency equation presented. Signatures processing represent the bigger contribution (about 55%) to the total latency of the operation execution. If we exploit these results considering a large scale network like the Internet, where the communication latencies are at least 100 times higher, the communication latency is expected to be much greater than 40% of the total access time observed in our experiments (communication plus latencies). In these scenarios, signature costs tend to dilute.

Latency	Cost (ms)	$\%L_{wsds}$
$L^{s\rightarrow g}_{comm}$	0.63	1.27
$L^{c\leftrightarrow g}_{comm}$	13.53	27, 27
L_{sign}	13.51	54.45
L_{ver}	0.85	5, 13
L_{tom}	5.90	11.88
L_{wsds}	49.63	100
L_{ds}	6.53	13.15

Figure 3. Latency costs.

6 Applications

In this section we present some examples of services that can be built over WSDS. The objective is not only to show that the WSDS is useful in the resolution of some different problems, but also to show that the WSDS is sufficiently generic to be used by many applications.

6.1 Secure Biddings

An application to manage biddings can be easily implemented over WSDS. Using WSDS, a consumer interested in some service (or product) inserts a tuple in the space, describing the characteristics of the service to be contracted (or product to be bought). Then, service providers (that can have been previously registered in WSDS) also insert a tuple, describing its proposal for service execution. Finally, after the proposal period, the consumer reads all proposal tuples and does one of the following: *(i.)* chooses the provider with best proposal, ending the bidding; *(ii.)* does not choose some proposal, because it decides that all proposals are inadequate in terms of quality of service or price (specified in the bidding description); *(iii.)* makes a summary of proposals and publishes it in the space, initializing another round of biddings (as an auction, where service providers could do new proposals lowering the price and/or offering better services). Figure 4 illustrates this application and shows tuples that can be inserted in WSDS.

This application needs some security mechanisms: *(i.)* a service provider can not be allowed to access the proposals

of others providers; *(ii.)* a provider can not be allowed to do more than one proposal; *(iii.)* if a previous registration is necessary, an unregistered provider can not be able to make proposals; and other requirements specific to each bidding. Using WSDS features, more specifically the access control mechanisms, it is possible to guarantee that: requirement *(i.)* can be implemented under the presentation of credentials for reading the proposals tuples; and requirements *(ii.)* and *(iii.)* can be implemented through fine-grained security policies like, for example: *"if exists a proposal of the provider A for the bidding X in the space, the inclusion of a new proposal of this provider for X is not allowed"* and *"the provider A can only insert a proposal if there is a tuple* $\langle PROVIDER, A \rangle$ *in the space"*[4], respectively[5].

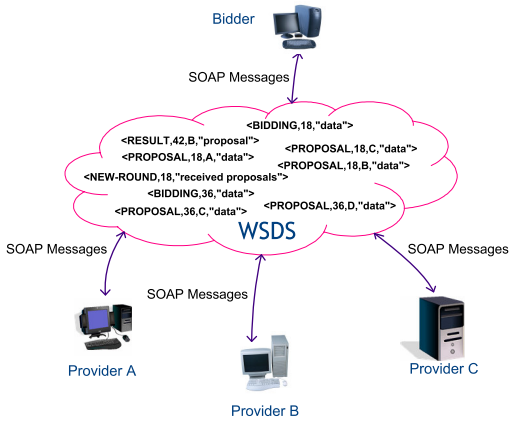

Figure 4. Secure biddings.

Other examples of distributed applications in which a tuple space is used as a decoupled communication mechanism are: the master-worker pattern, a classical parallel programming design pattern used to distribute tasks in cluster of machines, can be easily implemented over a tuple space [11]. If this space is deployed on the Internet, the same programming model can be used to distribute tasks in a computational grid [15]. The high availability and the fine-grained access control provided by WSDS make it able to ensure dependability properties even in the presence of faults and intrusions.

6.2 Data Sharing among Services

The applications that invoke several web services, in order to execute complex tasks, need to share information among these services. These applications may use a *shared database* (accessed by several services) or need the *session information* of each task to be included in all messages exchanged during the task execution. As the web services are not deployed in the same administrative domain, the existence of a database, accessed from the Internet, can incur into serious security problems. Moreover, the cooperating services becomes dependent of a single point of failure (the database). On the other hand, the session information exchange can demand that a high amount of information has to be sent together with all messages and, in this way, affecting the system performance. If a service like WSDS is available, cooperatives services can store the session information in a shared and dependable tuple space. The properties of WSDS ensure: *(i.)* reliability and availability of this repository (due to intrusion-tolerant replication); *(ii.)* space- and tuple-level access control; and *(iii.)* universal accessibility (through the gateways, that are web services).

A travel agency is an example of this type of system (Figure 5), where a set of web services are used by the agency to allocate the resources required in a vacation trip (e.g., air transport, accommodation, car renting, tours planning). To execute this automatically, information about the traveller, its preferences and its schedule must be available to those services. The agency can contract each service through secure bidding protocols (like the one presented in previous section) to find the best service at a lower price. Moreover, when a service executes its part of the travel planning it can modify that data (e.g., the traveller schedule).

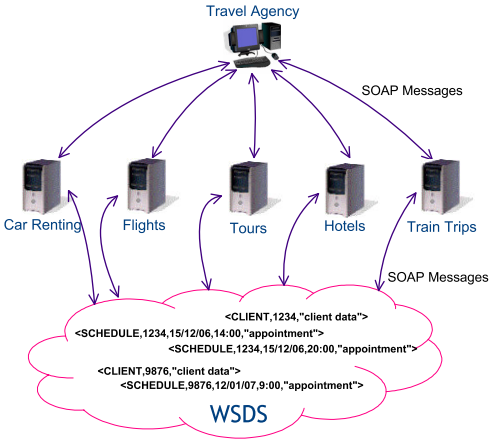

Figure 5. Travel agency.

This example is interesting because it explores the WSDS synchronization capabilities: the SCHEDULE tuples and the tuple space operations with synchronization power (cas and inp) can be used to solve concurrency problems (i.e., different services specifying simultaneous activities in the space).

The security policies can define what information (tuple) each service is allowed to access. Also, policies can specify that only the travel agency is allowed to remove a commitment or cancel a resource allocation (accommodation, car,...). Moreover, the tuples with client data must be confidential, in order to maintain information privacy.

Other examples of this class of services are: integration

[4]This rule must be complemented with another one which specifies that only an administrator will be allowed to insert this type of tuple.

[5]As already mentioned, DEPSPACE also supports confidentiality of the tuples [4]. This functionality was integrated in WSDS, however it is not discussed here due to space limitations.

of intrusion detection systems [29] – where the summary of events observed by different intrusion detection systems are shared (in the tuple space) to allow the correlation of suspect activities; and the support to virtual enterprises [7] – where a space is used by partners that compose the virtual enterprise to share information.

7 Relations with WS Specifications

There are many specifications developed (or in development) that aim the integration of web services. This section presents the relationship between WSDS and some of these specifications.

WS-Coordination. This specification defines a generic model that can be used by several web services in order to coordinate the execution of some distributed task [9]. The main component of this model is the *coordination context*, an abstraction responsible for managing information about the coordination of participants. At present, this model already was used to implement distributed transactions involving web services. The tuple space abstraction, provided by WSDS, can be implemented as an instance of WS-Coordination, where the coordination context is the logical tuple space and services registered in the context can be understood as the clients able to interact with the space.

WS-Orchestration. Orchestration of web services consists in the coordination of these services by using an orchestration engine. The basic idea is to define the invocations flow (i.e., the sequence that services will be invoked), using a coordination language, as WSBPEL (*Web Services Business Process Execution Language*) [1]. The travel agency (Section 6.2) is an example of task that can be implemented using orchestration. Currently, there is no standard based on shared data repositories to be used by cooperating web services being orchestrated. In this way, all information required by services must be maintained by the orchestration engine and sent to each service as a parameter of operations. If the amount of shared data is large, this approach can be very inefficient. Thus, a dependable data repository is useful and its integration with WSBPEL language is easy, i.e., all orchestrated services access shared data on WSDS.

WS-Choreography. Choreography is a composition of web services that allows: a formal definition of the interactions among the web services; the verification of the correctness (e.g., if it is deadlock-free); and code generation for the interactions [8]. The choreography differs from the orchestration in at least two aspects: it is distributed (while the orchestration has a coordinator – the orchestration engine); and the choreography languages (as the WSCL – *Web Services Choreography Language*) are used to specify the interactions required during the tasks execution, and not to execute the coordination (differing from the orchestration languages, such as WSBPEL) [25]. The choreography aims to make these interactions less susceptible to errors, through a more rigorous specification of the relations among the services. However, this technology does not guarantee that the interactions will occur (in execution time) as specified. The use of a mediator as WSDS fills this gap. The interactions can be controlled through tuples inserted and retried from the tuple space. Policies can be generated from the specified choreography (described in WSCL) and deployed in the tuple space to ensure that only legal interactions can be carried on, even in the presence of faults and intrusions. Moreover, the use of WSDS allows the implementation of multi-part interactions and the recording of all interactions executed by the services, ensuring the auditability of the system.

8 Related Work

Currently, there are is large effort in the development of standardized mechanisms that allow the cooperation and integration of web services. Some examples are WS-Coordination [9], WS-Orchestration [1] and WS-Choreography [8]. These approaches aim the web services integration through message exchanges, and do not define an abstraction to support shared data storage. This type of abstraction is crucial in applications where a large amount of data is shared or where decoupled communication is needed. WSDS implements this abstraction in a dependable way, i.e., the coordination service provided by WSDS is fault and intrusion-tolerant.

The integration of the tuple space coordination model in the web service environment has become an active research topic in the last years (ex. [3, 7, 19, 20]). These works highlight the advantages that a shared data repository with a well defined interface and some synchronization capacity can provide in the web services coordination [20] and in the workflows execution [3, 7]. An important aspect that is not addressed by most of these works is the dependability attributes [2] required for this service. The dependability must be provided at two levels: *(1)* security mechanisms for access control and *(2)* availability of the coordination service. The majority of the works in this area does not implement any of these levels, emphasizing the integration of the tuple space coordination model and web services standards [3, 7, 20]. WS-SECSPACES [19] is an exception, that contemplates the level *(1)*, providing a model with space (and tuple) access control. WSDS supports a model of access control similar to the one provided by WS-SECSPACES and also supports the definition of fine-grained security policies, that enable the coordination even in the presence of malicious processes [5]. Moreover, WSDS implements mechanisms to supply the level *(2)* of dependability, through a simple and efficient architecture that uses gateways to access a dependable coordination service – DEPSPACE [4]. The resulting system tolerates accidental and malicious fail-

ures in all of its components. These characteristics are provided in accordance with web services standards.

The type of policy enforcement offered by WSDS to protect distributed interactions between web services is similar to the one offered by Moses, a Law-Governed Interaction middleware [21]. Moreover, Moses is not dependable in the sense that it is not fault- and intrusion-tolerant and does not directly provides any means for implementing data-driven coordination.

9 Conclusions

This paper described WSDS, a dependable coordination service that can be used by web services to share data and synchronize their actions. The proposed architecture is based on a dependable tuple space [4] and stateless web services (gateways) that provide access to it. An important feature of this system is that it is completely fault- and intrusion-tolerant. This architecture integrates several dependability and security mechanisms in order to enforce dependability properties like integrity, confidentiality, and availability in a modular way.

The system was implemented using open source tools. A preliminary performance analysis done on a local area network shows that the main latency cost comes from the use of digital signatures. These relative costs are expected to be much small if the system runs on the Internet, where the communication latency is orders of magnitude greater than local area networks.

Acknowledgments This work was partially supported by the EC, through projects IST-2004-27513 (CRUTIAL), by the FCT, through the Multiannual (LaSIGE) and the CMU-Portugal Programmes, and by CAPES/GRICES (project TISD).

References

[1] A. Alves et al. Web services business process execution language, version 2.0. OASIS Public Draft. Availiable at `http://docs.oasis-open.org/wsbpel/2.0/`, Nov. 2006.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, Mar. 2004.

[3] U. Bellur and S. Bondre. xSpace: a tuple space for XML and its application in orchestration of web services. In *Proceedings of the 21st ACM symposium on Applied computing – SAC'06*, pages 766–772, 2006.

[4] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys'08*, pages 163–176, Apr. 2008.

[5] A. N. Bessani, M. Correia, J. S. Fraga, and L. C. Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *IEEE Transactions on Parallel and Distributed Systems*, 2008. to appear.

[6] M. Bichier and K.-J. Lin. Service-oriented computing. *IEEE Computer*, 39(3):99–101, Mar. 2006.

[7] D. Bright and G. Quirchmayr. Supporting web-based collaboration between virtual enterprise partners. In *Proc of the 15th International Workshop on Database and Expert Systems Applications*, 2004.

[8] D. Burdett and N. Kavantzas. The WS-Choreography model overview. W3C Draft. Available at `http://www.w3.org/TR/ws-chor-model/`, Mar. 2004.

[9] L. F. Cabrera et al. Web Services Coordination Specification - version 1.0. Available at `http://www-128.ibm.com/developerworks/library/specification/ws-tx/`, 2005.

[10] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, Feb. 2000.

[11] N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989.

[12] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

[13] T. Dierks and C. Allen. The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments, Jan. 1999.

[14] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, Apr. 1988.

[15] F. Favarim, J. S. Fraga, L. C. Lung, and M. Correia. GridTS: A new approach for fault tolerant scheduling in grid computing. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications - NCA'07*, pages 187–194, July 2007.

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

[17] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[18] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[19] R. Lucchi and G. Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proceedings of the 19th ACM Symposium on Applied Computing – SAC'04*, pages 487–491, Mar. 2004.

[20] Z. Maamar, D. Benslimane, C. Ghedira, Q. H. Mahmoud, and H. Yahyaoui. Tuple spaces for self-coordination of web services. In *Proceedings of the 20th ACM Symposium on Applied computing – SAC'05*, pages 1656–1660, 2005.

[21] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[22] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, University of Lisbon, September 2006.

[23] Object Management Group. The common object request broker architecture: Core specification v3.0. OMG Standart formal/02-12-06, Dec. 2002.

[24] G. Papadopolous and F. Arbab. Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, Aug. 1998.

[25] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, Oct. 2003.

[26] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[27] E. J. Segall. Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing – PDP'95*, pages 320–327, Oct. 1995.

[28] P. Verissimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. Springer-Verlag, 2003.

[29] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of the 11th Network and Distributed Security Symposium – NDSS 2004*, Feb. 2004.