International Journal of Web Services Research , Vol.7, No.2, 2010

A Dependable Infrastructure for Cooperative Web Services Coordination

Eduardo Adilio Pelinson Alchieri
Department of Automation and Systems Engineering
Federal University of Santa Catarina
Florianópolis, Brazil
alchieri@das.ufsc.br

Alysson Neves Bessani
Large Scale Informatics Systems Laboratory
University of Lisbon, Faculty of Sciences
Lisbon, Portugal
bessani@di.fc.ul.pt

Joni da Silva Fraga
Department of Automation and Systems Engineering
Federal University of Santa Catarina
Florianópolis, Brazil
fraga@das.ufsc.br

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 dependable 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.

KEY WORDS:
Dependability, Tuple Spaces, Web Services Coordination

INTRODUCTION

The Web Services technology, an instantiation of the service oriented computing paradigm (Bichier and Lin 2006), is becoming a *de facto* standard for the development of distributed systems on the Internet. The attractiveness of web services are 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.

The service oriented computing - and more specifically Web Services - is a natural evolution of classical concepts such as RPC and technologies like CORBA (Object Management Group, 2002). In the same way of these, web services provided by an organization must be described in an interface that can be understood and invoked by other parts of a distributed system. The main standards for web services, all based on XML, are: SOAP (*Simple Object Access Protocol*) - protocol for exchange messages among clients and services, which can operate over several communication protocols; the WSDL (*Web Service Description Language*) - language used to describe web services; and the UDDI (*Universal Description, Discovery and Integration*) - the repository where the web services are registered in order to be discovered by clients.

The main goal of web services is the interoperability. Thus, 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 (Bichier and Lin 2006; Peltz, 2003). Specifications like *WS-Orchestration* (Alves, 2006) and *WS-Choreography* (Burdett and Kavantzas 2004) address exactly this problem, specifying mechanisms for definition and execution of tasks that involve several web services.

Current approaches, as the cited above, aim the web services integration through the specification of message flows among the services (control-driven coordination (Papadopolous and Arbab 1998)). Another approach, which complements the first one, is the web services coordination through the use of a shared data repository (data-driven coordination (Papadopolous and Arbab 1998)). In this approach, the cooperating services communication is done through a shared data repository, which can be used as data storage or as a coordination mediator, providing decoupled communication. *Tuple spaces* (Gelernter, 1985) 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 called tuples are stored and retrieved. 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) (Cabri, Leonardi et al. 2000).

There has been some research about the integration of web services using the tuple space model (Bright and Quirchmayr 2004; Lucchi and Zavattaro 2004; Maamar, Benslimane et al. 2005; Bellur and Bondre 2006). 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 (Bessani, Alchieri et al. 2008; Bessani, Correia et al. 2009), 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) (Verissimo, Neves et al. 2003). Moreover, WSDS maintains all the dependability properties of a dependable tuple space (Bessani, Alchieri et al. 2008) *without further extensions to the web services core specifications.*

Why use WSDS to Coordinate Web Services?

Since or objective with WSDS is to provide a coordination infrastructure for web services, the attentive reader can ask: Why someone would use a service like WSDS to coordinate web services, instead of using some tool that implements web services coordination specifications? Here we list several reasons to use a service like WSDS to coordinate web services.

In usual web services coordination, the communication is often coupled in at least two levels: *(1)* cooperative services must be active and running at the same time in order to complete some interaction; and *(2)* the interactions are based on the services interfaces, if any interface change it is necessary reprogram these iterations.

These limitations can be overcome through the use of a coordination service based on tuple spaces (WSDS), since in this model any service (client of some other service) is allowed to send a request and terminate. After some time, this service can be reactivated to get the response of its request, which was produced while such service was disabled.

Other advantage of WSDS is related with services interfaces evolution. In this model, all interactions are mediated by a mediator (the tuple space - WSDS). Thus, services (and also clients of these services) do not need know their interfaces. This means that if some service interface changes it is not necessary to reprogram any interaction involving the service that had its interface modified. For example, services interfaces can change due to the evolution of these services, where new operations will become available to clients and other services.

Other point to highlight is that this type of coordination can be used as a mechanism for synchronization among multiple web services (e.g., to concurrency control). WSDS implements a coordination infrastructure that provides all of the advantages discussed here in a dependable way.

Summary of the Contributions

The main contributions of this paper can be summarized as follows:

1. The design and implementation of the WSDS infrastructure, which is the first dependable web services data-centered coordination service;

2. The evaluation of the cost to access this type of infrastructure through an analysis of the operations latency and its causes;

3. An analysis of some real applications that can be built over WSDS;

4. A study about WSDS relationship with the main standards for cooperating web services.

Paper Organization

This paper is organized as follows. First, the basic concepts about tuple spaces are discussed and a dependable tuple space, the DepSpace, is briefly reviewed. The WSDS, our proposal for a dependable infrastructure for coordinate web services, is then presented. An analysis, both analytical and experimental, about WSDS performance is then discussed. Some applications built over WSDS are then described. Interesting relations between the architecture proposed in this paper and some web services specifications are then presented. Finally, related work and conclusions are given.

TUPLE SPACES

A *tuple space* (Gelernter, 1985) 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 *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 *t' match* if they have the same number of fields and all defined field values of *t'* are equal to the corresponding field values of *t*. For example, template ‹*1, 2,**› 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(t')* is used to read tuples from the space, and returns any tuple of the space that *matches* the template *t'*. A tuple can be read *and* removed from the space using the *in(t')* operation. The *in* and *rd* operations are blocking. Non-blocking versions, *inp* and *rdp*, are also usually provided (Gelernter, 1985).

To increase the synchronization power of the tuple space, we consider also the *cas(t',t)* operation (conditional atomic swap) (Segall, 1995), that works like an indivisible execution of the code: **if** ¬ *rdp(t')* **then** *out(t)* (*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 (Segall, 1995; Bessani, Correia et al. 2009), which is a building block for solving many important distributed synchronization problems like atomic commit, total order multicast and leader election.

Table 1 presents a summary of the tuple space operations supported by WSDS.

Operation	Description
out(t)	Inserts the tuple *t* in the space.
rdp(t')	Reads a tuple that matches *t'* from the space (returning *true*). Returns *false* if no tuple is found.
inp(t')	Reads and removes a tuple that matches *t'* from the space (returning *true*). Returns *false* if no tuple is found.
rd(t')	Reads a tuple that matches *t'* from the space. Stays blocked until some matching tuple is found.
in(t')	Reads and removes a tuple that matches *t'* from the space. Stays blocked until some matching tuple is found.
cas(t',t)	If there is no tuple that matches *t'* on the space, inserts *t* and returns *true*. Otherwise returns *false*.

Table 1 - Operations supported by WSDS.

DEPSPACE: A DEPENDABLE TUPLE SPACE

A tuple space is dependable if it satisfies the *dependability attributes* (Avizienis, Laprie et al. 2004). 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;
- **Confidentiality**: the content of tuple fields can not be disclosed to unauthorized parties.

DepSpace (Bessani, Alchieri et al. 2008) is an implementation of a dependable tuple space that satisfies these properties using a combination of some mechanisms, which are described below.

The DepSpace architecture consists in a series of integrated layers that enforce each one of the dependability attributes listed above. Figure 1 presents the DepSpace architecture with all its layers.

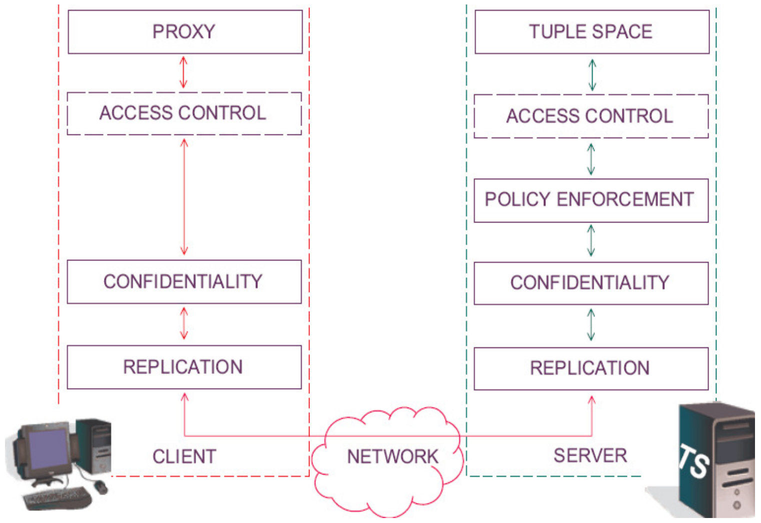


Figure 1 – DepSpace architecture.

On the top of the client-side stack is the proxy layer, which provides access to the replicated tuple space, while on the top of the server-side stack is the tuple space implementation (a local tuple space). The communication follows a scheme similar to remote procedure calls. The application interacts with the system by calling functions with the usual signatures of tuple spaces operations: *out(t)*, *rd(t')*, … These functions are called on the proxy. The layer below handles tuple level access control. After, there is a layer that takes care of confidentiality and then one that handles replication. The server-side is similar, except that there is a new layer to check the access policy for each operation requested.

Other important feature of DepSpace is its support for multiple logical tuple spaces: the system has administrative interfaces that allow (logical) tuple space creation and destruction. Two different logical tuple spaces have no relationship between themselves. We must remark that not all layers must be used in every logical tuple space configuration (only the replication layer is essential). The idea is that layers are added or removed according to the quality of service desired for the logical tuple space.

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 manage to overcome the misbehavior of the faulty ones. A simple approach for replication is *state machine replication* (SMR) (Schneider, 1990). SMR guarantees *linearizability* (Herlihy and Wing 1990), 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 (Castro and Liskov 2002).

Cryptography

DepSpace uses cryptographic processing to ensure confidentiality of tuples. The enforcement of confidentiality in a replicated tuple space is not trivial, since we can not trust a single server that can be compromised. Then, replication is often seen not as a helper but as an impediment for confidentiality. The reason is easy to understand: if secret information is stored not in one but in several servers, it probably becomes easier for an attacker to get it, not harder.

The solution for confidentiality adopted in DepSpace relies on a set of servers, where no server individually can have access to the tuples (Bessani, Alchieri et al. 2008). The basic tool it uses to implement confidentiality is a special kind of *secret sharing scheme* (Shamir, 1979).

To insert a confidential tuple in the space, the client encrypts the tuple with some secret *s* that it generates randomly. The client generates different secrets for each tuple that it inserts in the space. After this, client uses the secret sharing scheme to generate shares of *s*. Each server receives the encrypted tuple and one share of the secret *s* used to encrypt it. It is necessary a set of shares to obtain the secret *s* (to reconstruct *s*), the size of this set can be configured in the scheme (Shamir, 1979), usually this size is configured to $f+1$ in order to tolerate up to f failures of servers, i.e., it is necessary at least one correct server to disclosure the secret. Thus, a single server is not able to obtain *s* and decrypt the tuple.

Since DepSpace clients use total order multicast to insert tuples on the system, it is impossible to send different versions of a request to each server, containing only its secret share. Then, the client must encrypt each share with a secret key exchanged with the server that will hold this share. In this way, the request contains all encrypted shares, but each server is able to obtain only its share.

The confidentiality scheme has also to handle the problem of matching encrypted tuples with templates. When a client inserts a tuple in the space, it chooses one of three types of protection for each tuple field: *Public* - the field is not encrypted so it can be compared arbitrarily but its content may be disclosed if a server is faulty; *Comparable* - the field is encrypted but its cryptographic *hash* obtained with a *collision-resistant hash function* is also stored and can be compared; *Private* - the field is encrypted and no hash is stored so no comparisons are possible.

To access some tuple (read or remove operations), the client must wait for a set of responses from the servers. Each response contains the encrypted tuple and the share hold by the server that is replying (to avoid eavesdropping of responses, the share must be encrypted with the secret key exchanged by client and each server). Then, the client obtains the secret by combining a set of shares and decrypts the tuple. Further discussion about this mechanism, including all confidentiality protocols, is presented in (Bessani, Alchieri et al. 2008).

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 (Bessani, Correia et al. 2009). These 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(‹CLIENT,id,x›) can only be executed if there is no tuple on the space that matches (‹CLIENT,id,*›)"*. 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.

WS-DEPENDABLESPACE

This section describes the **WS-DependableSpace** (**WSDS**) infrastructure. First, our system model is presented. Then, the WSDS architecture and some details of the gateway operation are discussed. How our architecture ensures the security properties of the system despite the possibility of having malicious components is then described. Finally, some system implementation details are given.

System Model

The processes of the system are divided in three sets: n DepSpace servers $U = \{s_1,...,s_n\}$, g access gateways $G = \{g_1,...,g_g\}$ and an unknown set of clients $\Pi = \{c_1,c_2,...\}$. 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, that can be implemented using standard technologies such as SSL/TLS.

We assume an eventually synchronous system model (Dwork, Lynch et al. 1988): 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 + Δ. 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. This assumption is based on the fact that, even if the time required for some local operations is considerable, these computations are not susceptible to the external interferences, and therefore its asynchronous behavior is not observed in practice.

System processes are subject to Byzantine failures (Lamport, Shostak et al. 1982), 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 (Obelheiro, Bessani et al. 2006). WSDS works correctly while a bound of up $f_n \leq (n+1)/3$ DepSpace servers (the optimal resilience for this type of replication (Castro and Liskov 2002)), $f_g \leq g - 1$ *gateways* (at least one correct) and an arbitrary number of client fails. In the same way as in DepSpace, the secret sharing scheme used for confidentiality (Schoenmakers, 1999) requires at least $f_n + 1$ out of n shares of a secret to recover its secret.

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

WSDS Architecture

Figure 2 presents the WSDS architecture. The idea in this architecture is to have gateways connecting web service clients and the DepSpace servers. To do this, the gateways publish their WSDL interface in an UDDI repository. Thus, clients are able to access them.

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 DepSpace. The gateway receives SOAP messages sent by clients and transforms them in DepSpace requests.

The system works as follows: first of all, each gateway registers itself in one UDDI service. To improve the system availability, gateways can be registered in more than one UDDI service. Before accessing the tuple space, a client must find one or more gateway address on a UDDI service. 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. DepSpace servers process the request and send the response to the gateway, which collects $n - f_n$ 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 $f_n + 1$ servers (at least one correct server).

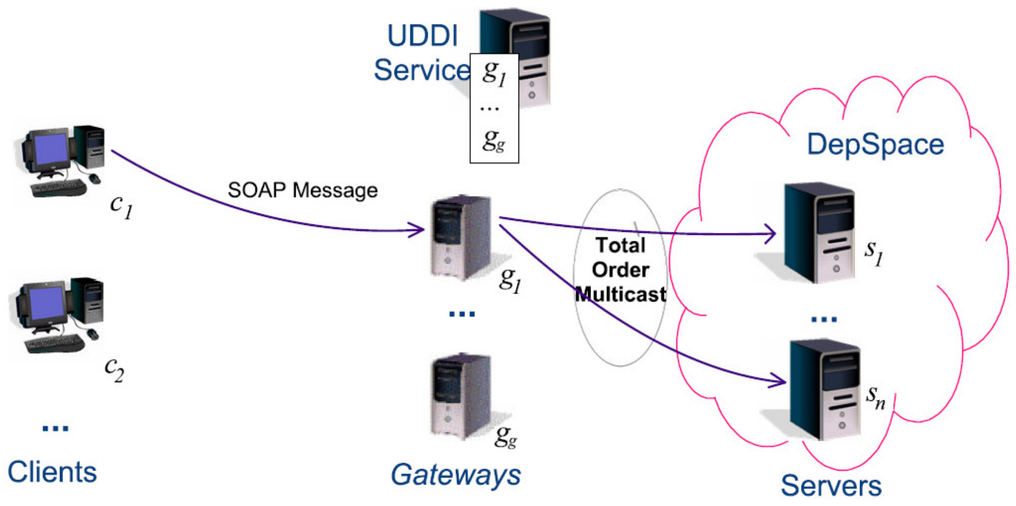


Figure 2 – WSDS architecture.

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 *g* gateways of the system.

Gateway Operation

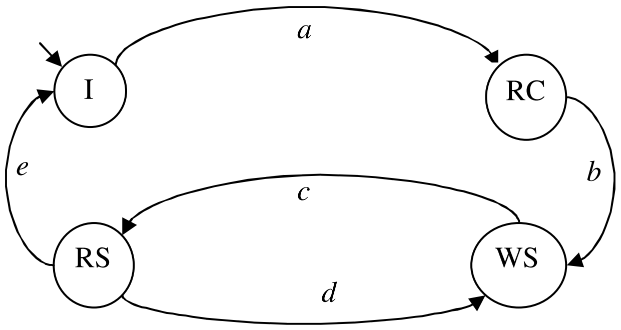
This section details the operation of the WSDS gateways, which provides the WSDS service to web service clients and act as clients of DepSpace.

Gateway State Machine

To better understand, Figure 3 presents the automaton that represents the processing at the gateway to execute some request. First, the gateway is idle and waiting some request (state 1). When some client sends a request to the gateway (transition *a*), it goes into state 2 where it has received the client request. Then, the gateway forwards the request to DepSpace servers (transition *b*) and goes to wait for replies (state 3). At this point, each correct DepSpace server executes the request and sends the reply to the gateway (transition *c*) that moves to state 4. In this stage, the gateway computes the reply received and takes one of two actions:

- If it has received enough replies (i.e., $n - f_n$): then it sends the collected replies to the client (transition *e*) and becomes idle again (goes to state 1).

- If more replies are necessary: then it comes back to state 3 (transition *d*) and waits for more replies before send them to the client.


$$\begin{tikzcd}
\arrow[r] & I \arrow[rr, bend left, "a"] & & RC \arrow[d, bend left, "b"] \\
& RS \arrow[u, bend left, "e"] \arrow[rr, bend right, "d"'] & & WS \arrow[ll, bend right, "c"']
\end{tikzcd}$$

States	Transitions
I: idle.	*a*: Client sends a request to the gateway.
RC: received client.	*b*: Gateway forwards the request to DepSpace servers.
WS: waiting server.	*c:* Some DepSpace server sends a reply to the gateway.
RS: received server.	*d*: Gateway come back to wait more replies.
	e: Gateway sends collected replies to the client.

Figure 3 – Request execution flow at the gateway.

This automaton represents the processing flow to execute some request. If the gateway receives a new request while it is processing some other request, it starts a new flow (in parallel) to attend this new request. All executions flows are independent.

Gateway to Client and Client to Gateway Messages

This section presents the structure of requests that are sent by clients to the gateways, as well as responses for these requests, that clients receive from gateways. The structures presented here correspond to operations executed in a logical tuple space where all layers are turned on. However, these messages present some variations depending of the layers used (e.g., if confidentiality layer is not active on DepSpace, no tuples are encrypted and no shares are necessary).

Figure 4 presents the structure of a request message that clients send to gateways. The structure is divided into two parts: context (CONTEXT) and contents (CONTENTS). The context contains information about the operation context, which comprises: *(1)* the request unique identifier (REQ_ID) that is formed by the concatenation of the client identifier (CLIENT_ID) and a timestamp (TIMESTAMP) that works like an incremental counter; *(2)* the operation identifier (OP_ID) that informs if the operation is an *out, rd, rdp, in, inp* or *cas*; *(3)* the logical tuple space name to be accessed (TS_NAME); and *(4)* the client signature (SIGNATURE) that ensures the authenticity of this request.

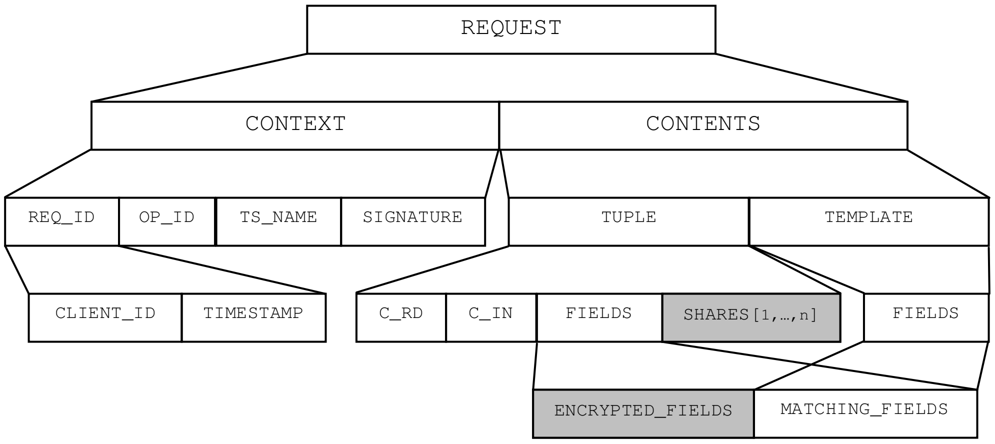


Figure 4 – Client to gateway request structure.

The contents side of requests contains the tuple (TUPLE) and/or template (TEMPLATE). Notice that, in write operations (*out*) this structure contains only the tuple, while that in read/remove operations (*rd, rdp, in, inp*) only the template is part of the message. However, in conditional atomic swap operations (*cas*) both, tuple and template are present in the message structure.

The tuple contains the following data: *(1)* credentials required to read (C_RD) ant to remove (C_IN) it from the space (to tuple access control); *(2)* tuple fields' data (FIELDS) that is composed by the encrypted fields (ENCRYPTED_FIELDS) and by the matching fields (MATCHING_FIELDS) that are constructed according with the protection chosen for each field and are used to compare tuples and templates; and *(3)* the set of *n* shares (SHARES[*1,…,n*]) of the secret used by the client to encrypt the fields of the tuple, each share is encrypted with the secret key exchanged by client and the server that will hold this share. The template contains only the fields used to compare it with tuples (MATCHING_FIELDS).

The confidential contents of requests are presented in gray color in Figure 4. Thus, as tuple fields are encrypted with some secret and also shares of this secret are encrypted, no gateway is able to access confidential contents of requests.

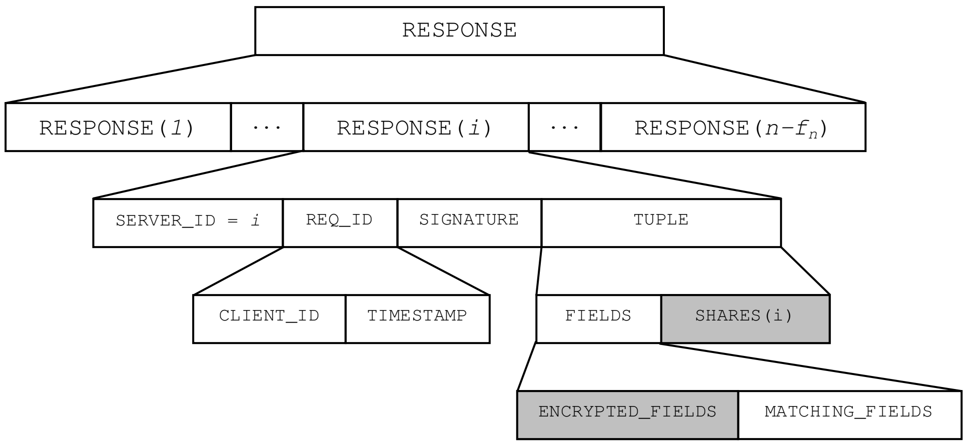


Figure 5 – Gateway to client response structure.

Figure 5 presents the structure of a response message that clients receive from some gateway. As already mentioned, this message contains a set of n - f_n responses of DepSpace servers. Each response is formed by: *(1)* the server identifier (SERVER_ID); *(2)* the request identifier (REQ_ID) that is the concatenation of the client identifier (CLIENT_ID) and a timestamp (TIMESTAMP), which works like an incremental counter; *(3)* the server signature (SIGNATURE) that ensures the authenticity of this response; and *(4)* the tuple read/removed from the space in read/remove operations (*rd, rdp, in, inp*), in write operations (*out*) this tuple contains only one special field indicating if the operation has successfully completed or not.

The tuple contains the following data: *(1)* tuple fields' data (FIELDS) that is composed by the encrypted fields (ENCRYPTED_FIELDS) and by the matching fields (MATCHING_FIELDS) that are constructed according whit the protection chosen for each field and are used to compare tuples and templates; and *(2)* the share (SHARES(i)) of the secret used by the client to encrypt the fields of the tuple that is held by this server, the share is encrypted with the secret key exchanged by this server and the client that is executing the operation.

As done with requests, the confidential contents of responses are presented in gray color in Figure 5. Thus, as tuple fields are encrypted with some secret and also the share of this secret held by the server that is replying are encrypted, no gateway is able to access confidential contents of responses.

Dealing with Faulty Gateways

Although the apparent simplicity of the architecture depicted in Figure 2, 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.

Authenticity and Integrity of Messages

The first problem to be solved in WSDS is that a faulty gateway can request the execution of invalid requests that were not issued by a valid client. Moreover, it would be able to corrupt requests, modifying some of their data (ex., parameters 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 non-corrupted requests issued by clients and valid responses produced by DepSpace servers are processed, avoiding all kinds of *man in the middle* behaviors that a faulty gateway can have.

Moreover, the client needs to send its credentials together with the requests, then the servers will be able to 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 ensure the authenticity and integrity of the end-to-end communications between WSDS clients and DepSpace servers, i.e., messages can not be modified by faulty gateways without being detected by DepSpace servers or WSDS clients. The certificates sent together with requests and responses are verified using some public key infrastructures such as X.509.

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. In fact, if the gateway accessed does not send the responses to the client, it will stay blocked indefinitely (waiting for the responses). 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 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).

Duplicated Requests

If some gateway is correct but very slow, the mechanism introduced in previous section can result in the same request being sent to more than one gateway. Consequently, it is possible that this 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 introduced in WSDS:

1. Each request has a 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 previously stored reply 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. This limitation can be relaxed for at most *k* requests if the servers store the last *k* replies to each client. 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.

Ensuring Confidentiality

Confidentiality is an important property of dependable systems. To ensure confidentiality, we must avoid that some components of the system (especially gateways and unauthorized clients) have access to the shares of the secrets used to encrypt tuples.

To insert a tuple in the space, client sends the tuple encrypted with a secret that is used to obtain a set of shares. The generated shares are also sent encrypted with secret keys exchanged by client and each server. In this way, each server will have the encrypted tuple and its share of the key needed to decrypt it. The gateways, by the other hand, can have access to the encrypted tuple and all encrypted shares, but they can not decrypt shares (and combine them in order to get the secret and decrypt also the tuple) without help from each server that will hold each share. Recall that at least $f_n + 1$ shares are required to rebuild the secret. Then, gateways only can access confidential tuples with help from at least one correct DepSpace server. Consequently, gateways will not be able to access any confidential tuple.

In read operations each server replies with both, the encrypted tuple and the share of the secret used to encrypt it. The share must be encrypted with the secret key established between the client and the server that is replying. Then, only the client is able to: *(1)* access the shares, *(2)* combine a set of them to get the secret, and *(3)* decrypt the tuple with this secret.

Implementation

The prototype was developed in Java and uses the DepSpace implementation described in (Bessani, Alchieri et al. 2008). 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. All cryptographic primitives were provided by the default provider of JCE (Java Cryptography Extensions) - version 1.5. The only exception was the secret sharing scheme used for confidentiality, which we implemented following the specification in (Schoenmakers, 1999), using algebraic groups of 192 bits (more than the 160 bits recommended). This implementation makes extensive use of the *BigInteger* class, provided by the Java API, which provides several utility methods for implementing cryptography.

The gateways were implemented using *Axis 1.3* (*http://ws.apache.org/axis/*), an open-source SOAP protocol implementation that provides a set of APIs for the development of client applications and web servers, and deployed on *Tomcat 5.5.20* J2EE container (*http://tomcat.apache.org/*) that is a robust and open-source application server.

All additional mechanisms required in our architecture (e.g., signature verifications and message identifiers) were integrated in DepSpace through *interceptors*. Interceptors allow change the behavior of the servers without alterations on their structure. The implementation of interceptors on DepSpace was inspired by CORBA portable interceptors (Object Management Group, 2002), and was integrated to the system architecture (Figure 1) 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, as we can see in Figure 6 that presents the gateways interface. In this interface, the abstraction *WSResponse* encapsulates the set of DepSpace server responses and is sent to the client as a reply to the request previously made by that client. The abstraction *WSDepTuple* represents tuples or templates, while the abstraction *WSMessageContext* represents the context of the operation, containing essential data for the execution of operations as already discussed.

```
public interface WSDepSpace extends java.rmi.Remote{

       WSReponse createSpace(Properties prop, WSMessageContext ctx);
       WSReponse deleteSpace(String name, WSMessageContext ctx);

       WSReponse out(WSDepTuple tuple, WSMessageContext ctx);
       WSReponse rd(WSDepTuple template, WSMessageContext ctx);
       WSReponse rdp(WSDepTuple template, WSMessageContext ctx);
       WSReponse in(WSDepTuple template, WSMessageContext ctx);
       WSReponse inp(WSDepTuple template, WSMessageContext ctx);
       WSReponse cas(WSDepTuple template, WSDepTuple tuple,
                                          WSMessageContext ctx);
}
```

Figure 6 – Gateway interface (without throws clauses).

The first two methods of gateways interface are to create and to delete “logical” spaces, respectively. In order to create a logical space, the client must supply information about the configuration desired to the space (e.g., what layers should be turned on). These informations are encapsulated in the `Properties` parameter of the `createSpace` operation. On the other hand, to delete a logical space, the client only should inform the space name (`deleteSpace` method).

The others methods represent each one of the operations defined to tuple spaces (Table 1). We must recall that the prototype implementation hides all aspects related to the access of these methods, i.e., the prototype contains administration interfaces that are used to create and delete spaces and accessor interfaces that are used to access the operations defined for tuple spaces.

EVALUATION

In this section we present both, an analytical analysis and an experimental analysis about the WSDS performance, considering the latency to access this service.

Analytical Evaluation

The latency of each operation is determined by the following equation:

$$L_{wsds} = L^{c}{}_{sign} + L^{c\rightarrow g}{}_{comm} + L^{g\rightarrow s}{}_{tom} + L^{s}{}_{ver} + L^{s}{}_{op} + L^{s}{}_{sign} + L^{s\rightarrow g}{}_{comm} + L^{g\rightarrow c}{}_{comm} + (f_n + 1)L^{c}{}_{ver}$$

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. In fact, the client needs verify f_n+1 replies from correct servers in order to get a response for some operation, then this equation represents the latency in failure-free scenarios. All equations presented here are for failure-free scenarios. When there are failures, more responses have to be verified and, also, more shares must be decrypted (see bellow).

The following two equations give the additional latency overheads on tuple insertions and reads (read or remove operations) when the confidentiality layer is enabled:

$$L(insert)_{conf} = L^{c}{}_{tencrypt} + L^{c}{}_{share} + n\,L^{c}{}_{sencrypt} + L^{s}{}_{sdecrypt}$$

$$L(read)_{conf} = L^{s}{}_{sencrypt} + (f_n + 1)L^{c}{}_{sdecrypt} + L^{c}{}_{combine} + L^{c}{}_{tdecrypt}$$

The additional sources of latency represented in these equation are: $L^{c}{}_{tencrypt}$ (resp. $L^{c}{}_{tdecrypt}$) - time spent on the client to encrypt (resp. decrypt) the tuple with some symmetric key; $L^{c}{}_{share}$ - latency for generating n shares from a given secret; $L^{c}{}_{sencrypt}$ (resp. $L^{c}{}_{sdecrypt}$) - time for the client to encrypt (resp. decrypt) the share using some symmetric key; $L^{c}{}_{combine}$ - time spent to combine f_n+1 correct shares and recover the secret. L^{s} components of the equation are exactly like their L^{c} counterparts, but the processing occurs at server side.

Latency with Confidentiality Disabled

Considering that clients and servers spend similar amount of time in cryptographic 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:

$$L_{wsds} = 2L_{sign} + L^{c\leftrightarrow g}{}_{comm} + L_{tom} + L^{s\rightarrow g}{}_{comm} + (f_n + 2)L_{ver}.$$

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.

Latency with Confidentiality Enabled

Using the same consideration of previous section and also assuming that the cost of symmetric encryption is similar to the cost of decryption – called L_{sym_crypto} - we can rewrite the confidentiality overhead equations as:

$$L(insert)_{conf} = L^{c}{}_{share} + (n + 2)L_{sym\_crypto}$$

$$L(read)_{conf} = (f_n + 3)L_{sym\_crypto} + L^{c}_{combine}$$

There are two interesting observations that we can make based on previous equations. First, most of the cryptographic work is performed on clients, which does not have impact on the service latency but improves the overall throughput of the system (servers can handle operations fast). Second, the scheme is heavily based on symmetric cryptography, which is usually considered fast. For more details about the cost of this kind of confidentiality scheme, see (Bessani, Alchieri et al. 2008).

Experimental Evaluation

In order to quantify this latency, some experiments were conducted in a local area network. The execution environment was composed by a set of 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, $n = 4$), one gateway ($g = 1$) 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 7 presents the latency observed in the execution of the three main operations defined to tuple spaces and supported by WSDS (varying the tuple size). We present results for the system configured with confidentiality layer activated and with this layer disabled. The figure shows that, even with large tuples, the decrease is reasonably small in the WSDS performance for both configurations, 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 - 6 ms*) is appropriate for this class of system (web services).

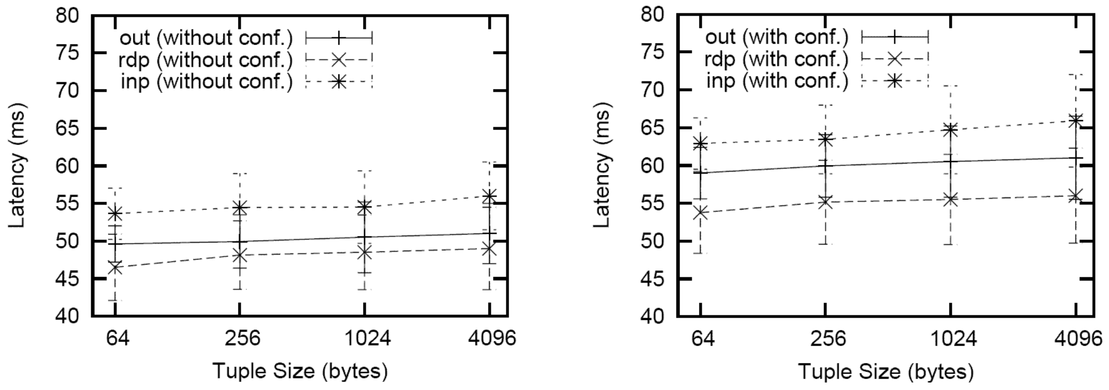


Figure 7 – WSDS latency with confidentiality layer disabled (left) and WSDS latency with confidentiality layer enabled (right).

Read operations (*rdp*) are faster due to an optimization incorporated in the read protocol of DepSpace, which execute these operations without use the agreement protocol (see (Bessani, Alchieri et al. 2008)). Also, Figure 7 shows that confidentiality impact only about *15%* in the system performance and that even for large tuples, the decrease in the performance is reasonable small. This happens due to one implementation feature: the secret shared in the secret sharing scheme is not the tuple, but a symmetric key used to encrypt it. Then, all cryptography required by this scheme can be executed in the same, which means that the tuple size has no effect in these computations and confidentiality implies almost the same overhead regardless of the tuple size.

To better understand the sources of latency, Table 2 presents the costs of each phase of the protocol to insert (*out*) and to remove (*inp*) a tuple of *64* bytes (recall that $n = 4$). The table shows also the percentage of the cost of each phase in the total latency observed by clients, in accordance with the latency equations presented. Cryptographic processing represents the bigger contribution (*~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 between client and gateway plus gateway and DepSpace servers). In these scenarios, cryptographic costs tend to dilute.

Operation	*out (without conf.)*		*out (with conf.)*		*inp (without conf.)*		*inp (with conf.)*	
Latency	*Cost (ms)*	$\%L_{wsds}$	*Cost (ms)*	$\%L_{wsds}$	*Cost (ms)*	$\%L_{wsds}$	*Cost (ms)*	$\%L_{wsds}$
$L^{s\rightarrow g}_{comm}$	00.63	01.27	00.65	01.10	01.31	02.44	02.25	03.58
$L^{c\leftrightarrow g}_{comm}$	13.53	27.27	16.23	27.50	17.39	32.42	19.95	31.71
L_{sign}	13.51	54.45	13.51	45.79	13.51	50.38	13.51	42.94
L_{ver}	00.85	05.13	00.85	04.32	00.85	04.75	00.85	04.05
L_{tom}	05.90	11.88	06.61	11.20	05.37	10.01	05.98	09.50
$L(insert)_{conf}$	---	---	05.95	10.09	---	---	---	---
$L(read)_{conf}$	---	---	---	---	---	---	05.17	08.22
L_{wsds}	49.63	100.00	59.01	100.00	53.64	100.00	62.92	100.00
L_{ds}	06.53	13.15	13.21	22.39	06.68	12.45	13.40	21.30

Table 2 - Latency costs for insert (*out*) and for remove (*inp*) a tuple of *64* bytes ($n = 4$).

APPLICATIONS

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

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 describing the service to be contracted (or the product to be bought) in the space. 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 8 illustrates this application and shows tuples that can be inserted in WSDS.

To properly work on an untrusted environment, this application needs to enforce the following prohibitions: *(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 containing rules like *"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 ‹PROVIDER,A› in the space"* (this rule must be complemented with another one which specifies that only an administrator will be allowed to insert this type of tuple), respectively. As already mentioned, WSDS also supports confidentiality of tuples. Thus, providers are able to insert proposals tuples in a way that no other provider can see its proposal even if some servers are compromised or messages are captured in the network (eavesdropping).

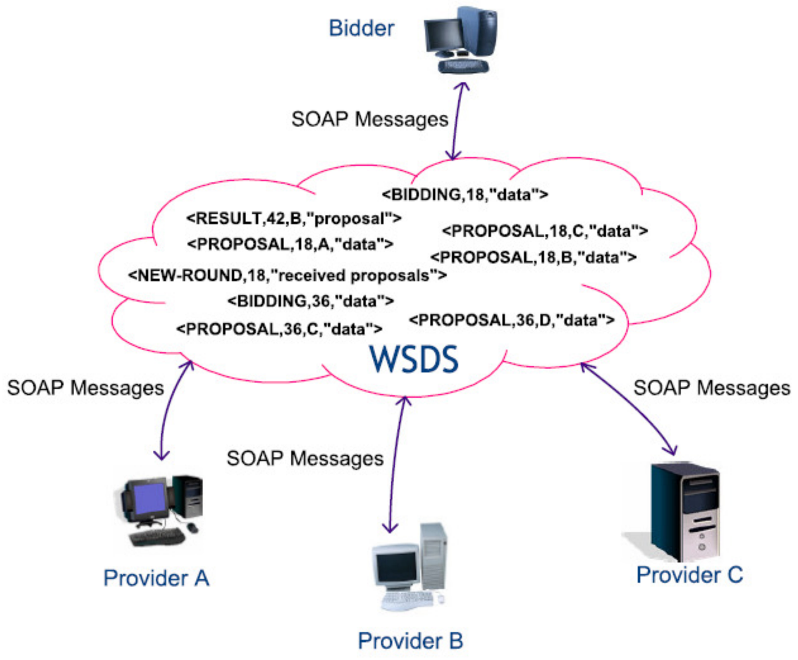


Figure 8 – Secure biddings.

Other examples of distributed applications in which a tuple space is used as a decoupled communication mechanism are: the I3 system (Stoica, Adkins et al. 2004) offers multicast, anycast and support to mobile communication on the Internet through an abstraction similar to a tuple space; the master-worker pattern, a classical parallel programming design pattern used to distribute tasks in cluster, can be easily implemented over a tuple space (Carriero and Gelernter 1989). If this space is deployed on the Internet, the same programming model can be used to distribute tasks in a computational grid (Favarim, Fraga et al. 2007). 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. Moreover, the decoupled coordination provided by the tuple space model allows distributed processes to interact without knowing their addresses and without be active at the same time (supporting temporary disconnections).

Data Sharing among Services

The applications that invoke several web services to execute complex tasks often 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.

Since web services are usually 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 become dependent of a single point of failure (the database). On the other hand, the session information exchange can demand that a high amount of extra data 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. 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).

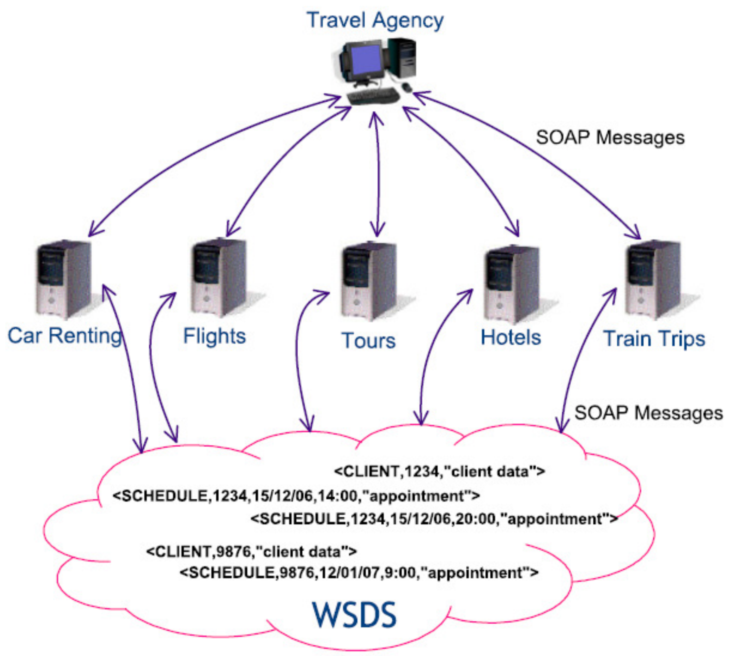

Travel Agency
SOAP Messages
Car Renting
Flights
Tours
Hotels
Train Trips
SOAP Messages
<CLIENT,1234,"client data">
<SCHEDULE,1234,15/12/06,14:00,"appointment">
<SCHEDULE,1234,15/12/06,20:00,"appointment">
<CLIENT,9876,"client data">
<SCHEDULE,9876,12/01/07,9:00,"appointment">
WSDS

Figure 9 – Travel agency.

A travel agency is an example of this type of system (Figure 9), 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).

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, and so on). Moreover, tuples with client data must be confidential, in order to maintain information privacy.

In this example, the agency can contract each service through the mechanism of secure biddings presented in the previous section. These applications can use the same tuple space or different logical spaces for each bidding and for the client schedule.

Other examples of this class of services are: integration of intrusion detection systems (Yegneswaran, Barford et al. 2004), 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 (Bright and Quirchmayr 2004; Ricci, Omicini et al. 2001), where a space is used by partners that compose the virtual enterprise to share information.

RELATIONSHIP WITH WS SPECIFICATIONS

There are many specifications that aim the integration of web services. This section presents possible relationships 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 (Cabrera, 2005). 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 was already 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 such as WSBPEL (*Web Services Business Process Execution Language*) (Alves, 2006). The travel agency presented in this paper 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 invoked 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 (Burdett and Kavantzas 2004). 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) (Peltz, 2003). 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.

RELATED WORK

Currently, there is a large effort in the development of standardized mechanisms that allow the cooperation and integration of web services. Some examples are WS-Coordination (Cabrera, 2005), WS-Orchestration (Alves, 2006) and WS-Choreography (Burdett and Kavantzas 2004). 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 (Bellur and Bondre 2006; Bright and Quirchmayr 2004; Lucchi and Zavattaro 2004; Maamar, Benslimane et al. 2005). These works highlight the advantages that a shared data repository with a well defined interface and some synchronization power can provide in web services coordination (Maamar, Benslimane et al. 2005) and in workflows execution (Bellur and Bondre 2006; Bright and Quirchmayr 2004).

An important aspect that is not addressed by most of these works is the dependability attributes required for this service (Avizienis, Laprie et al. 2004). 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 (Bellur and Bondre 2006; Bright and Quirchmayr 2004; Maamar, Benslimane et al. 2005). *WS-SecSpaces* (Lucchi and Zavattaro 2004) 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 coordination even in the presence of malicious processes (Bessani, Correia et al. 2009). 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* (Bessani, Alchieri et al. 2008). The resulting system tolerates accidental and malicious failures 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 (Minsky and Ungureanu 2000). 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.

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 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 reliability, integrity, confidentiality, and availability in a modular way.

The system was implemented using open source tools and a 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.

WSDS can be used to facilitate web services coordination in applications where this is required. For example, it can be used to implement secure interactions following the semantics defined by WS-Choreography specification (Burdett and Kavantzas 2004).

ACKNOWLEGMENT

Eduardo Adilio Pelinson Alchieri and Joni da Silva Fraga are supported by CNPq (Brazilian National Research Council). Alysson Neves Bessani is supported by FCT through Multi-annual and CMU-Portugal programs.

REFERENCES

Alves, A., et al. (2006). Web Services Business Process Execution Language, version 2.0, *OASIS Public Draft*. Retrieved November 16, 2006, from *http://docs.oasis-open.org/wsbpel/2.0/*.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, 1(1), 11-33.

Bellur, U., Bondre, S. (2006). xSpace: a Tuple Space for XML and its Application in Orchestration of Web Services, *Proceedings of the 21st ACM symposium on Applied computing - SAC'06*, Dijon, France, April 23-27, 766-772.

Bessani, A. N., Alchieri, E. A. P., Correia, M., Fraga, J. S. (2008). DepSpace: A Byzantine Fault-Tolerant Coordination Service, *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys'08,* Glasgow, Scotland, 31st March - 4th April, 163-176.

Bessani, A. N., Correia, M., Fraga, J. S., Lung, L. C. (2009). Sharing Memory between Byzantine Processes using Policy-enforced Tuple Spaces, *IEEE Transactions on Parallel and Distributed Systems,* 20(3), 419-432.

Bichier, M., Lin, K.-J. (2006). Service-Oriented Computing, *IEEE Computer*, 39(3), 99-101.

Bright, D., Quirchmayr, G. (2004). Supporting Web-Based Collaboration between Virtual Enterprise Partners, *Proceedings of the 15th International Workshop on Database and Expert Systems Applications*, Zaragoza, Spain, 30 August - 3 September, 1029-1035.

Burdett, D., Kavantzas, N. (2004). The WS-Choreography Model Overview, *W3C Working Draft.* Retrieved March 14, 2004, from *http://www.w3.org/TR/ws-chor-model/.*

Cabrera, L. F., et. al. (2005). Web Services Coordination Specification - version 1.0. Retrieved August 15, 2005, from *http://www-128.ibm.com/developerworks/library/specification/ws-tx/.*

Cabri, G., Leonardi, L., Zambonelli, F. (2000). Mobile Agents Coordination Models for Internet Applications, *IEEE Computer,* 33(2), 82-89.

Carriero, N., Gelernter, D. (1989). How to Write Parallel Programs: a Guide to the Perplexed, *ACM Computing Surveys*, 21(3), 323-357.

Castro, M., Liskov, B. (2002). Practical Byzantine Fault-Tolerance and Proactive Recovery, *ACM Transactions Computer Systems*, 20(4), 398-461.

Dwork, C., Lynch, N. A., Stockmeyer, L. (1988). Consensus in the Presence of Partial Synchrony, *Journal of the ACM (JACM),* 35(2), 288-322.

Favarim, F., Fraga, J. S., Lung, L. C., Correia, M. (2007). GridTS: A New Approach for Fault Tolerant Scheduling in Grid Computing, *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications - NCA'07*, IEEE Computer Press. Cambridge - MA, USA, July 12-14, 187-194.

Gelernter, D. (1985). Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, 7(1), 80-112.

Herlihy, M., Wing, J. M. (1990). Linearizability: A Correctness Condition for Concurrent Objects, *ACM Transactions on Programming Languages and Systems*, 12(3), 463-492.

Lamport, L., Shostak, R., Pease, M. (1982). The Byzantine Generals Problem, *ACM Transactions on Programming Languages and Systems*, 4(3), 382-401.

Lucchi, R., Zavattaro, G. (2004). WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications, *Proceedings of the 19th ACM Symposium on Applied Computing - SAC'04*, Nicosia, Cyprus, March 14 -17, 487-491.

Maamar, Z., Benslimane, D., Ghedira, C., Mahmoud, Q. H., Yahyaoui, H. (2005). Tuple spaces for self-coordination of web services, *Proceedings of the 20th ACM Symposium on Applied computing - SAC'05*, Santa Fe, New Mexico, March 13 -17, 1656-1660.

Minsky, N. H., Ungureanu, V. (2000). Law-Governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems, *ACM Transactions on Software Engineering and Methodology*, 9(3), 273-305.

Obelheiro, R. R., Bessani, A. N., Lung, L. C., Correia, M. (2006). How Practical are Intrusion-Tolerant Distributed Systems? *DI-FCUL TR 06-15, Dep. of Informatics, University of Lisbon*, September 2006.

Object Management Group (2002). The Common Object Request Broker Architecture: Core Specification v3.0, *Standart formal/02-12-06*, December, 2002.

Papadopolous, G., Arbab, F. (1998). Coordination Models and Languages, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, Academic Press.

Peltz, C. (2003). Web Services Orchestration and Choreography, *IEEE Computer*, 36(10), 46-52.

Ricci, A., Omicini, A., Denti, E. (2001). The TuCSoN Coordination Infrastructure for Virtual Enterprises, *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Cambridge MA, USA, June 20-22, 348-353.

Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting, *Proceedings of the 19th International Cryptology Conference on Advances in Cryptology – CRYPTO 1999*, Santa Barbara, California, USA, August 15-19, 148-164.

Schneider. F. B. Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial, *ACM Computing Surveys*, 22(4), 299-319.

Segall, E. J. (1995). Resilient Distributed Objects: Basic Results and Applications to Shared Spaces, *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing -- PDP'95*, San Remo, Italy, January 25-27, 320-327.

Shamir, A. (1979). How to Share a Secret, *Communications of ACM*, 22(11), 612-613.

Stoica, I., Adkins, D., Zhuang, S., Shenker, S., Surana, S. (2004). Internet Indirection Infrastructure, *IEEE/ACM Transactions on Networking*, 12(2), 205-218.

Verissimo, P., Neves, N. F., Correia, M. (2003). Intrusion-Tolerant Architectures: Concepts and Design, *Architecting Dependable Systems*, volume 2677 of *LNCS*, Springer-Verlag.

Yegneswaran, V., Barford, P., Jha, S. (2004). Global Intrusion Detection in the DOMINO Overlay System, *Proceedings of the 11th Network and Distributed Security Symposium - NDSS 2004*, San Diego, California, USA, February 4-6.

ABOUT THE AUTHOR

Eduardo Adilio Pelinson Alchieri received a Master degree on Electrical Engineering from the Federal University of Santa Catarina (UFSC), Brazil, in 2007, with Prof. Joni da Silva Fraga as the adviser. He is presently working toward his Doctorate degree at UFSC. Alchieri’s research interests span all areas of distributed computing, including both theory and practice. However, his current interests are reliable and secure algorithms to dynamic distributed computing systems.

Alysson Neves Bessani is a Visiting Assistant Professor of the Department of Informatics of the University of Lisboa Faculty of Sciences, Portugal, and a member of LASIGE research unit and the Navigators research team. He received his B.S. degree in Computer Science from Maring´a State University, Brazil in 2001, the MSE in Electrical Engineering from Santa Catarina Federal University (UFSC), Brazil in 2002 and the PhD in Electrical Engineering from the same university in 2006. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, middleware and systems architecture.

Joni da Silva Fraga is a Professor in the Department of Automation and Systems at Federal University of Santa Catarina, in Brazil. His research interests include distributed computing systems, algorithms, security and fault tolerance in distributed systems. He served as reviewer and as member of the PC in the main dependability related international conferences and as program chair. He is a member of the IEEE and the Brazilian Computer Society.