Executive Summary

This deliverable will identify and describe the concept of Triple Space Computing [Fensel D., 2004] (Triple Space) and find its
1. Introduction

The vision of Triple Space computing is introduced based on a combination of Semantic Web technology and space-based computing [Gelernter, 1992]. Triple Spaces provide a simple yet powerful communication paradigm that inherits the communication model for pervasive applications and projects in the context of Semantic Web services [Fensel D., 2004]. The name 'Triple Space' is derived from the use of RDF [Manola et al., 2004] triples, as explained later in this document.
The visionary approach of Triple Space Computing (TSC) was recently introduced based on the insight that Web Services do not follow the Web paradigm of 'persistently publish and read' [Fensel, 2004]. This approach is clear when Web pages are analyzed. Anybody can define a Web page persistently and publish it at an Internet service provider or at a server at home or at work (this is usually done with help of Web servers). Publication means that the Web page is made accessible by anybody who knows the URL of that Web page and has a Web browser to display it. Persistent publication means that the Web page does not get lost in case of power or network failures. Once the power is back on or the network link is re-established, the Web page is available again and unmodified compared to its content before the failure. Reading a Web page means to retrieve the Web page from the Web server by typing in the URL into a browser bar and requesting the display of it. Both, the publication and the reading can be done independently of each other, i.e., asynchronously. Fensel proposes to follow exactly this paradigm for the communication of data between software systems across the Internet: publish the data persistently and make it available for reading it. Triple Spaces, as introduced in this document, provide thus a persistent shared space for Web applications to prevent their interactions from relying on direct message exchange between them.

In the scope of this document the focus is in particular put on the use of Triple Spaces in the scope of the Web Service Execution Environment (WSMX). WSMX enables the dynamic discovery, selection, mediation, invocation, and inter-operation of Semantic Web services [Zaremba et al., 2004]. WSMX in its current state supports synchronous communication with other WSMXs and or systems. In addition, one WSMX may require the ability to communicate with several other WSMXs. Such communication will become costly when WSMXs are heavily loaded and their processing time is high. In such cases, Triple Space provides a medium for asynchronous communication minimizing the cost of interaction between WSMXs. One of the main benefits of asynchronous communication is that, unlike in synchronous communication, participants can come and go without hindering the communication process.

To conclude this introduction, we state in a summarized form the main benefits expected of Triple Space computing, especially in the scope of the WSMX development:

- **Web-style behaviour.** The Web paradigm is well-known and therefore easy to understand and use. This will allow setting up communication between communicating partners to be significantly more efficient.
- **Data semantics.** As the data is within the TSC infrastructure based on Semantic Web technology, the communication data is clearly semantically defined and hence machine understandable.
- **Asynchronous communication.** Asynchronous communication allows the data provider and reader autonomy. They can write and read according to their needs and independently from each other.
- **Reduction of uncertainty, delay and complexity.** The uncertainty of communication is reduced as the writer of the data can ensure that the data is persistently written if TSC does not return a fault message. The same applies for the reader. Interrupted network communication is no longer a problem of communication accuracy. Communication delay is eliminated since the reader can read whenever necessary and the data provider can write whenever necessary. There is no need to wait for each other in order to enable synchronous communication.
- **Minimal impact in provider and reader’s information systems.** There is no imposed impact to the writer and reader as accessing triple spaces through simple write and read interfaces was already introduced in [Gelernter, 1995].
1.1 Purpose of this Document

This document describes the possibility of using Triple Space computing in WSMX to enable distributed asynchronous communication between WSMXs and between components internally to a single WSMX. The document starts with the definition of Triple Spaces as derivation from Tuple Space computing and WSMX; finding the possible relationships between them. One of the main focuses of WSMX-Triple Spaces is the provision of a shared space for WSMX and the provision of a new means of asynchronous communication.

1.2 Document Overview

An overview of WSMX is presented in Section 2. As part of the background information, Section 3 introduces Tuple Space computing, and describes some major implementations of this concept. Triple Space computing is introduced in Section 4, while Section 5 describes the publish-subscribe paradigm based on Triple Space computing, i.e., how Triple Spaces could be enhanced through notification and advertisement services. Section 6 highlights the possible integration of Triple Space and WSMX architecture to enable inner-WSMX communication, while Section 7 takes a look at the possibility of asynchronous communication between WSMXs through Triple Spaces and finally. Section 8 eventually concludes the document and gives an insight on future work.

2. Web Service Execution Environment (WSMX) Overview

The Web Services Execution Environment (WSMX) [Zaremba et al., 2004] is an execution environment for dynamic discovery, selection, mediation and invocation of semantic web services. WSMX builds on the Web Services Modelling Ontology (WSMO) [Roman et al., 2005] that describes various aspects related to semantic web services.

WSMO is based on four concepts: web services, ontologies, goals and mediators. Web services are units of functionality; every Web service has exactly one capability, which describes logically what this Web service can offer. Every Web service has a number of interfaces, which specify how to communicate with it. Goals describe some state that a user may want to achieve. Ontologies are the formal specification of the knowledge domain used by both the Web service to express its capability, and by the goal to express the desired world state. Mediators are used to solve different interoperability problems, like differences in ontologies used by a Web service and a goal.

WSMX is developed as a reference implementation of an execution environment for web services. WSMX manages a repository of web services, ontologies and mediators. WSMX can achieve a user's goal by dynamically selecting a matching Web service, mediating the data that needs to be communicated to this service and invoking it.
**WSMX Architecture:**

In this section we use the term architecture to introduce the abstract software components that make up WSMX. The WSMX Manager and the Execution Engine co-ordinate the activities of WSMX following the execution semantics defined in [Oren, 2004]. All data handled inside WSMX is represented internally as an event with a type and state. The WSMX Core manages the processing of all events passing them to other components in the logic layer as appropriate. This Core could become redundant if the events are stored in the WSMX Triple Space once it is implemented.

The main purpose of Service Discovery is to provide functionality on matching of usable SWS with the goals. Selection of the Web service might happen in WSMX and for that purpose a selection component is used. The Mediator component provides a means of transforming data based on concepts in one ontology to data based on concepts in another ontology. The mapping is based on rules defined between concepts in the source and target ontologies. In the event that data mediation is required and the mediated data is in a non-XML format, the XML Converter can be invoked to translate the results of the Mediator into XML. This is necessary as the Web service invocations are via SOAP and the message format for SOAP messages is XML. The Compiler component parses WSML messages received from the WSMO Editor in the User Interface layer, validates the messages against WSMO and then stores the message elements in WSMX repository. The elements compiled to WSMX are the metadata for web services, ontologies, mediators. Once any of these elements have been compiled to WSMX, they are available for use during execution of goals sent to WSMX. The Message Parser parses the WSML Messages containing goals sent to WSMX. The goal is parsed and stored persistently. The functionality of the Message Parser is similar to that of the compiler but there is a conceptual difference. The Message Parser operates on instances of goals while the Compiler operates on the metadata for web services, ontologies, and mediators. Adapters allow applications which can not directly communicate with the interfaces provided by WSMX to communicate with WSMX. The role of the Choreography Engine is to mediate between the requester's and the provider's communication patterns. Reasoner will provide reasoning services for the mapping process of mediation, as well as functionality relating to validation of a possible composition of services, or determination if a composed service in a process is executable in a given context. The Resource Manager is responsible for management of Repositories. These Repositories are used to store the definitions of goals, web services, ontologies and mediators within WSMX. The repositories can be either internal to WSMX or external as, for example, the API to the UDDI described in the WSMO Registry [Herzog et al., 2004]. The figure below shows the WSMX architecture and the position of each component with it.
Messages and Workflow:

WSMX is actually based on the concept model of an event architecture. Components subscribe to a core component, WSMX Manager. This manager generates events for the subscribed components. The listener of each of these components can create, update and consume events. The interaction between components is done by events that contain messages. These events are java objects with an internal data structure and an interface specification. The execution semantics of WSMX as described in [Oren et al., 2004] is implemented in this component. Consequently the WSMX Manager is responsible for coordinating the overall operation of a WSMX system.

The event listener components are capable of accepting and processing events picked up by the events scanner from the event repository (in other words: the events scanner queries the event repository looking for "unlocked" events and if it finds any, the broadcasting mechanism of the WSMX manager core distributes this information to each listener of each WSMX component). The event repository is the persistent mechanism where all the events that are processing by the system are stored.

A design based in Service Oriented Architecture and a workflow engine that will allow the definition of multiple execution
semantics are the main goal for future revisions of WSMX architecture. This improved architecture will allow for easy to plug and unplug components and to integrate components from different vendors/developers.

3. Tuple Space Computing

A Tuple Space is a globally shared, associatively addressed memory space that is organized as a bag of tuples [Gelernter, 1985]. Gelernter and colleagues proposed a coordination language called Linda that introduces a small number of Tuple Space communication primitives. These primitives can be easily embedded in standard sequential computation language (such as C, Prolog or FORTRAN) and produce a complete parallel programming language (e.g., C-Linda, Shared Prolog or FORTRAN-Linda).

The basic element of a Tuple Space system is a tuple, which is a sequence of typed value fields (called actual fields), or non-valued typed fields (called formal fields). Note that the in [Gelernter, 1985] the first field is defined to contain a actual string value as identifier of the tuple. Thus, a possible tuple for Linda might look like: <tuplename, ex:integer, FALSE>, where tuplename is the tuple's identifier, ex:integer a formal field and FALSE an actual field of type boolean.

Besides the tuples communicated over the space there exist so-called templates in Linda. A template is similar to a tuple and are used to search a tuple by pattern matching in the Tuple Space. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field in type or value. The template <tuplename,2,ex:boolean>> matches for example the previously introduced tuple <tuplename, ex:integer, FALSE>.

The tuples are created by processes and placed in the Tuple Space by use a "write" primitive (called "out" in Linda). Processes read or remove tuples with read ("rd" in Linda) and take ("in" in Linda) primitives, which take a template and return the first matching tuple. The spaces are defined to be unstructured and thus only one tuple is returned - which tuple that is, is determined by the concrete implementation of the Tuple Space. Most Tuple Space implementations extend Linda to provide both blocking and nonblocking versions of the primitives for reading and removing tuples. A blocking read, for example, waits until a matching tuple is found in the Tuple Space, whereas a nonblocking version will return a "tuple not found" value if no matching tuple is immediately available.

Existing publish/subscribe technologies like above introduced Tuple Space computing [Gelernter, 1992], but also Blackboard systems [Penny Nii, 1989], Shared Object Space or Persistent Message-based Architecture [Alonso et al., 1995] are at the basis of Triple Space computing. To be able to understand and to define Triple Space, we first take a look at existing space-based computing implementations.

3.1 Linda

Linda was developed as a parallel programming languages in the mid-80's [Gelernter, 1985] at Yale University. It provides a communication paradigm between parallel processes called "generative communication model" which differs from other
communication models (i.e., monitors, message passing and message queueing) developed in distributed computing. In this paradigm the results of a computer's process or the processes themselves are added as messages in tuple-structured form to the computation environment, where they can be accessed as named, independent entities until some process chooses to receive them. The *Tuple Space* for process creation and co-ordination is a form of shared "associative memory" for tuples where a tuple is an ordered sequence of typed or valued fields.

Tuples are distinguished by tags (i.e., symbolic names) and by the number and types and values of their fields. There are two types of tuples: active tuples and passive tuples. Active tuples are basically processes or task-descriptions used for process creation. They contain functions and elements (i.e., formal parameters and actual parameters) that need to be evaluated by the Linda server in order to be placed in the Tuple Space; they require computation. Passive tuples, also called data tuples, are results of the computation, data values (i.e., actual parameters) that are stored in the space and which are used for process co-ordination. By evaluation an active tuple becomes a passive tuple. During this process all entries in the active tuple are evaluated in order where each entry in the tuple is an expression. Once the entries have been evaluated, the "passive" result replaces the original expression in the tuple. When a process is complete, its tuple becomes passive data (i.e., actual values) stored in the Tuple Space. Therefore active tuples can contain a combination of formal and actual values while passive tuples can only contain actual values. A short overview of the basic Linda primitives can be found in Appendix A.

### 3.2 TSpaces

TSpaces is a Java-based intermediary built upon the Linda coordination model with added middleware extensions developed by IBM at the Almaden Research Centre [Wyckoff, 1998]. It is a network communication buffer with database capabilities which enables communication between applications and devices in a network of heterogeneous computers and operating systems. TSpaces provides group communication services, event notification services, URL-based file transfer services, and database services, such as transactions, persistent data and flexible queries. Furthermore it provides XML support and is lightweight enough to be used to bring network services to small (palm-top computers) and embedded systems [Lehman et al., 2001].

The client-server (spaces) communication is established by a separate socket connection per client. This socket is then shared for all interactions between that client and any number of spaces on the server (A TSpace server may contain multiple Tuple Spaces). The TSpace implementation support moreover multiple active transactions simultaneously. All it needs are multiple instances of the Tuple Space all pointing to the same space. A short overview of basic TSpaces operations is given in Appendix A.

### 3.3 JavaSpaces

JavaSpaces [Freeman et al., 1999] technology by Sun Microsystems is a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources like clients and servers. The use of Java allows heterogeneous clients and servers to interoperate regardless of their processor architectures and operating systems. JavaSpaces adds transactional semantics, tuples leasing and event notification. In JavaSpaces tuples are called *entries* and are
represented as serialized Java technology-based objects. This allows participants in a distributed solution to exchange tasks, requests, and information as objects. A process that reads or takes an entry can invoke the operations that are associated with the entry. A Client operates on a JavaSpace to write new entries, look up existing entries, and remove entries from the space. This allows storage of 'states' by Java programs. In particular when security restrictions for applets do not allow the storage of state information on a client machine, then the state may be stored on the server. A short overview of basic JavaSpaces operations can be found in Appendix A.

3.4 Other Frameworks

**XMLSpaces by Technische Universität Berlin** [Tolksdorf and Glaubitz, 2001]. XMLSpaces extends the Linda coordination language for Web-based applications. It supports XML documents as tuple fields and multiple matching routines implementing different relations amongst XML documents, including those given by XML query-languages. XMLSpaces is distributed with an encapsulated open distribution strategy.

**Ruple by Rogue Wave Software** [Thompson, 2003]. Ruple is an Internet shared memory space that stores XML documents rather than Java objects and that is accessible over the Internet using standard Internet protocols such as HTTP and SOAP. Applications can place documents in a Ruple Space and then retrieve them using an XML query expression. A space can be located anywhere on the Internet, for example at "space.roguewave.com" or "spacely.sprockets.com", so spaces are easy to find using standard Internet protocols such as DNS. A space is also accessible as a Web service.

**Semantic Web Spaces by Free University of Berlin** [Tolksdorf et al., 2004]. Semantic Web Spaces are an extension of XMLSpaces shortly presented above. Semantic Web Spaces remodel the fundamental XMLSpace ideas to provide a suitable middleware for the Semantic Web to support technologies like RDF(S) and OWL.

**CORSO - COoRdinated Shared Objects by Vienna University of Technology.** CORSO provides a shared memory for heterogeneous distributed systems that can be seen as a middleware – between the distributed application and the operating system – that provides a solution for the problems of distributed application: replication, migration of data, information access, transactions and failure tolerance. The software abstraction CORSO offers is reached by alleviating the developer with the problems just described and by merging the local memory of each client with all the others in the space.

4. Triple-Space Computing

The visionary approach of Triple Space Computing (TSC) was recently introduced based on the insight that Web Services do not follow the Web paradigm of 'persistently publish and read' [Fensel, 2004]. This approach is clear when Web pages are analyzed. Anybody can define a Web page persistently and publish it at an Internet service provider or at a server at home or at work (this is usually done with help of Web servers). Publication means that the Web page is made accessible by anybody who knows the URL of that Web page and has a Web browser to display it. Persistent publication means that the Web page does not get lost in case of power or network failures. Once the power is back on or the network link is re-established, the Web page is available...
again and unmodified compared to its content before the failure. Reading a Web page means to retrieve the Web page from the Web server by typing in the URL into a browser bar and requesting the display of it. Both, the publication and the reading can be done independently of each other, i.e., asynchronously. [Fensel, 2004] proposes to follow exactly this paradigm for the communication of data between software systems across the Internet: publish the data persistently and make it available for reading it. The location for storing and accessing data is called triple space, and, as it will be later introduced, it is a virtual storage location. Triple Space Computing

Instead of following the 'persistently publish and read' paradigm, many Web Services based on the Web Service Definition Language (WSDL, [Christensen et al., 2001]) and the Simple Access Object Protocol (SOAP, [Mitra, 2003]) require a synchronous Hypertext-Transfer-Protocol (HTTP, [Fielding et al., 1999]) connection to transmit data transparently bypassing the power of the Web paradigm. This means that many Web Services require the sender and receiver of data to have a tight same-time synchronous connection, to agree on the data format, to know each other and share a common representation. There are of course several attempts to provide asynchronous communication to Web Services like substituting HTTP by Simple Mail Transfer Protocol (SMTP, [Postel, 1982]), WS-Notification [Graham and Niblett, 2004] and WS-Eventing [Geller, 2004], but we will see in this document that Triple Space Computing can improve a better scenario to achieve persistent asynchronous communication for Web Services.

Triple Space Computing establishes the mechanism to publish communication data according to the Web paradigm of 'persistently publish and read' and in this way TSC brings machine-to-machine Web Service communication to the Web in its real sense: 'Web' Services. Moreover, Triple Space Computing, follows the same goals for the Semantic Web services as the Web for humans: re-define and expand current communication paradigm (Cf. Figure 4.1). Triple Space is the necessary communication infrastructure where Semantic Web and Semantic Web services will become true. As [Fensel, 2004] pointed out: "Triple Space may become the web for machines as the web based on HTML became the Web for humans".

![Diagram](http://wsmo.org/TR/d21/v0.1/ (10 of 40))
Triple Space is based on the evolution and integration of several well-known technologies such as Tuple Space Computing [Gelernter, 1992], Shared Object Space (http://www.tecco.at/en/eProducts.html), Persistent Message-based Architecture [Alonso et. al., 1995], and RDF [Klyne and Carroll, 2004].

This has several benefits. The provider of data can publish it at any point in time (time autonomy), independent of it internal storage (location autonomy), independent of the knowledge about potential readers (reference autonomy):

- Processes communicating with each other do not need to know explicitly from each other. They exchange information by writing and reading tuples from the tuples pace; they do not need to set up an explicit connection, i.e., reference-wise the processes are completely de-coupled.
- Processes that interact through Triple Space do not need to be up at the same time, i.e. time-wise the processes are partially de-coupled.
- The processes can run in completely different computational environments as long as both can make access to the same tuple space, i.e., space-wise the processes are completely de-coupled.

This decoupling has obvious design advantages for defining reusable, distributed, heterogeneous, and quickly changing applications as promised by Web services technology. Also, complex APIs of current Web services technology will boil down to a read and write operation in a tuple space. It is worth to note that a service paradigm based on the tuple space paradigm also revisits the web paradigm; information is persistently written on a global place where other processes can smoothly access it without starting a cascade of message exchanges.

One of the important goals is to bring TSC to an Internet-scale level. This means that any number of data providers and readers can write and read communication data across the whole Internet. Like in the case of Web pages, there should be no limitation in principle: the architecture is independent of system properties like response-time and throughput. The limitations that exist come only through the technology used. In order to become Internet-scalable, TSC is based on proven technology and their combination. We will study the applicability of HTTP protocol and the Resource Description Framework (RDF, [Klyne and Carroll, 2004]) technology and commercially available storage components like databases or file systems. In that regard, exactly the same technology is used as in case of (Semantic) Web pages.

4.1 Requirements

Stating requirements is very important for the design of systems, be it their architecture or their implementation, in order to have guidance when choices have to be made. In the following section, the basic relevant requirements for a TSC architecture are discussed:

- Autonomy. As discussed earlier, there are four basic forms of autonomy that have to be realized by the architecture and its
implementation: time, location, reference and data schema autonomy. Since those have been discussed already, they are not again discussed here.

- **Viral technology.** In order for technology to be picked up to be useful on a wide basis it has to be simple and useful at the same time. Only then maximal impact can be achieved on an Internet-scale.

- **Efficiency.** The architecture must be efficient, meaning few components and few essential interactions between them. The implementation, if efficient, is cost effective and easy to maintain. Both achieve also Internet-wide high-performance and high-scalability.

- **Scalability.** A standard requirement for any information dissemination system is scalability, which is the ability to support a large number of clients (as many as the Web has).

- **Decentralized Architecture.** Like the Web, our proposal cannot be operated centrally. Decentralization is particularly necessary in order to avoid single-point-of-failures or performance bottlenecks in the system.

- **Security and Trust mechanisms.** The protection of private and restricted information published in Triple Spaces and the inclusion of trusted mechanisms to guarantee the validity, or trustworthiness of the information accessed are critical requirements for a successful development of a distributed information infrastructure.

- **Persistent communication.** Persistency based on write and read operations in a persistent storage infrastructure can increase reliability of the communication between participants, facilitate re-starting the execution of an aborted interaction process, and track communication between participants in the space.

- **History.** A triple space implementation can provide a history functionality that ensures that every 'movement' of triples, be it writing them, reading them, deleting them or updating them is recorded as history. At any point in time the history can make a precise statement about the ongoing operations.

### 4.2 Architecture Concepts

This section introduces the main elements of the TSC Architecture. We identify the concept of Triple Space, we define three different roles that participants can play in a Triple Space infrastructure, and we use an API to allow processes to interact with the space. This APIs are strongly influenced by the work done in TSpaces.

**Triple Space.** A triple space is a virtual space. A triple space is identified through a unique URL. Triples are written and read with this URL as triple storage location. Thus, RDF triples representation requires the specification of context information (in this case, in which triple space was created each RDF triple).

A triple space as such is not associated in a particular way with two possible implementations called triple-space-servers and triple-space-heavy-clients. An implementation, of course, has to provide a physical storage location like a database or file system directory or directory itself. However, one implementation can implement a 'host' of many triple spaces. On the other hand, a triple space can be built using several implementations that altogether provide a virtual unique space. The relationship is many-to-many between a triple space server/heavy-clients and (virtual) triple spaces. Moreover, a virtual triple space can be composed by several sub-spaces that basically restrict access to a set of RDF triples.

**Triple-Space-Server.** A triple space server is an individual persistent infrastructure that can partially host many triple spaces.
Consequently, management operations have to be available on a triple space server to create, delete and empty triple spaces:

- `create_triple_space (URL)` returns success | `triple_space_already_exists`
- `delete_triple_space (URL)` returns success | `triple_space_does_not_exist`
- `empty_triple_space (URL)` returns success | `triple_space_does_not_exist`

TSC clients do not know about triple space servers but only the virtual triple spaces.

**Triple-Space-Lightweight-Client.** A triple-space-lightweight-client writes triples, reads triples or does both either at the same time or sequentially. Lightweight clients are therefore not distinguishable from the viewpoint of a triple space. Every lightweight client in general can read and write triples. The lightweight-client API defines the basic operations that a client can execute in the Triple Space. These operations are based on TSpaces API. Table 4.1 provides a brief description of the API.

<table>
<thead>
<tr>
<th>Triple Space API for light-weight-clients</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void write (Set t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Write one or more triples in a concrete Triple Space identified by a unique identifier. A triple or set of triples that is written is not visible outside its transaction until the transaction successfully commits.</td>
</tr>
<tr>
<td><strong>Triple take (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Return a triple (or nothing) that match with the template and delete the matched triple from a concrete Triple Space. A read may match any entry written under that transaction or in the entire concrete space identified by a unique id.</td>
</tr>
<tr>
<td><strong>Triple waitToTake (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <code>take</code> but the process is blocked until the a triple is retrieved.</td>
</tr>
<tr>
<td><strong>Triple read (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <code>take</code> but the triple is not removed.</td>
</tr>
<tr>
<td><strong>Triple waitToRead (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <code>read</code> but the process is blocked until the a triple is retrieved.</td>
</tr>
<tr>
<td><strong>Set scan (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <code>read</code> but returns all tuples that match with 't'</td>
</tr>
</tbody>
</table>
Table 4.1 Triple Space API for lightweight-clients

**Triple Space Heavy Clients.** Heavy clients provide storage, query and versioning support to let users to work off-line with their own virtual spaces. Replication mechanisms keep replicas in heavy clients and servers up-to date. The API for heavy clients combined Server and Client APIs.

5. Publish-Subscribe Model

As a relevant part of the *Service Oriented Architecture* (SOA), *notification* is expected to play an essential role in the development of asynchronous, loosely-coupled and dynamic systems, where entities receive messages based on their registered interest in certain occurrences or situations. There is a long tradition in the use of these kinds of technologies in the areas of distributed objects, message oriented middleware (MOM) and Peer to Peer systems. Recently, two new specifications, *WS-Notification* [Graham and Niblett, 2004] and *WS-Eventing* [Geller, 2004], bring the publish-subscribe communication paradigm once again to the fore.

The publish-subscribe paradigm is an asynchronous, many-to-many communication for distributed systems [Eugster et al., 2003]. The model defines two main roles for participants: **source**, which generates notifications; and **sink**, which expresses its interest in concrete event notifications or pattern of event notifications. Typically a source can act as a producer and as a publisher. **Producers** encode information into notification messages, while **publishers** make accessible these notifications. Similarly, a sink can act as a subscriber and as a consumer. **Consumers** express interest in concrete notifications and consume those notifications when are published by a source. **Subscribers** are responsible for registering the consumer’ interests.

In the case of a loosely-coupled configuration (independent producers, publishers, subscribers and consumers), the publish-subscribe model de-couples the processes involved in information exchange in four orthogonal dimensions (partially adapted from [Eugster et al., 2003]):

- **Space decoupling:** Producers and consumers can run in completely different computational environments as long as both can make access to the same event service, i.e., space-wise the processes are completely de-coupled
- **Reference decoupling:** the processes that interact through an event service do not need to know each other (anonymous). The notifications published by publishers are accessed by consumers indirectly. In general, notifications do not include references to concrete consumers, and similarly consumers do usually not include specific references to producers.
- **Time decoupling:** the processes that interact through an event service do not need to be up at the same time (asynchronous). In particular, producers might generate some notifications while related consumers are not connected with the event service, and the other way around, consumers might get notifications while the original producers are not online.
- **Flow decoupling**: participants are not block while producing/receiving notifications. Consumers can receive a notification while performing some concurrent activity (i.e. through a 'callback'). Producers can produce notifications and continuous with their execution flow. In other words, main flows of producers and consumers are not affected for the generation or reception of notifications.

![Figure 5.1: Space, reference, time and flow decoupling (To be added)](http://wsmo.org/TR/d21/v0.1/)

Publish-subscribe mechanisms can be classified in two major categories: topic-based or content-based. **Topic-based** publish-subscribe system is the earlier variant of the publish-subscribe model in which each notification is published with respect to one of a fixed set of topics, also referred as groups, channels, or subjects. A publisher labels each message that producers generate with a particular topic. Similarly, a subscriber targets consumer's subscriptions with a particular topic. Thus, consumers get all messages associated with that topic. In a **content-based** system, producers use a predefined message schema to create messages, and consumers submits the subscription as a query (or filter, or pattern) against the message schemas. One advantage of a content-based system is that the consumers have the flexibility to choose multiple filtering criteria instead of being limited to pre-defined topics. However, message schemas in content-based system are under the trade-off between scalability and expressiveness. The more expressive message schema, the more difficult it is to evaluate it.

**WS-Notification** [Graham and Niblett, 2004] is part of the Web service Resource Framework (WSRF) [Globus et. al., 2004], a new proposal to extend the dominant Open Grid Service Infrastructure (OGSI) ([Foster et. al., 2002], [Tuecke et. al., 2003]) by integrating Web Services technologies. The WS-Notification specification is composed of a set of specifications: **WS-BaseNotification** [Graham and Niblett, 2004a], **WS-BrokeredNotification** [Graham and Niblett, 2004b] and **WS-Topics** [Graham and Niblett, 2004c]. **WS-BaseNotification** defines the Web services interfaces and standardizes message exchanges for Notification-Producers and Notification-Consumers. **WS-Brokered Notification** specification defines the Web services interface for the Notification-Broker. A Notification-Broker is an intermediary mechanism between Notification-Producers and Notification-Consumers. It is responsible to distribute notifications produced by Notification-Producers to interested Notification-Consumers based on their subscription specifications. **WS-Topics** defines a mechanism to organize and categorize subscriptions and notifications based on a hierarchical set of topics. WS-Notification also currently uses two related specifications from the WSRF specification: **WS-ResourceProperties** [Graham, 2003] to describe data associated with resources, and **WS-ResourceLifetime** [Frey Graham, 2004] to manage lifetimes associated with subscriptions and publisher registrations (in WS-BrokeredNotifications).

On the other hand, **WS-Eventing** [Geller, 2004] can be considered as a subset of the WS-Notification specification, and more precisely, roughly equivalent to WS-BaseNotification. Differences arise between both specifications: the complexity of the specifications, message definitions, delivery modes, subscription operations, Topic Space management and publishing. A detailed analysis of both proposals can be founded in [Pallickara and Fox, 2004].

A fundamental problem of these two specifications, and in general of most publish/subscribe systems, is how match the interests of consumers with the available notifications generated by producers. Simple strings such as “Weather/Warnings” for complex XPath or SQL queries do not provide enough expressivity to perform a sophisticated matching of interests and data. However, in
Li and Jiang, 2004, a proposal for a Semantic Message Oriented Middleware based in DAML+OIL is presented to overcome this limitation. We agreed with Li and Jiang that a formal knowledge representation language for expressing sophisticated classifications and for executing automated inference can be the way to go in future implementations of publish-subscribe systems.

5.1 Extending Triple Space paradigm

Triple Space paradigm has the same limitation than tuple space for readers (consumers) processes. An application which wants to read a concrete triple or set of triples has to interrupt the main process flow or run a concurrent thread that periodically checks if the data is available. JavaSpaces and TSpaces provide a simple notification mechanism to mitigate the problem. Thus publish-subscribe model can complement Tuple-Space with a sophisticated notification and subscription mechanism that allow a proper asynchronous interaction from the consumers/reader side. Figure 5.2. shows an example of a simple producer-consumer interaction between two processes. Process B is the consumer and search for the data before de data is available in the space. Process A publishes the data in the space. In the left side, process B queries the space (and blocks main flow) until data is available. In the right side, process B is subscribed to the data, and when the data is available the publish-subscribe mechanism send a notification to process B indicating that the data is available, and can be collected.

Figure 5.2: Example of interaction in Triple Space and Publish-Subscribe models
On the other hand, Triple Space paradigm provide a more direct way of communication for producers and consumers than publish-subscribe. When a consumer requires some concrete data just a read operation is needed. The same happen with producers (only a write operation is needed to publish information in the space). Furthermore Triple Space can improve publish-subscribe model by providing a shared persistent space in which store and replicate subscriptions and notifications. Using RDF data model, the space can be more structured and the data can be better described. Scalability issues in publish-subscribe systems motivated by large amount of data embedded in notifications and event storms produced by numerous notifications can be mitigated by an effective use of the shared space paradigm.

Table 5.1 shows a first proposal of a coordination model API for Triple Space with proper extensions for handling publish-subscribe model. This coordination API is inspired by the combination of TSpace API, JavaSpace API and SIENA API [Carzaniga, 1998].

<table>
<thead>
<tr>
<th>Extended Triple Space API and description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void write (Set t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Write one or more triples in a concrete Triple Space identified by a unique identifier. A triple or set of triples that is written is not visible outside its transaction until the transaction successfully commits.</td>
</tr>
<tr>
<td><strong>Triple take (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Return a triple (or nothing) that match with the template and delete the matched triple from a concrete Triple Space. A read may match any entry written under that transaction or in the entire concrete space identified by a unique id</td>
</tr>
<tr>
<td><strong>Triple waitToTake (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <em>take</em> but the process is blocked until the a triple is retrieved</td>
</tr>
<tr>
<td><strong>Triple read (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <em>take</em> but the triple is not removed</td>
</tr>
<tr>
<td><strong>Triple waitToRead (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <em>read</em> but the process is blocked until the a triple is retrieved</td>
</tr>
<tr>
<td><strong>Set scan (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <em>read</em> but returns all tuples that match with t</td>
</tr>
<tr>
<td><strong>long countN (Template t, Transaction txn, IdSpace id)</strong></td>
</tr>
<tr>
<td>Like <em>read</em> but returns all tuples that match with t</td>
</tr>
</tbody>
</table>
### void subscribe ((IdSubscriber s, Template t, Callback c), IdSpace id)
A consumer expresses its interested on triples that match with template t in a concrete Triple Space. Any time that there is an update in the space, the consumer receives a notification that indicates that there are triples available that match the template. The notification is executed by calling a method/routine specified by the subscriber.

### void unsubscribe ((IdSubscriber s, Template t, Callback c), IdSpace id)
A consumer deletes its subscription, and no more related notifications are received.

### void advertise ((IdPublisher p, Template t), IdSpace id)
A producers shows its intention to provide tuples that match t. Advertisement provides information to the system that can be used in advance to improve the distribution criteria of data and participants.

### void unadvertise ((IdPublisher p, Template t), IdSpace id)
A producers shows that it will not provide more tuples that match t

Table 5.1 Extended Triple Space API

6. Integration of Triple Space Architecture with WSMX

The interaction model Triple Space Computing together with proper extensions for handling publish-subscribe model envision a new communication paradigm for Semantic Web services. Given the fact that the WSMX Architecture relies on the main principles of Service Oriented Architecture (SOA), the benefit of the integration of Triple Space and publish-subscribe model in WSMX is threefold. Firstly, WSMX can take advantage of the features of Triple Space and publish-subscribe models in order to achieve easier several main goals:

- A loosely coupled architecture (space, reference, time and flow) where WSMX components can interact asynchronously with each other by publishing and reading triples in a share persistent place. A subscription mechanism guarantees that consumers can access asynchronously to data that it is not published yet in the space.
- Hide heterogeneity and provide uniform view of the coordination model for components and the persistent storage mechanism. Since Triple Space use RDF as a data model, and WSML provide an RDF syntax, WSMO elements can be stored in the space and can be easily accessed and modified by simple operations read and write. Contrary to blackboard systems, Triple Space can be not only a coordination mechanism but also the repository in which the data required by WSMX components is stored. Merging the persistent storage repository and the coordination mechanism has the advantage that a simple common API can manage all the data flowing/stored in the system. Another advantage is that data for coordination purposes can be enriched by data stored in repositories. Thus less data for coordination purposes is required because it is possible to refer to data stored in the repositories.
Reliability and robustness will be increased by using a communication mechanism that is persistent. Because the space can be replicated, the changes in the space can be tracked (allowing the integration of versioning mechanisms), and the access can be restricted, the communication between components is more reliable than sending and receiving messages.

The debugging and monitoring of the system can also be simpler because the information flows generated during a concrete execution can be easily retrieved and visualized.

Secondly, the presence of Triple Space as a part of the WSMX Architecture allows components to interact through a semantic platform. Invocation and interoperation between components can be done by reading and writing data semantically described. Adding semantics to the WSMX architecture drives the evolution of the system from Service Oriented Architecture (SOA) towards Semantic Service Oriented Architecture (SSOA).

Finally, the definition of a real scenario (the interaction between the WSMX components) where we will study the applicability of Triple Space from a practical point of view will provide many useful experiences that should contribute to develop the concept of Triple Space from a bottom-up approach.

6.1 A close look in the WSMX Core

In section 2, a brief description of the conceptual view of WSMX architecture was provided. In this section, we delve inside the WSMX core in order to understand how Triple Space could be integrated as a part of the WSMX architecture.

WSMX Core [Haselwanter, 2005] is a microkernel implementation based on the JMX specification. There are three layers currently defined in JMX (Cf. Figure 6.1) which are briefly described below:

- **Instrumentation Level**: in this level, interfaces for each component are defined that allow WSMX Core to uniformly manage and accesses these components. Basically, it is done by creating Java objects (called Managed-Beans or MBeans) that expose configurable attributes, accessible operations, and events. Legacy (non-JMX) devices and servers can be readily "adapted" into a JMX-manageable resource by providing a Java MBean wrapper.

- **Agent Level**: A JMX agent is a standard management agent that directly controls resources and makes them available to remote management applications. JMX agents are usually located on the same machine as the resources they control, but this is not a requirement. The core component of a JMX agent is the MBean server, a managed object server in which MBeans are registered.

- **Distributed Services Level**: Protocol adaptors and standard connectors make a JMX agent accessible from remote management applications outside the agent's Java Virtual Machine (JVM).
The WSMX microkernel is a management agent according to the JMX specification, and it is responsible for loading and configuring, executing, monitoring WSMX components. In the current specification it is possible to achieve hot-deployment, avoiding the limitation of stopping the execution of the system anytime that we want to load a new component. Remote consoles...
for administration purposes can interact with the kernel through WSDM (Web Services Distributed Management), HTTP (Hypertext Transfer Protocol) and RMI (Remote Method Invocation) adapters. An important element that it is tightly integrated with the microkernel is the workflow engine that it is responsible of identifying and executing the appropriate set of operations (execution logic) that can resolve a concrete request received by the system. Currently four execution semantics are defined:

realiseGoal(WSML):WSMLConfirmation
This entry-point allows a requester to discover and invoke a Web service without exchanging additional messages.
receiveGoal(WSML):WebService[]
This entry point allows a requester to get all Web Services that fulfilled a given Goal.
receiveMessage(WSML):WSMLConfirmation
This entry point allows a requester to invoke a discovered/selected Web service based on a related and particular choreography.
storeEntity(WSML):WSMLConfirmation
This entry point provides an interface for storing any WSMO-defined entities.

6.2 Anatomy of a WSMX Component

Three main elements can be identified in a standard WSMX component [Haselwanter, 2005] (Cf. Figure 6.2):

- The **component** itself;
- The **interface(s)** based on the WSMX W3C specification; and
- The **Java Service Wrapper**. The Java Service Wrapper made the components accessible to the WSMX Core by exposing attributes, accessible operations and events. Java Service Wrapper separate component implementation from communication issues. Through the Java Service Wrapper, a component can communicate with other components, and the WSMX Core can instantiate threads for each component and monitor the resources that each thread is using. This information allow the system to determine how many threads can be executed at the same time. In case of degradation of performances, the system can limit or reduce the numbers of threads.

The **Java Service Wrapper** encapsulates the functionalities of different components in three different modules: the **reviver** module, the **proxy** module and the **transport** module. The **transport** module implements the low level details of the communication. There will be a transport module for each communication infrastructure supported inside of WSMX (queues, tuple space, triple space, etc).

The **reviver** module manages the interaction of each of the threads that represent a running instance of every component(s) with the transport layer. The reviver defines the set of relevant events that should be attended, for each instance of every component.

The **proxy** module makes the interaction between components possible as well as simulating asynchronous communication between them. The wrapper of each component contains as many proxies as components registered to the WSMX Core. Each of these proxies included a description of the interface of the component that represents. When a component needs to interact with
other component, the related proxy will provide the name of the methods that can be invoked and a description of the associated parameters that should be provided. The suitable proxy stops the thread, packages the synchronous call and asks the transport module to put the information in an appropriate queue or in the triple space. Because several threads can be associated to a single component, the component communication behavior looks asynchronous although each thread only supports synchronous communication.

![Figure 6.2: Anatomy of a WSMX Component](http://wsmo.org/TR/d21/v0.1/)

6.3 TSC in a local WSMX configuration

In a *local WSMX configuration*, all the components are executed in the same machine or at least in the same Local Area Network (LAN). Currently, the WSMX Core designers want to distinguish between the data flows related with the *business logic* (execution of components based on the requirements of a concrete operational semantic) and the data flows related with the *management logic* (monitoring the components, load-balancing, instantiation of threads, etc).
Our proposal takes this distinction into account and distributes the WSMX information flows into two main communications mechanisms (Cf. Figure 6.3): JMX notifications (management logic) and Triple Space (business logic).
A local configuration (same machine) of WSMX Kernel and components will require a simplified version of the Triple Space infrastructure [Cf. Figure 6.4]. Sophisticated mechanisms for providing remote access (through HTTP for instance), security and trust will not be necessary.

The **Transport** module of the wrapper of each component will access the Triple Space through simple **APIs**. These APIs will implement the operations described in section 5.1. Triple Space infrastructure for WSMX components will be implemented around 6 main components: management module, publish-subscribe module, query module, data module, resource handler layer and security module. The **management module** will coordinate all the reading and writing requests received; will dispatch the request to the appropriate functional module (data module or query module); will monitor the appropriate execution of the rest of the elements of the system; and will periodically check the coherence of the information stored in the space. The monitoring activities will be collected and evaluated to determine which components are producing or getting more data. This information can be combined with the activity data that WSMX kernel is collecting for each of the components (threads, memory, etc) in their JVM. The result of this merging can provide a complete picture of the workload of the system, and can be the basement in which builds mechanisms for balancing the workload. The **publish-subscribe module** (motivated in section 5.1) is in charge to collect and store subscriptions and advertisements from consumers and producers. Anytime that a triple is published in the space, this module will check if related subscriptions are stored. In case that there are related subscriptions, publish-subscribe module will notify to the consumers of those subscriptions that there are triples available. Based on the management information collected by the management module, the publish-subscribe module can prioritize the order of notifications and deliver first to those components which have less workload. The **query module** will verify the correctness (syntax level) of the query received based on an standard query language (to be defined). The **data module** will execute all the operations that are related with the manipulation of data in the space (basically writing, modifying and deleting the triples). The **version module** will track and store the changes performed by the Data module, and will provide a versioning support to identify different versions, to re-construct a previous version of the space, etc.

It is important to take into account that query and data manipulation operations are not performed by the Query and Data modules directly. Instead, the **resource handler** will hide heterogeneity by providing a uniform view of different repositories (RDBMS, ODMS, Memory, etc.). For instance, this module will transform a query from the Query Module into the concrete query language used to query a repository). The repositories should provide native data manipulation operations and query/reasoning mechanisms.

Finally, the **security module** maintains an ontology with the access rights of each component to any of the elements stored in the Space. For instance, when a query is executed the results are filtering to verify that only authorized data will be delivered to a specific component.
The **Triple Space component** and the **execution semantics component**, currently integrated as a part of the **WSMX Core**, work tightly together because the former is the mechanism used by the execution semantics component to generate and distribute sets of operations that have to be executed by the appropriate components (i.e. discovery, mediation, parser, communication, etc). Any time that the WSMX Core received a request, the Core identifies the appropriate execution semantics and generates a suitable RDF representation of the set of operations associated with the selected execution semantics. Thanks to the Triple Space that *keeps track* of all (read and write) operations and the components that are interacting with the space, the Core can *monitored* the state in the processing of each execution semantic.

### 6.4 Triple Space in a distributed WSMX configuration.

[Haselwanter, 2005](http://wsmo.org/TR/d21/v0.1/) envisions a distributed configuration of WSMX components executed in different machines using a hierarchical configuration of WSMX microkernels (Figure 6.5).

Figure 6.5: WSMX distributed configuration [Haselwanter, 2005] *(to be added)*
In our opinion, Triple Space can still be the communication mechanism used in distributed WSMX configurations. In this scenario, all components independently of the WSMX microkernel in which are plugged, will collaborate to complete tasks defined by published RDF representation of execution semantics that were generated in response to requests received in each microkernel. We are studying different approaches to implement a distributed configuration of Triple Space, and we are also evaluating the possibility to define subspaces inside of the virtual Triple Space to restrict access to the communication of the data manage for the different components (Figure 6.6).

Figure 6.6: WSMX distributed configuration using a virtual Triple Space formed by several sub-spaces *(to be added)*

7. Interaction between WSMXs through Triple Space

The communication model used in the current implementation of WSMX is synchronous. Synchronous communication is beneficial when immediate responses are required. Since WSMX is dealing with Web service Discovery, Mediation and Invocation, immediate responses are usually not available. The reasons for such high response latency being network congestion, slow processing, third party invocation, etc. In such situations, the synchronous communication will be costly as it forces the system (component) to remain idle until the response is available. In order to minimize such overhead imposed by synchronicity, Triple Space can serve as a communication channel between WSMXs *(Cf. Figure 7.1)* thereby introducing synchronicity between communicating parties. The Triple Space supports purely asynchronous communication that optimizes performance as well as communication robustness. Enabling asynchronous communication between WSMXs brings them a step closer to their architectural goal i.e., to support greater modularization, flexibility and decoupling between communicating WSMXs. Similarly, it enables WSMX to be highly distributed and easily accessible. Furthermore, being a third party element Triple Space has added advantage for resolving any communication disputes that may arise.
Before going into the detail of how WSMXs can communicate through Triple Space, a simple logical sequence of basic
communication between a WSMX and a Triple Space is described below.

1. A sending (or receiving) WSMX performs a write (or read) request to Triple Space through Triple Space API.
2. After receiving the write (or read) request, Triple Space executes a write (or read) operation if and only if the sending (or receiving) WSMX is genuine. In other cases, the Triple Space refuses to execute the write (or read) request made by the sending (or receiving) WSMX.
3. After successfully executing the write (or read) request made by the genuine WSMX, the Triple Space responds the sending (or receiving) WSMX with a positive acknowledgement (or result set). If the write operation fails, the Triple Space responds to the sending WSMX with a negative acknowledgement. The refusal of execution of the write (or read) request made by the non-genuine WSMX is responded to with a no-operation acknowledgement.

In order to enable such communication, following two basic operations are defined in Triple Space API:

- **write**: the write operation puts a set of triples into the Triple Space. The signature of the write operation is the following:

  \[
  \text{write} \{ \text{Triples}, \text{spaceAddress} \} 
  \]

  The parameters **Triples** and **spaceAddress** specify the data to be stored in the Triple Space located at **spaceAddress**. The Triples are RDF triples and the spaceAddress is a valid URL. An example Triples (written in Notation 3 [Berners-Lee] format) of a goal description for buying a book can be found in Appendix A.

- **read**: the read operation gets a set of triples from the Triple Space. The signature of the read operation is the following:

  \[
  \text{<Triples> read} \{ \text{spaceAddress}, \text{query} \} 
  \]

  The parameters **spaceAddress** and **query** specify the location of the Triple Space from where to get the data as specified in the **query**. A example query that can be executed in read operation is listed in Appendix A. Both read and write operations are atomic in nature.

**Handling Context Information**

The information stored in Triple Space are in the form of RDF triples. Multiple users may exist for a Triple Space. A subset of these users may be interested for a particular information stored in the Triple Space that is produced by an specific user. Therefore, unlike in TS, not only the information but also the source (context) of this information is important in Triple Space communication. In addition, context information may be necessary for tracking the source of information stored in the Triple Space. Tracking the source is one of the fundamental requirements when the quality of the information is important [Guha et al., 2004]. Though each RDF triple can be uniquely identified through URIs, it is not possible to relate these triples with other factors that influence its existence, e.g., source of the triples. Techniques like reification are introduced in order to embed context...
information in RDF triples. However, usage of reification introduces a significant performance overload. At least one extra triple would be necessary for providing context information to each triples. One possible solution to this problem could be to extending RDF triples to RDF quads. However, it is against the spirit of RDF graph model.

In WSMX Triple Space, the context information is captured by adding an extra node called ‘context node’ (Cf. Figure 9). Where context node can either be an empty node or may accommodate context id. The addition of context node and handling of context information is done within application. Handling context information through application eliminates the risk of breaking RDF graph model. In order for enabling Triple Space Manager to handle context information, the write operation defined above is extended as follows.

\[
\text{write } \{ \text{<Triples>, <spaceAddress>, <contextInfo>} \}
\]

However, the read may remain unchanged as the context information, if needed, can be formulated in the query parameter.

![Context Information with context node](image)

Figure 9: Context Information with context node

### 8. Conclusions and Future Work

In this deliverable we introduced the concept of Triple Space computing, as a technological combination of space-based computing and Semantic Web, in particular in the form of RDF. Triple Spaces provide an asynchronous communication mechanism for machine-to-machine communication that supports four types of autonomy: time, space, reference (from space-based computing) and data schema autonomy due to the Semantic Web. The Triple Space approach follows the Web-style of persistently publishing and read.

The introduced communication mechanism is then applied to the Web Services Execution Environment (WSMX) to provide it with an asynchronous communication means for inner- and inter-WSMX data exchange. To better understand the benefits for (and the effects on) WSMX, a closer look at the WSMX architecture is given. Furthermore we proposed to enhance the fundamental
Triple Space idea presented by [Fensel D., 2004] with notification, advertisement and non-blocking reads. These additional interfaces are primarily borrowed from publish/subscribe paradigms.

Based on the background research and the technical results presented in this deliverable, we are implementing prototypical Triple Space infrastructure for inner- and inter-WSMX communication. In order to be able to test the implementation, further efforts are put in the definition of a triple notation for the WSML languages and tools to automatically map WSML conceptual syntax to a WSML/RDF syntax [WSML]. A first example of a goal description in triple notation is given in Appendix B.

Acknowledgment

The work is funded by the European Commission under the projects DIP, Knowledge Web, Ontoweb, SEKT, and SWWS; by Science Foundation Ireland under the DERI-Lion project; and by the Austrian government under the CoOperate programme.

The authors would like to thank all members of the WSMX working group for their advises and inputs to this document, and in particular to Thomas Haselwanter.

References


Appendix A

A1. Linda

Linda has four essential functions to access modify and delete tuples in the TS, which may be implemented in any programming language to form a Linda dialect of that language. A short overview of these functions is given below.

**out(<tuple>)**

inserts a tuple in the TS. Out never blocks a tuple.

**rd(<template(tag, list)>)**

retrieves all tuples which match a given template from the TS. It can be both blocking and non blocking.

**in(<template(tag, list)>)**

retrieves and removes all tuples which match a given template from the TS. Note that this function blocks until at least one
matching tuple appears in the Tuple Space.

\[ \text{eval(<function-tuple>)} \]

creates an active tuple and evaluates it. The results are stored as a passive tuple in the TS.

Since TS is an "associative memory", tuples do not have a physical address. A tuple can be identified and retrieved by matching its data fields with the elements found in the requesting template. Matching is carried out according to the following criteria. A tuple to be retrieved must contain the same number of elements, the same types of elements in the same order, and possibly the same values as the elements contained in the template. For example a tuple \("\text{numbers}"\), \(2\), \(2.1\) in the TS can be referenced by a template requesting a tuple whose first element is a string, its second element is an integer and its third element is a float, by using the following operation: \(\text{rd(?s, ?i, ?f)}\). Once the match has been found, \(s\) is assigned "\text{numbers}"\), \(i\) is assigned 2, \(f\) is assigned 2.1 and this data tuple is stored in TS. The tuples can be matched by using any combination of its ordered element values and/or types. [Christian, 1997] (todo reference)

The actual parameters appearing in a tuple collectively constitute a structured name. For example \(\text{in(P, 2, j:Boolean)}\) requests a tuple with a structured name "\text{P, 2}". Structured naming is in principle similar to a "select" operation in a relational database, and can make TS content addressable. Any component of a tuple, except the initial name-valued actual may be a formal like in this case: \(\text{out(P, i:integer, FALSE)}\). Actuals in templates match tuple actuals and formals. Formals in templates cannot match formals in tuples.

Linda differs from other distributed programming languages in the fact that it is a "coordination language" with the following characteristics:

1. **Reference decoupling.** The exchange of messages between processes applied to an address-space is replaced with a simple set of operations applied to a "shared memory" (i.e the TS). So objects in the TS are located based on matching the object's data fields (i.e values and types) with those contained in the requesting template. (i.e associative lookups)

2. **Space and Time decoupling.** An application can perform an operation even when the application that makes use of it does not yet exist (i.e space decoupling), and can terminate before the operation is used (i.e time decoupling). One of the effects of space and time decoupling is that address-space disjoint processes should be able to share the same variable by depositing it in the TS. The TS ensures that this global variable is maintained atomically.

3. **Separation of coordination and programming.** Linda focuses on coordination only. It is not influenced by the characteristics of a specific programming language, leading to a clear coordination model.

4. **Universality.** Linda is capable to express all major styles of coordination in parallel programs [CG89a] (todo reference).

5. **Reduced development.** There is no need to name receivers or senders but just to write objects to the TS and read them later. Linda's simplicity, portability, ease of use, and efficiency have enabled implementations on different computer platforms (e.g Intel iPSC-2 hypercube, VAX/VMS, Sequent, AT&T Bell Lab'sS/Net, Sun, DEC, Apple Mac II, and Commodore AMIGA 3000UXworkstations) and many different programming languages (i.e C, FORTRAN,Modula-2, and Lisp) [Cline 1994](todo-reference) [Gelernter, 1985].

These characteristics can benefit distributed applications on the Web.
The processes of a distributed application share a single namespace and a single TS. The internal representation of tuple names in one application is prefixed with program IDs to protect the set of tuples from unwanted reference, augmentation or deletion from another program. In order for an application to access names in the namespace of another application, names of tuples may be exported from one application to another for use in out() statements only. The kernel can enforce limitations on the uses to which these names can be put by trapping the in() and rd() statements that occur outside the application which has exported the namespace.

In addition to the well known Linda implementation for the SBN network at Stony Brook other implementations have provided solutions to Tuple Space related implementation issues. For example in order to overcome the problem of having multiple concurrent readers of a set of tuples, Antony Rowston developed a primitive for Linda called "copy-collect", which makes use of multiple tuple spaces. Also, he developed Bonita in order to overcome the problems of high latency in TS based coordination languages and to provide the ability to request multiple tuples and act on them as they arrive. With this solution an in() operation is decomposed into a request for a tuple and a receive tuple. Added primitives block until a result tuple is available.

A2. TSpace

Tuples can be accessed and modified using a simple Java API. The TSpaces basic operations set [Lehman et al.,2001] used to access, modify and delete the tuples in the TS is given below:

- **write(tuple)**
  adds a tuple to TS.

- **take(<template>)**
  performs an associative search for a tuple that matches the template. If the tuple is found it is removed from the TS and is returned. If not found null is returned.

- **waitToTake(<template>)**
  like take except that it blocks until a match is found.

- **read(<template>)**
  like take except that the tuple is not removed from the TS.

- **waitToRead(<template>)**
  like waitToTake except that the tuple is not removed from the tuple space.

- **scan(<template>)**
  like read except that the entire set of tuples that matches is returned.

- **eventRegister(command, template, callback routine)**
  register for an event corresponding to the command and the template tuple.

- **countN(<template>)**
  like scan except that it returns a count of matching tuples.

- **rendezvous**
  operator rhonda (to be investigated).
Here is an example of how easy is to create a TS and write a tuple into the TS.

```java
String host = "localhost";
TupleSpace ts = new TupleSpace("Example1", host);
Field f1 = new Field("Key1");
Field f2 = new Field("Data1");
Tuple t1 = new Tuple();
t1.add(f1);
t1.add(f2);
ts.write(t1);
```

TSpaces extend the basic Linda TS framework with relational data management, event notification, access controls features and the ability to download both new data types and new semantic functionality. Any client application providing a service to the user can be added to TSpaces (e.g. email service, printing service, pager service and so on). System upgrades can be performed while the TSpaces server is running. This reduces costly downtime for system upgrades.

Future development of TSpaces has been stopped after it was officially declared a success in 2001. Currently TSpaces is being used as a communication paradigm for component interaction and management to explore Grid Computing in the IBM OptimalGrid project. [Foster et al., 2001]

A3. JavaSpace

A short overview of basic JavaSpaces operations, to access and modify tuples is given below:

- **write(entry):**
  puts a entry into the Tuple Space.

- **read(templateEntry):**
  return a matching entry from the Tuple Space, or a indication that the no match was found. If there are multiples entries that match the template an arbitrary entry is returned. There are two types of read:
    - **read()**
      is blocking; returns null if timeout expires.
    - **readIfExists()**
      is nonblocking.

- **take(templateEntry):**
  like **read** operation but if a match is found, the matching entry is remove the Tuple Space. There are two types of take:
    - **take()**
      is blocking; returns null if timeout expires.
    - **takeIfExists()**
      is nonblocking.
notify:
   send event if matching entry is written into the Tuple Space.

All the operations that were mentioned before are performed in a transactionally secure manner using the two-phase commit model [JavaSpace, 2003]. JavaSpaces services can provide a reliable distributed storage system for the entries.

JavaSpaces also uses features of the Jini [Arnold et al., 1999] network technology, such as leases, transactions and events.

Jini is an open architecture based on idea of federating groups of users and the resources required by those users [Jini, 1999]. Using Jini technology one can build adaptive networks where many devices can join in a scalable way.

Appendix B

B.1 Example Triple Notation for WSML Goal

The triples given here describe a goal as it is defined in WSML. The goal is: "Buy a Book".

```sparql
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix wsml: <http://www.wsmo.org/wsml#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

wsml:buyBookGoal rdf:type wsml:goal .

   _:p1 dc:title "Purchase Order Book Ontology"^^xsd:string .

   _:p2 dc:creator "DERI International"^^xsd:string .


   _:p4 dc:publisher "DERI International"^^xsd:string .

```

wsml:buyBookGoal wsml:hasPostcondition _:z.
_:z wsml:hasAxiom <http://www.wsmo.org/goals/buyBookGoal#test3>.
_:z wsml:hasLogicalExpression "input is input"^^xsd:string.

B.2 Example Query Template

<> ql where {
  wsml: buyBookGoal rdf: type wsml: goal.
}

This query retrieves all wsml:buyBookGoal Triples that are of type wsml:goal

---

Change Log

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005/06/21</td>
<td>0.1</td>
<td>B. Sapkota</td>
<td>Updated section 7.</td>
</tr>
<tr>
<td>2005/06/20</td>
<td>0.1</td>
<td>E. Kilgarriff</td>
<td>Reviewed entire content for grammatical, formatting &amp; conceptual mistakes.</td>
</tr>
<tr>
<td>2005/06/19</td>
<td>0.1</td>
<td>Francisco</td>
<td>Added section 6.4</td>
</tr>
<tr>
<td>2005/06/19</td>
<td>0.1</td>
<td>Francisco</td>
<td>Added Chris as author and re-organized author list in lastname alphabetic order</td>
</tr>
<tr>
<td>2005/06/19</td>
<td>0.1</td>
<td>Francisco</td>
<td>Rewrote section 4, adding input from Chris</td>
</tr>
<tr>
<td>2005/06/19</td>
<td>0.1</td>
<td>Francisco</td>
<td>Added introduction to section 3</td>
</tr>
<tr>
<td>2005/05/16</td>
<td>0.1</td>
<td>Francisco</td>
<td>Added comments of Thomas Haselwanter on section 6</td>
</tr>
<tr>
<td>2005/05/16</td>
<td>0.1</td>
<td>Francisco</td>
<td>rewrote section 5 and added comments of last review on section 6</td>
</tr>
<tr>
<td>2005/05/16</td>
<td>0.1</td>
<td>Francisco</td>
<td>Added changelog</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Author</td>
<td>Change Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-----------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>2005/05/16</td>
<td>0.1</td>
<td>B. Sapkota</td>
<td>Section 7: Added handling context information</td>
</tr>
<tr>
<td>2005/05/08</td>
<td>0.1</td>
<td>B. Sapkota</td>
<td>Section 7: Example moved to Appendix A</td>
</tr>
</tbody>
</table>