WSMX Working Draft

D21.v0.1 WSMX Triple-Space Computing

WSMO Working Draft 11 February 2005

This version:
 http://www.wsmo.org/TR/d21/v0.1/20050211
Latest version:
 http://www.wsmo.org2/TR/d21/v0.1/
Previous version:
 http://www.wsmo.org/2005/d21/v0.1/20050202

Authors:
 Brahmananda Sapkota
 Edward Kilgarrif
 Francisco Jose Martin-Recuerda Moyano
 Ioan Toma
 Reto Krummenacher
Editors:
 Brahmananda Sapkota
 Francisco Jose Martin-Recuerda Moyano

This document is also available in non-normative PDF version. Copyright © 2005 DERI, All Rights Reserved. DERI liability, trademark, document use, and software licensing rules apply.

Executive Summary

This deliverable will identify and describe the concept of Triple Space Computing [Fensel D., 2004] (TSC) and find its scope in WSMX [Oren et. al., 2004]. It will specifically concentrate on facilitating asynchronous communication between geographically distributed WSMXs providing a shared RDF space between them. To identify the scope of TSC in WSMX, a brief study of WSMX will be done and relationship between WSMX and TSC will be presented. Finally, interaction between WSMXs through TSC has been presented.

Table of contents

1. Introduction 1.1 Purpose of this Document 1.2 Document Overview 2. Web Service Execution Environment (WSMX) Overview 3. Tuple Space Computing 3.1 Linda 3.2 TSpaces 3.3 JavaSpaces 3.3 Other Frameworks 4. Triple Space Computing 5. Publish Subscribe Paradigm 6. Integration of TSC with WSMX Architecture 7. Interaction Between WSMXs Through TSC 8. Conclusions and Future Work References Acknowledgment

1. Introduction

Triple-Space Computing is a simple and powerful paradigm that inherits the communication model from Tuple Space Computing (TSC) model and projects in the context of Semantic Web Service [Fensel, 2004]. It is based on the evolution and integration of other known technologies such as Tuple-Space Computing [Gerlernter, 1992], Persistence Message-based Architecture [Alonso et. al., 1995], Semantic Web and on RDF Schema [Brickley and Guha, 2004]. It provides a persistant shared space for participating applications to enable seamless interaction without having the need of direct message exchange between them. Web Service Execution Environment (WSMX) enables the dynamic discovery, selection, mediation, invocation, and interoperation of Semantic Web Services [Oren et. al., 2004]. WSMX in its current state supports synchronous communication with other WSMXs and or systems. While integration process incorporates large number of systems, more than one WSMX will be engaged for facilatating such integration processes. In addition, one WSMX may require to communicate with other WSMX in the due course. Such communication will become costly when WSMXs are heavily loaded and the processing time is high. In such cases TSC 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.

1.1 Purpose of this Document

This document describes the possibility of using TSC in WSMX to enable distributed asynchronous communication between WSMXs. This document starts with the definition of TSC and WSMX; finding the possible relationships between them. One of the main focuses of WSMX-TSC is to provide a shared space for WSMX and provide new means of asynchronous communication.

1.2 Document Overview

An overview of WSMX is presented in section two. As part of the background information, section three introduces tuple space computing. Triple Space Computing is presented in section three of the document. Likewise, publish-subscribe paradigm is presented in section five and section six highlights possible integration of TSC and WSMX architecture to enable inter WSMX interaction. Section seven is describes asynchronous communication between WSMXs through TSC and finally, section five concludes the document and looks at future works.

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., 2004] 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 manager manages the processing of all events passing them to other components in the logic layer as appropriate.

The MatchMaker is responsible for matching goals to web service capabilities. In the event that multiple web services are found matching a specific goal, the Selector is invoked to select the web service that best fits the requirements of the goal's owner. 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 Invoker is responsible for making invocations to web services as part of the execution of a goal. Invocations are based on the WSML description of the web service. The 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.

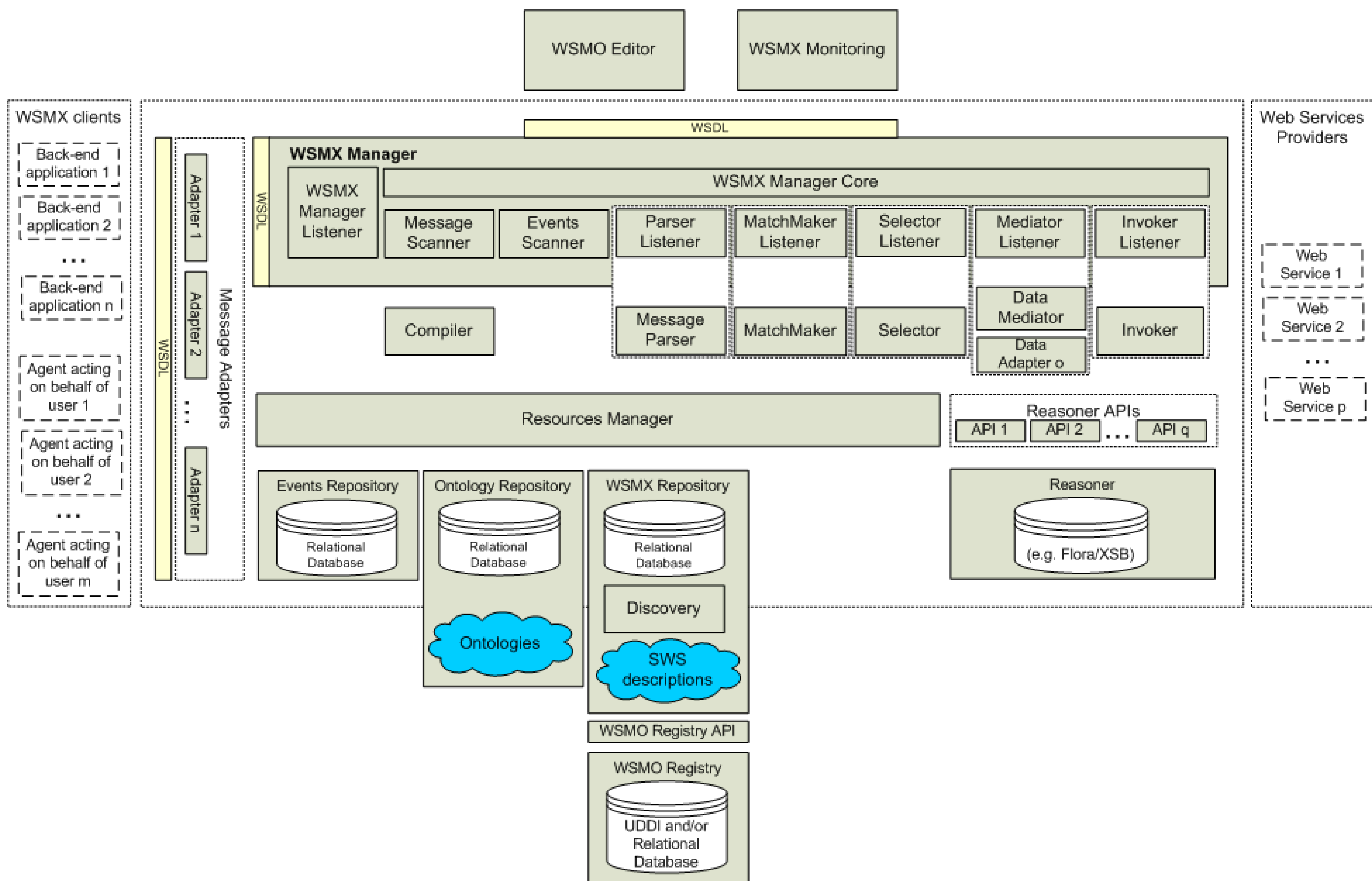

Figure 1: WSMX-Architecture

Messages & 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

Existing publish/subscribe technologies like Tuple Space Computing [Gerlernter, 1992], Shared Object Space, Persistent Message-based Architecture [Alonso et al., 1995] can serve as a fruitful basis for our endeavour. Hence to be able to understand and define Triple Space, we first need to have a look at these underlying technologies.

3.1 Linda

Linda was developed in parallel programming languages in the early 80s by D. Gelernter [Gelernter, 1992] 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 accesed as named, independent entities until some process chooses to receive them. The "Tuple Space" (TS) for process creation and co-ordination is a form of shared "associative memory" for tuples where a tuple, the primary unit of communication in Linda, is an ordered sequence of typed values. The TS shared memory may be distributed or may be served from a single server to which remote processes may connect.

Tuples are distinguished by tags (i.e symbolic names) and by the 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 TS; 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 TS 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 TS. Therefore active tuples can contain a combination of formal and actual values while passive tuples can only contain actual values.

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.

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 <"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 and integer and its third element is a float, by using the following operation: rd(?s, ?i, ?f). Once the match has been found, s is assigned "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 in(P, 2, j:Boolean) requests a tuple with a structured name "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: 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, 1988].

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.

3.2 TSpaces

TSpaces is a Java-based intermediary built upon the Linda TS coordination model with added middleware extensions developed by IBM at the Almaden Research Center. 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. The communication becomes asynchronous and anonymous. TSpaces incorporates database features, such as transactions, persistent data, flexible queries and XML support. In summary, it can be used for bringing network services to small (palm-top computers) and embedded systems [Lehman et al., 2001]. As TSpaces is implemented in Java (running on all platforms from JDK1.0), it inherits the ability both to run on virtually any platform and to download new datatypes and new functionality dynamically.

TSpaces provides a space where tuples are added and removed. In TSpaces there are two ways to define tuples:

- Directly as a Tuple object, where the individual Field instances that make up the Tuple are created immediately.
- Indirectly by defining a new object that subclasses SubclassableTuple.

The client-server communication is currently built of a seperate 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 many Tuple Spaces). It is moreover possible to have multiple transactions active simultaneously. All it needs are multiple instances of the Tuple Space all pointing to the same space.

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.

```
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]

3.3 JavaSpaces

JavaSpaces [Freeman et al., 1999] is a Java-based implementation of tuplespaces by Sun Microsystems, in which tuples are represented as serialized objects. 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*. A process that reads or takes an entry can invoke the operations that are associated with the entry. A short overview of basic JavaSpaces opreations, to access and modify tuples is given below:

write(entry):
: puts a entry into the Tuplespace.

read(templateEntry):
: return a mathcing entry from the Tuplespace, or a indication that the no match was found. If there are multiples entries that match the template an arbitary entry is retuned. There are two types of read:
  read()
  : is blocking; returns null if timeout expires.
  readIfExists()
  : is nonbloking.

take(templateEntry):
: like **read** operation but if a match is found, the matching entry is remove the Tuplespace.Thre are two types of take:
  take()
  : is blocking; returns null if timeout expires.
  takeIfExists()
  : is nonbloking.

notify:
: send event if matching entry is written into the Tuplespace.

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.

Differences between JavaSpaces and TSpaces

3.3 Other Frameworks

4. Triple-Space Computing (TSC)

Triple Space Computing (TSC) [Fensel, 2004], follows the same goals for Semantic Web Services than the Web achieved for humans: re-define and expand previous ways of communication (cf. Figure 2). TSC will become in the necessary infrastructure where Semantic Web and Semantic Web Services will become true, and as [Fensel, 2004] pointed out: "Triple Space may become the web for machines as the web based on HTML became the Web for humans".

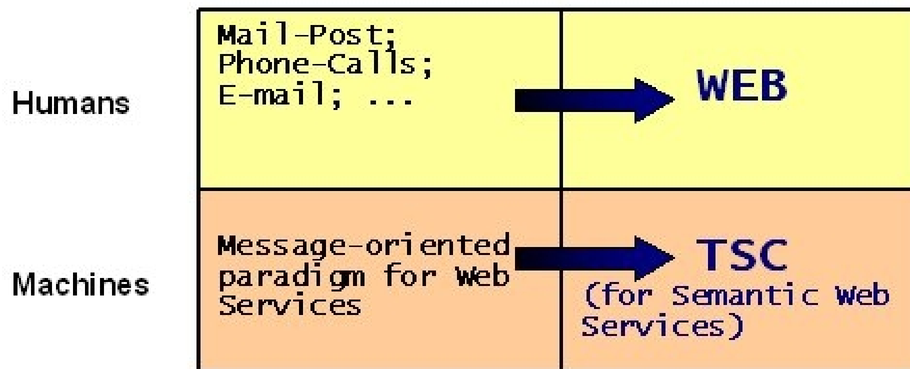

Figure 2: Evolution of communication mechanisms for humans and machines

TSC is based in the evolution and integration of several well-known technologies: Tuple Space Computing [Gerlernter, 1992], Shared Object Space (http://www.tecco.at), Persistent Message-based Architecture [Alonso et. al., 1995], Semantic Web and in particular RDF Schema [Brickley and Guha, 2004]. TSC is based on the communication model of Tuple Space Computing: Instead of sending messages forward and backward much simpler means for communication are provided. Processes can write, delete, and read tuples from a global persistent space.

- Tuple or space-based computing has one very strong advantage: It de-couples three orthogonal dimensions involved in information exchange (cf. Figure 3): reference, time, and space.
- Processes communicating with each other do not need to know explicitly from each other. They exchange information by writing and reading tuples from the tuplespace, however, they do not need to set up an explicit connection channel, i.e., reference-wise the processes are completely de-coupled.
- Communication can be completely asynchronous since the tuplespace guarantees persistent storage of data, i.e., time-wise the processes are completely de-coupled.
- The processes can run in completely different computational environments as longas both can make access to the same tuplespace, i.e., space-wise the processes are completely de-coupled.

This strong decoupling in all three relevant dimensions has obvious design advantages for defining reusable, distributed, heterogeneous, and quickly changing applications like promised by web service technology. Also, complex APIs of current web service technology boil down to a read and write operation in a tuplespace. Notice that a service paradigm based on the tuple 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.

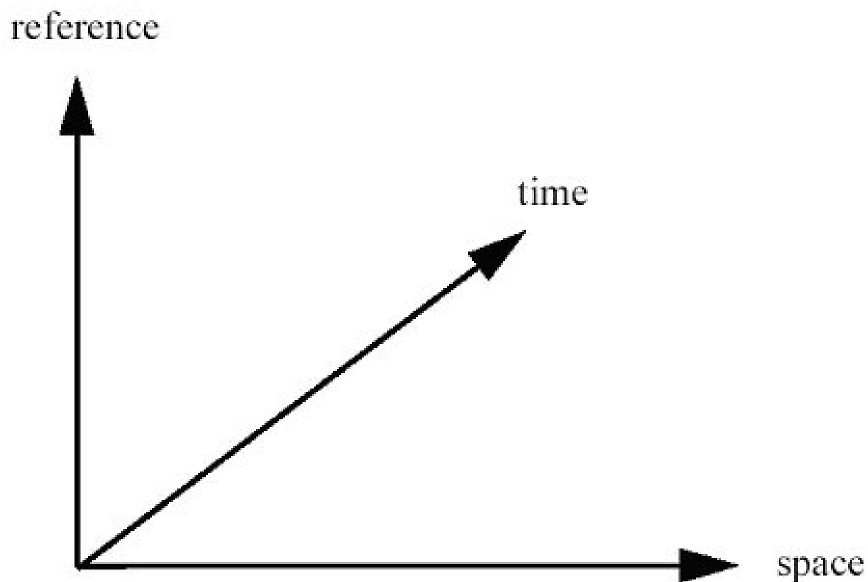

Figure 3: Three separate dimensions of cooperation [Angerer, 2002]

However, [Johanson and Fox, 2004] reports some shortcomings of the current tuple space models. They lack any means of name spaces, semantics, unique identifiers and structure in describing the information content of the tuples. This tuple space provides a flat and simple data model that does not provide nesting, therefore, tuples with the same number of fields and field order but different semantics cannot be distinguished.

We propose a simple and promising solution for this problem extending the tuple space into a triple space, where <subject, predicate, object> triples describe content and semantics of information. The object can become a subject in a new triple and so defining a graph structure capturing structural information. Fortunately, with RDF [Klyne and Carroll, 2004] this space already exists on the web and provides a natural link from the space-based computing paradigm into the semantic web. Notice that the semantic web is also not becoming unnecessary based on the Tuple-spaced paradigm, i.e. we envision current Semantic Web technologies and Tuple Space as the building blocks for a "Global Semantic Space" on the web. The global space can help overcome heterogeneity in communication and cooperation: The Tuple Space simply brought to a global scale only does not provide any answer to data and information heterogeneity. Nevertheless, this aspect is what the semantic web is all about, and what we aim to fruitfully combine.

To make this vision real, two core components are currently identified. The first one is the technical infrastructure that it will include a distributed persistent mechanism for storing triples and should also offer the following desirable features: asynchronous communication, reliability, high-availability, robustness and security. The second, core component is the ontological infrastructure that will provide a semantic specification of the domain where business processes will interact each other.

5. Publish Subscribe Paradigm

The publish-subscribe interaction paradigm is designed to be able to support asynchronous communication between communication parties by decoupling them in time and reference. It is an event based interaction paradigm where sending parties publish their events or information and the receiving parties subscribe for the event of their interest [Eugster et. al., 2003]. The events registered by the publishers are propagated asynchronously to the subscribers of

that event. This kind of interaction mechanism has been largely recognized as a promising means of communication in distributed environment [Huang and Garcia-Molina, 2001]. However, the fundamental problem in Publish-Subscribe interaction mechanism is the lack of semantics. As the interaction mechanism is purely static, it is unable to fully match subscribers interests to that of the publishers.

6. Integration of TSC with WSMX Architecture

In this section we present a basic TS architecture and discuss an integration approach between TS and WSMX. The TS architecture presented in the document is minimal and extendible. It will be extended in the due course. Integration of TSC and WSMX, we believe, will enhance the functionality of WSMX by enabling non-blocking asynchronous communication between WSMXs. The simple TS architecture consists of loosely coupled components (Cf. Figure 5). The functionalities of each individual components are briefly described below. The basic required functionalities for enabling WSMX-TS communication are as follows:

- **up** WSMX notifies to the TS that it is up and running.
- **read** WSMX reads data from the TS
- **write** WSMX writes data on to the TS

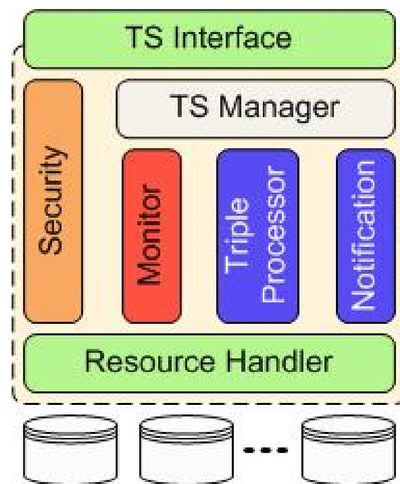

Figure 4: Triple Space Architecture.

TS Interface: the TS Interface provides functionalities that are required to enables communication between WSMX and TS. All interaction between WSMX and TS takes place through this interface.

TS Manager: the TS Manager is the gateway to the TS. It can be accessed via the TS interface. It processes all incoming requests and schedules the job. It forwards every requests that has been received to the monitoring component as well to store for future statistics.

Security: the Security component is responsible for ensuring the communication between WSMX and TS is secure.

Monitor: the Monitor component monitors all in- and out-flows of the communication.

Triple Processor: the Triple Processor component is responsible for processing all read and write operations.

Notification: the Notification component notifies the departure and the arrival of messages of messages to the sending and receiving WSMXs respectively.

Resource Handler: the Resource Handler component is responsible for dealing with any persistent repositories. These repositories could be internal or external to the TS.

In order to enable WSMXs communicate with each other TS architecture above will have to be integrated with WSMX architecture. The notification of delivery is necessary to ensure that the message has reached to the destination WSMX. Similarly, the notification of arrival of message has to be notified to the receiving WSMX as it may never read the message from TS. As the arrival and departure of the messages need to be notified, WSMX invoker has to be extended so that it can accept the notification and act accordingly.

7. Interaction Between WSMXs Through TS

In its current state, WSMX supports synchronous communication with other WSMXs. Synchronous communication is necessary when immediate responses are required. However, when there is high response latency because of some reason, e.g., network congestion, slow processing, third party invocation, etc, the synchronous communication will be costly as it forces the system (component) to remain idle for a long time. As WSMX may invoke other WSMXs and Web Services,it may not be able to provide immediate responses. Such problems can be addressed by using TSC as communication channel between WSMXs (Cf. Figure 5), aas TSC supports purely asynchronous communication. Enabling asynchronous communication between WSMXs brings WSMX as step close to achieving its 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.

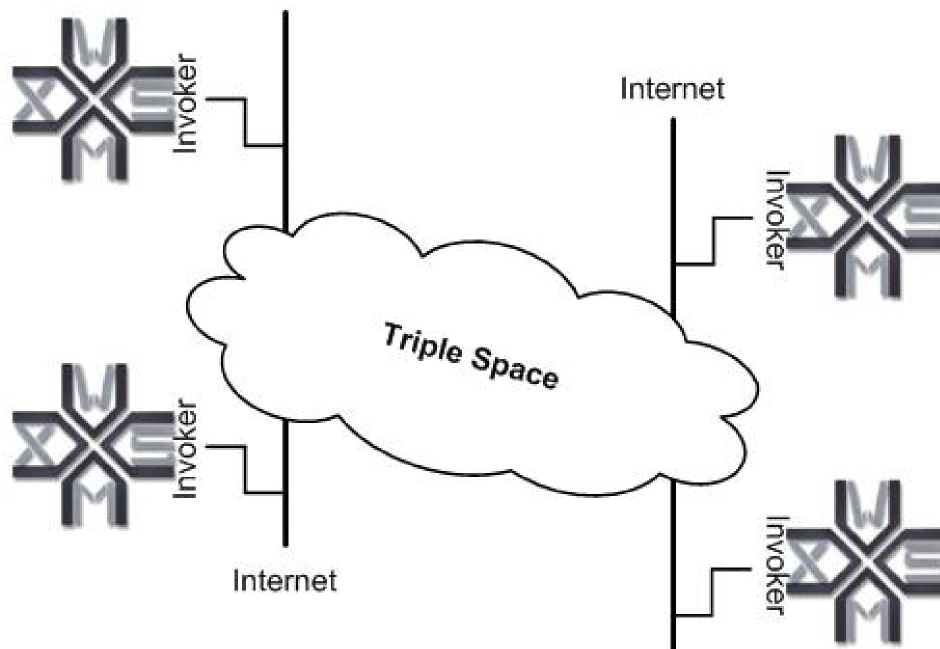

Figure 5: Communication between WSMXs through TS

The communication between WSMXs occurs in the following manner. A sending WSMX writes its message (that need to be sent to the receiving WSMX) onto the Triple Space (TS). While writing, the sending WSMX also includes its own address, the address of the receiving WSMX and the message that

needs to be delivered to the receiving WSMX. This means that the 'write' operation will look like the following: ***write*** *<source, destination, data>*. TS notifier will then notify the receiving WSMX, when it is available, about the arrival of the data destined for it. When the receiving WSMX requires this data, it will read the data from the TS. The 'read' operation willl look like the following: ***read*** *<destination, query>*. Where *destination* is the destination address specified by the sending WSMX while doing the write operation and the *query* specifies the intent of the read operation. In order to guarantee the transactability, when the receiving WSMX reads the data, TS notifier will notify the sending WSMX that the data has been consumed by the receiving WSMX. This data when no longer needed will then be removed from the TS.

8. Conclusions and Future Work

In its current state, this document presents only some sections of what we intend to do in this deliverable. Future work will provide elaborated specification of the TSC functionalities that will be used to support asynchronous communication between WSMXs.

A. References

[Alonso et. al., 1995], G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, M. Kamath: Exotica/FMQM: A Persistent Message-Based Architecture, In the proceedings of IFIP Working Conference on Info Sys for Decentralized Organizations, Trondheim, August 1995; *Also available as IBM Research Report RJ9912, IBM Almaden Research Center, November 1994.*

[Angerer, 2002], B. Angerer: Space Based Computing: J2EE bekommt Konkurrenz aus dem eigenen Lager, Datacom, no 4, 2002.

[Brickley and Guha, 2004], D. Brickley and R.V. Guha (eds.): RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation, February 2004, http://www.w3c.org/TR/rdf-schema/

[Eugster et al., 2003], P.Th. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec: The Many Faces of Publish/Subscribe, ACM Computing Survery, 2003.

[Fensel D., 2004], D. Fensel: Triple-based Computing. DERI Research Report, 2004-05-31.

[Gerlernter, 1992], D. Gerlernter: Mirrorworlds, Oxford University Press, 1992.

[Herzog et al., 2004], R. Herzog, P. Zugmann, M. Stollberg, and D. Roman(ed.): WSMO Registry. WSMO Working Draft D10,http://www.wsmo.org/2004/d10/v0.1/

[Huang and Garcia-Molina, 2001], Y. Huang and H. Garcia-Molina: Publish/Subscribe in a Mobile Environment, In Proceedings of MobiDE, 2001

[Johanson and Fox, 2004], B. Johanson and A. Fox: Extending Tuplespaces for Coordination in Interactive Workspaces, Journal of Systems and Software, 69(3), January 2004:243-266.

[Klyne and Carroll, 2004], G. Klyne and J. J. Carroll (eds.): Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation, February 2004, http://www.w3.org/TR/rdf-concepts/

[Lehman et al., 2001], T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman: Hitting the distributed computing sweet spot with TSpaces, Special Issue on Computer Networks, 35(2001):457-472.

[Oren, 2004], E. Oren. WSMX Execution Semantics. WSMO Working Draft v0.1,http://www.wsmo.org/2005/d13/d13.2/v0.2/

[Oren et. al., 2004], Moran M. and Zaremba M.: Overview and Scope of WSMX, WSMO Working Draft v0.1, http://www.wsmo.org/2004/d13/d13.0/v0.1/

[Roman et al., 2004], D. Roman, H. Lausen, and U. Keller. Web Services Modeling Ontology Standard. WSMO Working Draft v02, http://www.wsmo.org/2004/d2/v0.2/

[Zaremba et al, 2004], M. Zaremba, A. Haller, Maciej Zaremba, and M. Moran: WSMX - Infrastructure for Execution of Semantic Web Services, 2004

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 to all the members of the WSMO working group for their advises and inputs to this document.