



D21.v0.1 WSMX Triple-Space Computing

WSMO Working Draft 11 April 2005

This version:

<http://www.wsmo.org/TR/d21/v0.1/20050411>

Latest version:

<http://www.wsmo.org/TR/d21/v0.1/>

Previous version:

<http://www.wsmo.org/TR/d21/v0.1/20050307>

Authors:

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

Editors:

Brahmananda Sapkota
Francisco Jose Martin-Recuerda Moyano

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] (TripleSpace) 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 TripleSpace in WSMX, a brief study of WSMX will be done and relationship between WSMX and TripleSpace will be investigated. Finally, interaction between WSMXs through TripleSpace will be 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
 4. Triple Space Computing
 5. Publish Subscribe Paradigm
 6. Integration of Triple-Space Computing Architecture with WSMX
 - 6.1 A close look in the WSMX Architecture
 - 6.2 Anatomy of a WSMX Component
 - 6.3 TripleSpace in local WSMX Configuration
 7. Interaction Between WSMXs Through TripleSpace
 8. Conclusions and Future Work
- References
- Acknowledgment
- Appendix A

1. Introduction

Triple Space Computing (TripleSpace) is a simple yet powerful communication paradigm that inherits the communication model of pervasive communication and projects in the context of Semantic Web Service [Fensel D., 2004]. It is based on the evolution and integration of other known technologies such as Tuple Space Computing [Gelernter, 1992], Persistence Message-based Architecture [Alonso et. al., 1995], Semantic Web and on RDF Schema [Brickley and Guha, 2004]. It provides a persistent shared space for participating applications to enable their interaction without having the need of direct message exchange between them. This is made possible as applications may write, delete, and read triples in this global space.

Web Service Execution Environment (WSMX) enables the dynamic discovery, selection, mediation, invocation, and inter-operation 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 have to be engaged in order to facilitate such integration processes. In addition, one WSMX may require the ability to communicate with other WSMXs in the due course. Such communication will become costly when WSMXs are heavily loaded and their processing time is high. In such cases, TripleSpace 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 TripleSpace in WSMX to enable distributed asynchronous communication between WSMXs. This document starts with the definition of TripleSpace and WSMX; finding the possible relationships between them. One of the main focuses of WSMX-TripleSpace 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, and describes some previous implementations of this concept. Triple Space Computing is presented in section four of this document. Section five describes the publish-subscribe paradigm while section six highlights the possible integration of TripleSpace and WSMX architecture to enable inter WSMX interaction. Section seven describes the possibility of asynchronous communication between WSMXs through TripleSpace and finally, section eight 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 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 triplespace 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.

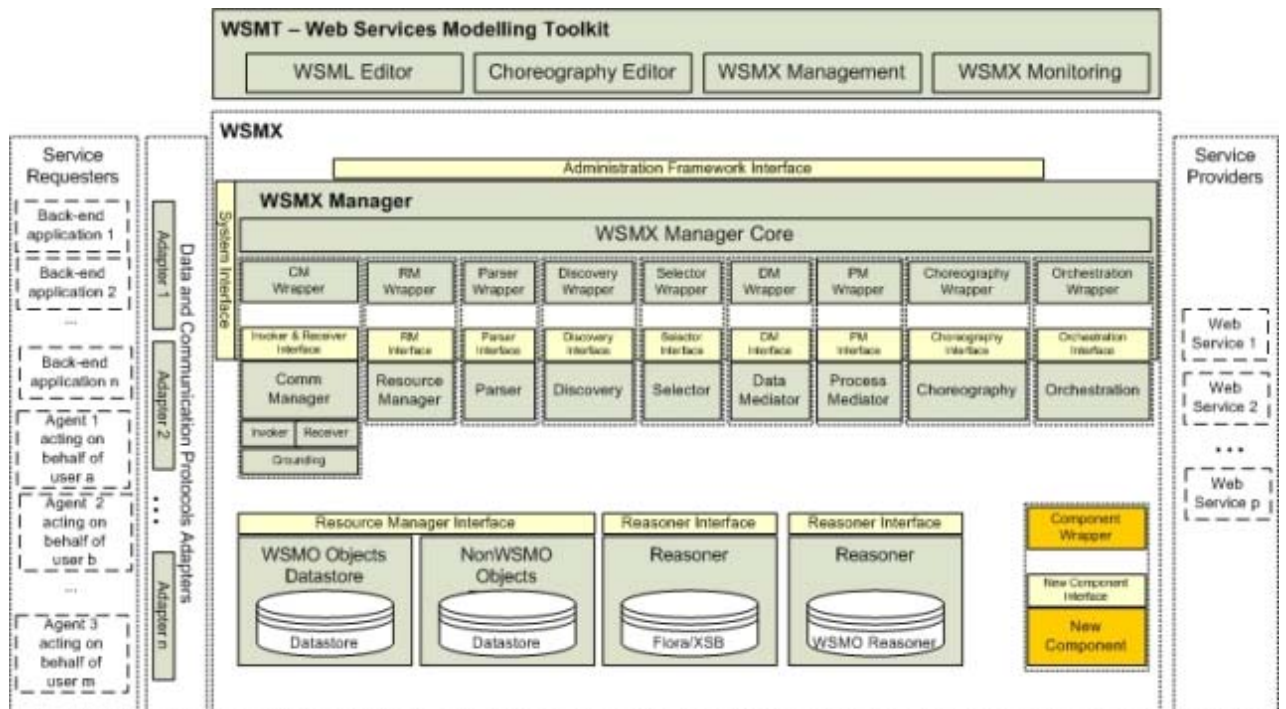


Figure 1: WSMX-Architecture

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

Existing publish/subscribe technologies like Tuple Space Computing [Gelernter, 1992], Shared Object Space, Persistent Message-based Architecture [Alonso et al., 1995] can serve as a fruitful basis for our endeavor. To be able to understand and to define TripleSpace, we first need to have a look at these underlying technologies. Therefore, we describe some of these appealing underlying technologies in the subsequent sub-sections.

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 accessed 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. A short overview of basic Linda operations can be found in [Appendix A](#)

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 Centre. 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 data types 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 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 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. A

short overview of basic TSpaces operations can be found in [Appendix A](#).

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 Client operates on a JavaSpace to write new entries, look up existing entries, and remove entries from the space. This allows storage of 'state' by Java programs. In particular when security restrictions means that applets are not allowed to store state on client machines, then the state may be stored on servers. JavaSpaces is based on a value-matching lookup routine for specified fields. A short overview of basic JavaSpaces operations can be found in [Appendix A](#).

Differences between JavaSpaces and TSpaces

To be written

3.4 Other Frameworks

To be written

4. Triple-Space Computing

Triple Space Computing (TripleSpace) [Fensel, 2004], follows the same goals for the Semantic Web Services as the Web for humans: re-define and expand current communication paradigm (Cf. Figure 2). TripleSpace will become a 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".

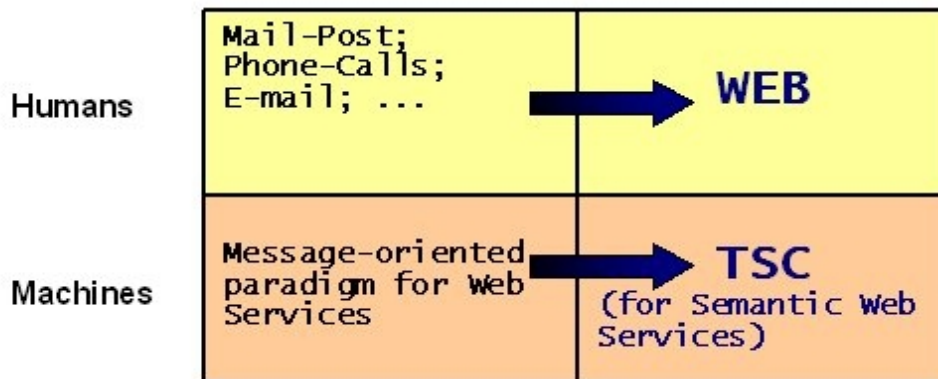


Figure 2: Evolution of communication mechanisms for humans and machines

TripleSpace 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], Semantic Web and in particular RDF Schema [Brickley and Guha, 2004]. It is derived from the communication model of Tuple Space Computing; instead of sending messages back and forth much simpler means of communication can be provided. Processes can write, delete, and read tuples from a global persistent space.

The reason for adopting Tuple Space Communication in TripleSpace is because of its inherent novel characteristics as listed below.

- Tuple or space-based computing has one very strong advantage: It de-couples the processes involved in information exchange in three orthogonal dimensions (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; they do not need to set up an explicit connection, 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 long as both can make access to the same tuplespace, i.e., space-wise the processes are completely de-coupled.

These 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 tuplespace. 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.

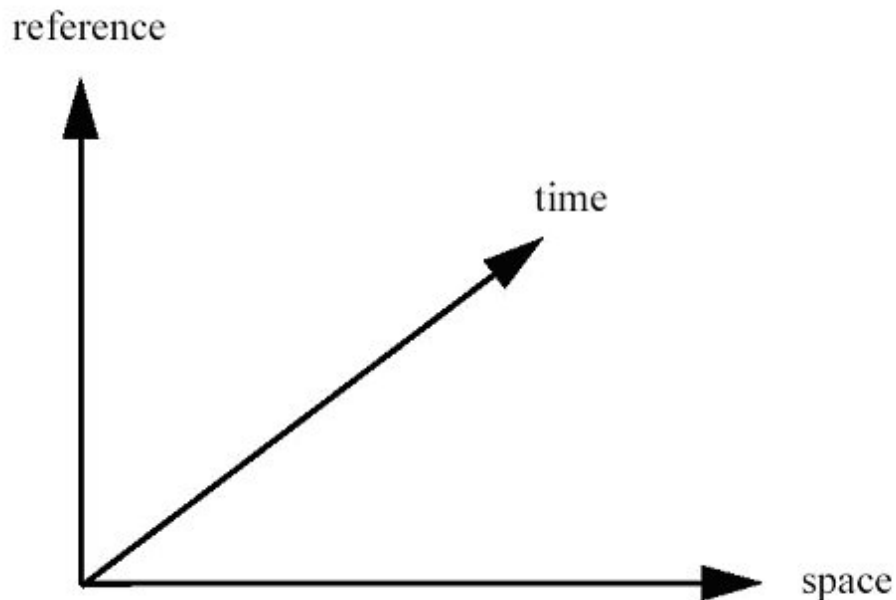


Figure 3: Three separate dimensions of cooperation [Angerer, 2002]

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 thereby capturing the 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 Triple Space Architecture with WSMX

Triple Space Computing envisioned a new paradigm communication for Semantic Web services. Given the fact that WSMX Architecture relies on main principles of Service Oriented Architecture (SOA), the benefit of the integration of TripleSpace in WSMX is twofold. Firstly, WSMX can take advantage of the features of TripleSpace in order to achieve its several main goals:

- *Loosely coupled* architecture where WSMX components can interact asynchronously with each other.
- *Hide heterogeneity* and provide uniform view of the components and the resources. WSMX (or external) Repositories can be integrated to the TripleSpace as a part of its Ontological Infrastructure thereby keeping the communication intact.
- *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* of the system can also be easier because the information flows generated during a concrete execution can be easily retrieved and visualized.

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

6.1 A close look in the WSMX Architecture

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 TripleSpace will be integrated as a part of the WSMX.

WSMX Core is designed based on the *JMX microkernel specifications*. There are three layers currently defined in JMX (Cf. Figure 4) 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:** *(to be completed)*
- **Distributed Services Level:** *(to be completed)*

Figure 4: JMX Levels *(To be added)*

6.2 Anatomy of a WSMX Component

Three main elements can be identified in a standard WSMX component:

- 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. Through the Java Service Wrapper, the WSMX Core can instantiate threads and monitor the resources that each component has been assigned in order to allow or limit the number of threads of the same component that can be executed at the same time.

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 spaces, etc).

The reviver module manages the interaction between the WSMX Core and each of the threads that represent a running instance of component(s) (Cf. Figure 5). Any interaction between WSMX Core and the components and vice-versa is done through this module.

The Proxy module handles the interaction between components (on the contrary of the reviver that only handle the interaction between each component and the core) and simulates 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 component that is represent.

When a component needs to interact with other component, the request is captured by the suitable proxy that stops the thread, packages the synchronicity of the request and asks the transport module to put the information in an appropriate queue or in the space.

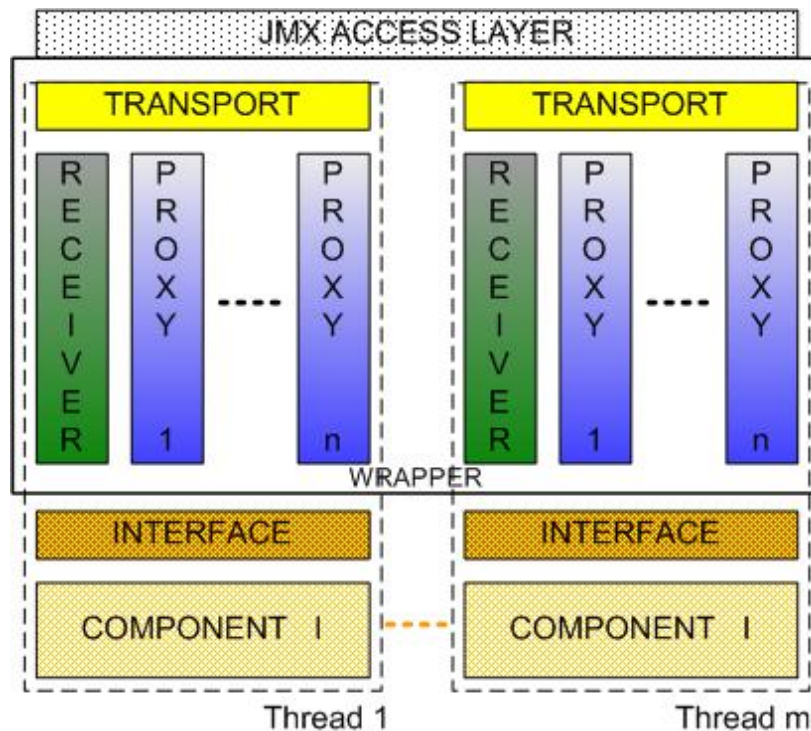


Figure 5: Anatomy of a WSMX Components

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, take this distinction into account and distribute the WSMX information flows in two main communications mechanism (see figure 6.3.1): JMX notifications (management logic) and TripleSpace (business logic).

Figure 6: Triple Space in a local WSMX configuration (*To be added*)

Because in this implementation all the components of the WSMX architecture are executed locally in the same machine or in the same LAN, the requirements of the TripleSpace Infrastructure are sensibly reduced (see figure 6.3.1). Sophisticated mechanism for providing remote access (through HTTP for instance) or security and trust will not be necessary in this local configuration.

The **Transport** module of the wrapper of each component will be able to access the Space through simple **APIs**. These APIs will implement the basic operations similar to those defined in tuple-coordination

mechanisms such as *Linda*, *Java Spaces* etc. The **Triple Manager** will coordinate all the 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 **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.

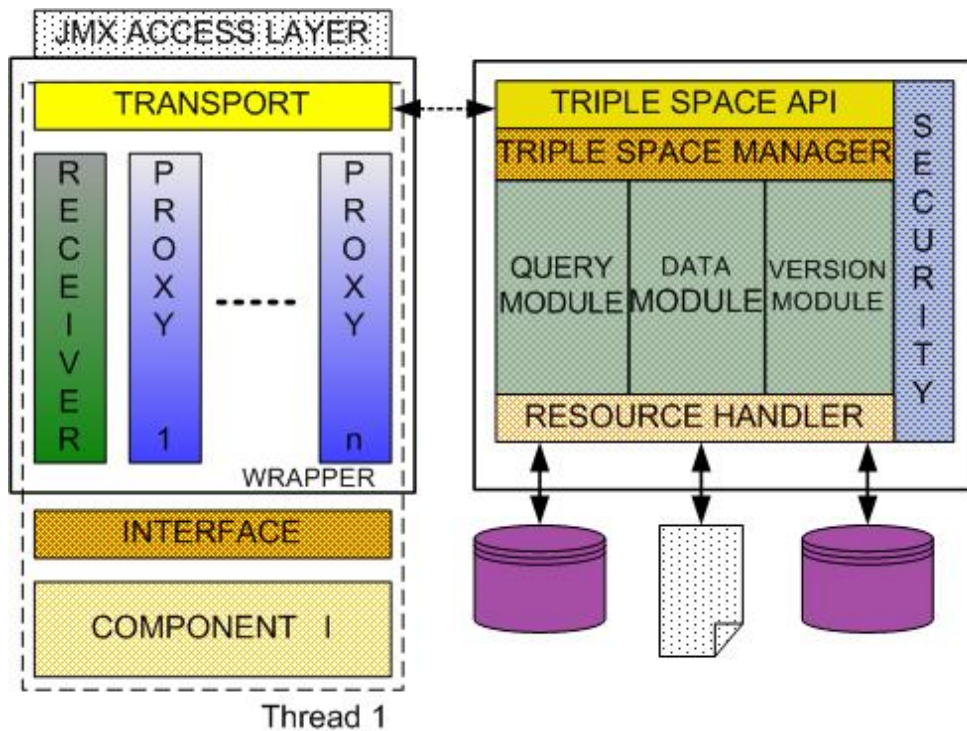


Figure 7: Interaction of a local component with the triple space

6.4 TripleSpace in a distributed WSMX configuration.

To be added and discuss and include a proposal for a distributed WSMX configuration).

7. Interaction between WSMXs through TripleSpace

The communication model used in 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, the immediate responses are usually not available. The reason of such high response latency being network congestion, slow processing, third party invocation, etc. In such situation, the synchronous communication will be costly as it forces the system (component) to remain idle until the response is available. In order to reduce unnecessary overhead imposed by synchronous communication TripleSpace can be used as a communication channel between WSMXs (Cf. Figure 8). The TripleSpace supports purely asynchronous communication thereby optimizing communication performance between parties involved in an interaction. 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 TripleSpace has added advantage for resolving communication disputes between interacting parties.

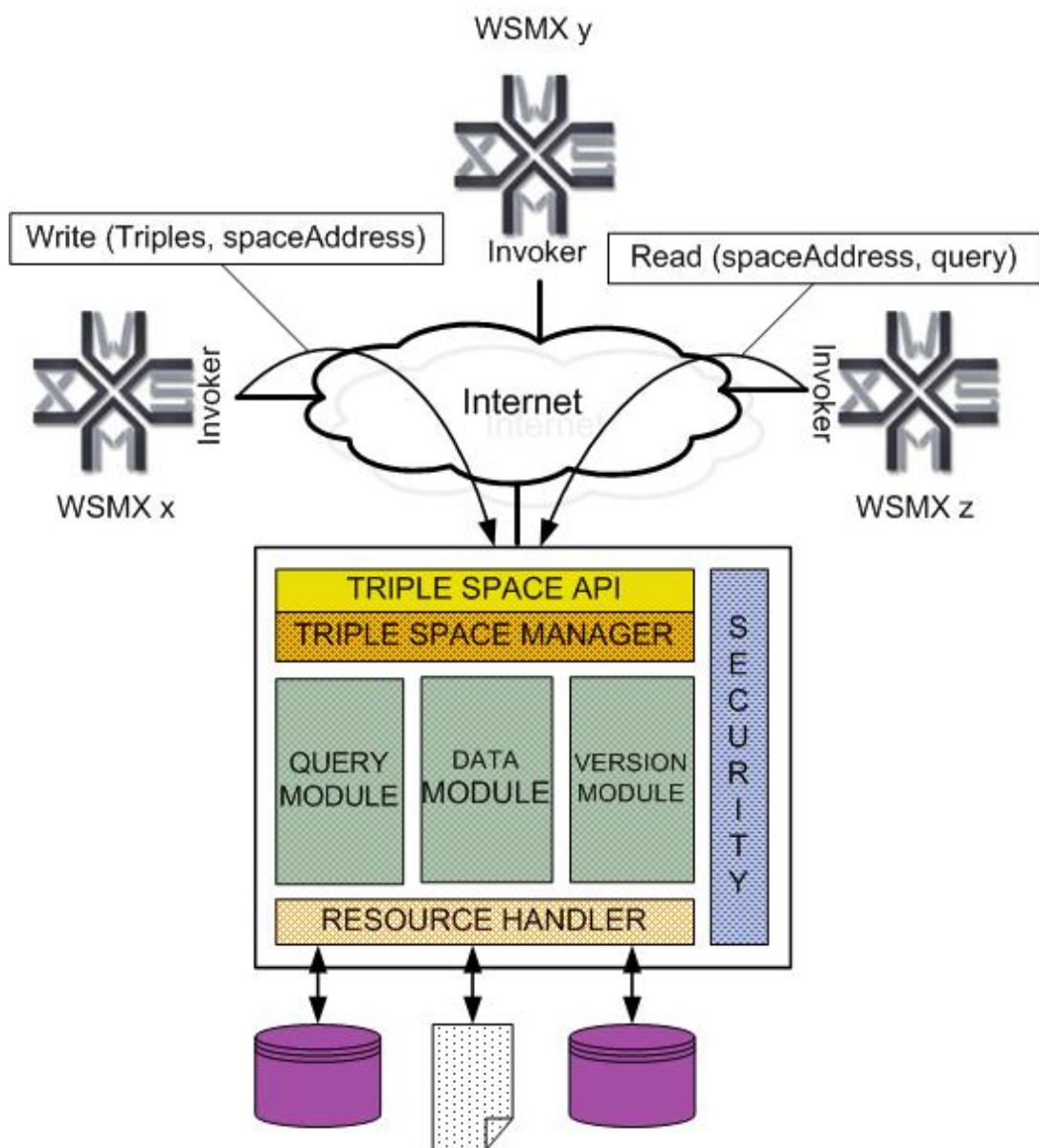


Figure 8: Communication between WSMXs through TripleSpace

Before going into detail how WSMXs can communicate through TripleSpace, a simple logical sequence of basic communication between a WSMX and a TripleSpace is described below.

1. A sending (or receiving) WSMX performs a write (or read) request to TripleSpace through Triple Space API.
2. After receiving the write (or read) request, TripleSpace executes a write (or read) operation if and only if the sending (or receiving) WSMX is genuine. In other cases, the TripleSpace 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 TripleSpace responds the sending (or receiving) WSMX with a positive acknowledgement (or result set). If the write operation fails, the TripleSpace responds 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 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 TripleSpace. The signature of the write operation is the following:

```
write { <Triples>, <spaceAddress> }
```

The parameters **Triples** and **spaceAddress** specify the data to be stored in the TripleSpace located at *spaceAddress*. The Triples *Notation 3* [N3] triples and spaceAddress is a valid URL. The example Triples of a goal description for buying a book is shown below:

```
@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 .
```

```
wsml:buyBookGoal wsml:nfp _:p1 .
```

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

```
wsml:buyBookGoal wsml:nfp _:p2 .
```

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

```
wsml:buyBookGoal wsml:nfp _:p3 .
```

```
_:p3 dc:description "Purchase Order of Book"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p4 .
```

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

```
wsml:buyBookGoal wsml:nfp _:p5 .
```

```
_:p5 dc:contributor "Brahmananda Sapkota"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p6 .
```

```
_:p6 dc:date "2005-05-10"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p7 .
```

```
_:p7 dc:type "domain ontology"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p8 .
```

```
_:p8 dc:format "text"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p9 .
```

```
_:p9 dc:language "en-us"^^xsd:string .
```

```
wsml:buyBookGoal wsml:nfp _:p10 .
```

```
_:p10 dc:rights <http://www.deri.org/privacy.html> .
```

```
wsml:buyBookGoal wsml:nfp _:p11 .
```

```
_:p11 dc:version "1.3"^^xsd:string .
```

```

wsml:buyBookGoal wsml:usedMediators _:m1 .
_:m1 wsml:useMediator <http://www.wsmo.org/med1> .
_:m1 wsml:useMediator <http://www.wsmo.org/med2> .
_:m1 wsml:useMediator <http://www.wsmo.org/med3> .
_:m1 wsml:useMediator <http://www.wsmo.org/med4> .

```

```

wsml:buyBookGoal wsml:hasPostcondition _:x .
_:x wsml:hasAxiom <http://www.wsmo.org/goals/buyBookGoal#address> .
_:x wsml:hasLogicalExpression
"full_address:address[address_number->number:address_number[name->12],
street->street_name:street[name->Street 1], city->city:location[name->Galway]]"^^xsd:string .

```

```

wsml:buyBookGoal wsml:hasPostcondition _:y .
_:y wsml:hasAxiom <http://www.wsmo.org/goals/buyBookGoal#product> .
_:y wsml:hasLogicalExpression "product_data:Book[Book->Book_name:Book[name->Book, Harry
Potter#3], price->price:price[name->Cheap]]"^^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 .

```

read: the read operation gets a set of triples from the TripleSpace. The signature of the read operation is the following:

```
<Triples> read { <spaceAddress>, <query> }
```

The parameters **spaceAddress** and **query** specify the location of the TripleSpace from where to get the data as specified in the *query*. A example query that can be executed in read operation is the following:

```

<> q| where {
    wsml : buyBookGoal rdf : type wsml : goal .
    ?x ?p ?o .
} .

```

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

Handling Context Information

to be written

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 TripleSpace functionalities that will be used to support asynchronous communication between WSMXs.

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.
- [Foster et al., 2001] I. Foster, C. Kesselman, and S. Tuecke: The anatomy of the grid, International Journal of Supercomputing Applications, 2001.
- [Gelernter, 1985], D. Gelernter, N. Carriero, S. Chang: Parallel Programming in Linda, Proceedings of the International Conference on Parallel Processing, 1985.
- [Gelernter, 1992], D. Gelernter: 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/>
- [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.
- [N3], <http://www.w3.org/DesignIssues/Notation3.html>
- [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], E. Oren, M. Moran and M. Zaremba: 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.

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.

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 an 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, 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.

```
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 Tuplespace.

read(templateEntry):

return a matching entry from the Tuplespace, 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 Tuplespace. 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 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.