TripCom

Triple Space Communication

FP6 – 027324

Deliverable

D3.1

Specification and Implementation of a semantic Linda model

Lyndon Nixon, Elena Simperl, Reto Krummenacher, Francisco Martin-Recuerda, Vassil Momtchev, Martin Murth, Geri Joskowicz, eva Kuhn

April 10, 2007

Executive Summary

This deliverable specifies the co-ordination model and language for Triple Space. In order to achieve a new Web scale communication solution for Semantic Web services, Triple Space applies the tuplespace paradigm to the communication between services. The use of the tuplespace paradigm for the exchange of knowledge is a radically different situation than that of classical Linda systems. In order to ensure that the co-ordination model and language takes into account the interaction patterns of Semantic Web Services and the meaning of co-ordination that arises from co-ordinating knowledge rather than plain data, we assess the changes and extensions that are necessary to Linda, the original co-ordination language of tuplespace.

As a result we specify a semantic Linda as a co-ordination model and language for semantic tuplespace systems.

Document Information

IST Project Number	FP6 – 027324	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	`http://www.tripcom.org/`		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	3.1	**Title**	Specification and Implementation of a semantic Linda model
Work Package	**Number**	3	**Title**	Triple Space Interaction

Date of Delivery	**Contractual**	M12	**Actual**	31-Mar-07
Status	version 1.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Lyndon Nixon (FU Berlin), Elena Simperl (FU Berlin), Reto Krummenacher (LFUI), Francisco Martin-Recuerda (LFUI), Vassil Momtchev (Ontotext), Martin Murth (TU Vienna), Geri Joskowicz (TU Vienna), eva Kuhn (TU Vienna)			
Resp. Author	Lyndon Nixon		**E-mail**	nixon@inf.fu-berlin.de
	Partner	FU Berlin	**Phone**	+49 (30) 838-75225

Abstract (for dissemination)	Triple Space will be a new type of tuplespace-based platform for the co-ordination of services exchanging semantic information. The Linda co-ordination language was designed for the co-ordination of tuples containing plain data. Knowledge contained in tuples imposes new semantics upon the co-ordination primitives of Linda. This deliverable assesses which changes and extensions are necessary in Linda to support semantic service co-ordination. The result will be a semantic Linda, a co-ordination model for Triple Space.
Keywords	Linda, tuplespace, Semantic Web, RDF

Version Log

Issue Date	Rev No.	Author	Change
13-06-06	1	Lyndon Nixon	First draft structure
25-07-06	2	Lyndon Nixon	Revision following 21 July telephone conference discussion
16-08-06	3	Lyndon Nixon	Text on Linda extensions and recommendations converted to LaTeX
21-08-06	4	Martin Murth	First Draft: Text on Interaction Patterns
24-08-06	5	Lyndon Nixon	First Draft: Recommendations on Linda Extensions
04-09-06	6	Lyndon Nixon, Reto Krummenacher	Final Draft: Recommendations on Linda Extensions
11-09-06	7	Martin Murth	Final Draft: Text on Interaction Patterns
25-09-06	8	Lyndon Nixon	First Draft: Requirements from other workpackages
26-09-06	9	Lyndon Nixon	Final Draft: Requirements from other workpackages
28-09-06	10	Lyndon Nixon	First Draft: Triple Space API
11-10-06	11	Lyndon Nixon	Final Draft: Initial Triple Space API as specified in Task 3.1
15-02-07	12	Lyndon Nixon, Vassil Momtchev, Elena Simperl, Martin Murth, Reto Krummenacher	Integration of prototype implementation plan into Prototype chapter
27-02-07	13	Lyndon Nixon	Revisions in Prototype chapter
01-03-07	14	Vassil Momtchev	Completed Prototype chapter
02-03-07	15	Lyndon Nixon	Full revision of deliverable, added Conclusion
15-03-07	16	Lyndon Nixon	Prefinal version following QA
16-03-07	17	Lyndon Nixon, Reto Krummenacher, Martin Murth	Submission version to QAC
30-03-07	18	Lyndon Nixon	Prefinal version following QAC

Project Consortium Information

Acronym	Partner	Contact
Leopold Franzens University Innsbruck `http://www.deri.at`	LFUI	Prof. Dr. Dieter Fensel Digital Enterprise Research Institute (DERI) Innsbruck, Austria E-mail: dieter.fensel@deri.org
National University of Ireland, Galway `http://www.deri.ie`	NUIG	Dr. Laurentiu Vasiliu Digital Enterprise Research Institute (DERI) Galway, Ireland Email: laurentiu.vasiliu@deri.org
University of Stuttgart `http://www.iaas.uni-stuttgart.de/`	USTUTT	Prof.Dr. Frank Leymann Inst. für Architektur von Anwendungssystemen (IAAS) Stuttgart, Germany E-mail: frank.leymann@informatik.uni-stuttgart.de
Vienna university of Technology `http://www.complang.tuwien.ac.at/`	TUW	Prof.Dr. eva Kühn Institut für Computersprachen Vienna, Austria E-mail: eva@complang.tuwien.ac.at
Free University Berlin `http://www.ag-nbi.de/`	FUB	Prof. Dr.-Ing. Robert Tolksdorf AG Netzbasierte Informationssysteme Berlin, Germany E-mail : tolk@inf.fu-berlin.de
Ontotext Lab, Sirma Group Corp. `http://www.ontotext.com/`	ONTO	Atanas Kiryakov, Vassil Momtchev, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: vassil.momtchev@ontotext.com
Profium OY `http://www.profium.com/`	Profium	Dr. Janne Saarela Profium OY Espoo, Finnland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. `http://www.cefriel.it/`	CEFRIEL	Davide Cerri CEFRIEL SCRL. Milano, Italy E-mail: cerri@cefriel.it
Telefonica I+D `http://www.tid.es/`	TID	Noelia Pérez Crespo Telefonica I+D Madrid, España E-mail: npc@tid.es

Table of Contents

List of Abbreviations

API	Application Programming Interface
DAM	Digital Asset Management
RDF	Resource Description Framework
RDMS	Relational Database Management System
SPARQL	SPARQL Protocol and RDF Query Language
TripCom	Triple Space Communication
TSC	Triple Space Computing
TS API	Triple Space API
WP	Work Package
WS	Web Service
WSDL	Web Service Description Language

1 Introduction

This deliverable specifies the co-ordination model and language for Triple Space.

In order to achieve a new Web scale communication solution for Semantic Web services, Triple Space applies the tuplespace paradigm to the communication between services. The use of the tuplespace paradigm for the exchange of knowledge is a radically different situation than that of classical Linda systems. In order to ensure that the co-ordination model and language takes into account the interaction patterns of Semantic Web Services and the meaning of co-ordination that arises from co-ordinating knowledge rather than plain data, we assess the changes and extensions that are necessary to Linda, the original co-ordination language of tuplespace. As a result we specify a semantic Linda as a co-ordination model and language for semantic tuplespace systems.

In order to achieve this, we consider extensions to classical Linda implementations in Chapter 2. As our interest is in collecting necessary extensions to Linda to be supported by the Triple Space, we consider in particular the APIs of a number of proposed semantically enabled tuplespace systems in Chapter 3.

We combine this overview of Linda extensions with requirements taken from the other TripCom workpackages in Chapter 4.

This has enabled us to specify a Semantic Linda API for Triple Spaces, following agreement on a range of open issues, in Chapter 5 and to validate the coordination model and language by comparing its expressiveness to core interaction patterns in Chapter 6 and implementing a first prototype, which is described in Chapter 7.

As an important basis for the ongoing work in TripCom, Chapter 8 concludes with an assessment of the work done and future plans for its development and use within the project.

2 Overview of Linda extensions

2.1 Introduction

The first step is to collect and provide an overview of extensions made to the Linda co-ordination language. The original Linda [24] foresaw a simple set of four primitives:

- *out* add a tuple to the tuplespace
- *in* destructively remove a tuple from the tuplespace
- *rd* non-destructively remove a tuple from the tuplespace
- *eval* start a new process

The eval primitive gradually fell out of use as its semantics were unclear and its use found to be often unnecessary. The basic remaining Linda operations of out, in and rd form the basis for any tuplespace implementation. However these too have proven insufficient, and implementations of tuplespace platforms based on Linda have liberally extended the coordination language for their needs. In this chapter we will look at which types of extensions have been made and to what purpose they can be used in tuplespace implementations.

Initial observations about Linda extensions were gleaned from [32], particularly the first chapter [3].

2.2 The importance of a good coordination model

A coordination model is a framework for the synchronisation of communication among concurrent processes. One wider definition of coordination models is [33]:

"to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems"

A coordination language is a linguistic embodiment of a coordination model [25]. In other words, it defines a set of operations with given parameters and possibly an associated defined semantics which reflect what response should be expected from a certain request. Linda as a coordination language stands out as it is defined to be general. It was specified deliberately to be very simple (three core operations on the tuplespace plus eval). This approach to simplicity in a computer language has been termed "conceptual economy" [26].

However, when applying a Linda-based coordination model in real scenarios, and in open distributed systems, the core Linda principles may be maintained yet also built upon by further operations (or extensions of given operations and/or their semantics) to support additional requirements. One aspect to consider in extending the Linda coordination language is whether the extension adds to the expressiveness of the coordination model, e.g. many operations can already be expressed as a combination of existing operations. Another aspect to consider is how the change in the coordination language impacts on the complexity of the implementation of that language in a tuplespace system, e.g. operations on the global tuplespace could require freezing the entire tuplespace until all answers have been found.

We consider extensions to the Linda coordination language under a number of categories and consider to what extent they add to the expressiveness of the model and to the complexity of an implementation.

2.3 Observations on the core Linda primitives

Firstly we return to the core Linda primitives themselves to consider what base expressiveness they provide to a coordination language. Linda itself has been demonstrated to be sufficient for writing parallel applications [10], designing distributed computing platforms e.g. [46]; [23] and for agent-based systems [14].

We already noted that the *eval* primitive has largely not been taken up in Linda implementations.

The *rd* primitive appears conceptually equivalent to an *in* followed by an *out* of the same tuple. The only difference is that in the latter case there will be a moment, determined by latency, in which the tuple does not exist in the space while in the former the tuple is always present in the space as only a copy is returned to the client. Hence in an extended Linda model, with a test-for-absence primitive or simultaneous reads (such as in Bonita [35]), the *rd* primitive does ensure different interactions in the space and hence is justifiably retained in the language.

Likewise there is no update operation in Linda, rather a client can *in* a tuple, make some change to it and *out* it to the space again. As noted above, this is not exactly an update as there will be some space between the removal of the original data and the appearance of the updated data.

The *out* primitive can have three different semantics [5]. Either the insertion of the tuple into the space is considered to be instantaneous, or the insertion of tuples is considered to occur in the same order as the execution of the out primitives by the clients (ordered) or the order of insertion of tuples may differ from that of the execution of the out primitives (unordered). The last two options consider the latency that might occur in a distributed system, hence would be the two options for Triple Space. Ordered insertion would require that the client waits for an acknowledgement from the space that the tuple has been inserted (note that without an acknowledgement the client can offer no guarantee that the tuple is actually present in the space). Without waiting for acknowledgement, the standard semantic of out will be unordered.

2.4 Overview of extensions in Linda

We note that extensions to the coordination language can take different forms:

- Extension of the coordination primitives, i.e. the specification of a new operation
- Extension of the coordination semantics, i.e. the redefinition of the meaning of an operation
- Extension of the coordination model, i.e. providing other instruments for expressing coordination such as programmability

In this overview we have split the extensions into various categories and for each, we identify the primitives in extended coordination languages used by tuplespace implementations and summarize with a table the identifiable pros and cons of each extension category.

The categorization and evaluation of the extensions are based on the analysis of literature in the Linda and coordination community, in which the various extensions are introduced and compared. In the referenced papers, we note the stated advantages and disadvantages of the operation given by the authors. In addition, we have added our own observations on these primitives, focused on the particular issues surrounding Triple Space (e.g. the tuple model being Semantic Web data).

2.4.1 Transaction operations

A transaction operation has the ability to act atomically on a group of data. Transactions are a common part of RDMSes in order to enable concurrency control. Systems that implement transactions need means to create a transaction, begin it and finally commit or abort it.

In JavaSpaces [23], transactions are handled by a transaction manager. This can be based on an external transaction service, which is encapsulated in a Transaction object and passed as a parameter in JavaSpace operations e.g. write, read and take (equivalent to out, rd and in). Other tuplespace platforms such as MARS [8] also add a Transaction object to the parameters of the Linda operations, using a system-external implementation of transactionality such as the Java Transaction API (JTA). .

In TSpaces [46], a new operation *multiwrite* permits the insertion in the tuplespace atomically of all the data contained in an array. Transactions are implemented within the system rather than handled externally and passed by parameter. Hence while the client sees an atomic operation, on the system side each tuple is written separately within a transaction.

Pros	**Cons**
Introduces ACID properties to a tuplespace ACID has become a vital part of RDBS	Introduces complexity to the tuplespace implementation (transaction management)
Transactions play an important role in Web service communication and hence can be expected to be required in Triple Space	How long can a transaction take (and tuples locked)?
	How to deal with deadlocks?
	Difficulty to implement in real time, distributed and fault tolerant systems.

In Persistent Linda [1], a guarantee is made that concurrent transactions will have the same effect as sequentially committed transactions while aborted transactions have no effect. The assumption of fail-stop failures is that a failure in a transaction automatically triggers a stop, which can not be made in distributed systems where network failures may stop a transaction process, hence a slow-down failure approach is considered where transaction processes that slow down (it is assumed they may speed up again) are supported by new processes which complete the transaction [38]. As a result, a process might fail without the transaction failing.

In CORSO [19], transactions are implemented in the form of optimistic transactions which means that a commit is no longer guaranteed to successfully make all changes. Rather a callback indicates if the commit was successful or has failed. This breaks the principle of ACIDity. However, this reduces overhead in managing transactions as the system now does not have to freeze the tuplespace until the new state (after the commit) is reached.

2.4.2 Global view operations

A global view operation is an operation which classically requires that it is possible to know the actual state of the entire dataspace, such as test for absence or a count of all matching tuples. Given that this is hardly possible in distributed systems, "best as" solutions are generally implemented which do not guarantee complete results.

TSpaces includes the *count* operation, which returns an integer which is the number of tuples in the tuplespace which match a given template.

Another extension is non-blocking variants of the *in* and *rd* operations, usually named *inp* and *rdp*. While acting like their blocking versions when a matching tuple is found, they also return a fail notification if no matching tuple is found in the entire tuplespace [4]. In JavaSpaces they are named takeIfExists and readIfExists.

A variant of the non-blocking operations is operations with a specified timeout, such as in Jada [13], JavaSpaces and MARS.

Test for absence operations return true if a matching tuple is not found in the space, in other words act as the logical opposite of the non-blocking retrieval operations.

Pros	**Cons**
Makes Linda more deterministic (we can know that no match is present in the space at some specific timepoint)	Difficult to guarantee correctness in a distributed system (Before the operation completes, the state of the tuplespace can change)
Extends expressiveness of the coordination model non-blocking read is supposed to be more expressive than notification	Requires freezing the entire tuplespace until operation completes to ensure correctness
There may be scenarios in which global operations are necessary	Need to decide what the bounds of the global space are for a client (e.g. access for count but not read?)
	If Triple Space is considered to use the Open World Assumption, global operations may not make any sense

2.4.3 Multiple read

Both transactionality and a global view can be combined in a single operation which reads or consumes a multiset of data from the global dataspace. collect and copy-collect [37] are proposals for solving the Linda multiple read problem: that two sequential read operations could return the same tuple. MARS-X [9] has the equivalent operations readAll and takeAll. Collect is used in Bonita [35] and WCL [36].

Jada extends this even further with the getAll operation which returns all tuples that match all of a set of templates, and getAny which returns all tuples that match any of a set of templates.

Pros	Cons
Solves Linda multiple read problem	Typical approach is to isolate answers until completion by copying them into a protected dataspace

2.4.4 Retrieval operations

Classical retrieval in Linda is the *rd* primitive. In Bonita, the retrieval operation is separated from the check if a matching tuple has been returned. This means that retrieval can be done in parallel and the sychronisation takes place in the subsequent check. The corresponding operations are dispatch which is non-blocking and obtain which is blocking. The use of these primitives has been illustrated thus:

in(X)
in(Y)
in(Z)
	id1 = dispatch(ts,X,d)
	id2 = dispatch(ts,Y,d)
	id3 = dispatch(ts,Z,d)
	
	obtain(id1)
	obtain(id2)
	obtain(id3)

While in Linda (on the left) each *in* must complete before the next operation can be handled, in Bonita all three retrievals can occur in parallel and only the confirmation of their resolution blocks. In the Bonita literature, this finer grained retrieval has demonstrated performance gains.

In Persistent Linda, an operation for updates is introduced as the combination of a retrieval operation with *out* into one operation and the addition alongside the template of a code block which defines the update to be made to a matched tuple, e.g. to remove a tuple matching the pattern ("example",?integer) from the space and emit a new tuple into the space with the integer value of the second field incremented by one, one could write:

```
in-out(example,?i){i++;}
```

In TSpaces a novel rendezvous operation is supported: *rhonda*. It takes a tuple and a template as parameters and swaps them with a template and a tuple from a rhonda operation executed by another process. For example, if a process executes rhonda(⟨buy,part123,500,2.73⟩,⟨confirm,?PurchaseID⟩) and another process executes rhonda(⟨confirm,AB8874⟩,⟨buy,?PartID,?quantity,?price⟩) the latter process will receive the purchase request and the former process receives a confirmation. This example illustrates the usefulness of rhonda for atomic synchronisation of process communication.

Pros	Cons
Bonita allows parallel retrieval without losing Lindas synchronisation capabilities reportedly with a resulting improvement in performance	
Update operations allow for atomic modification of tuples in the space	
Rhonda is useful for atomic synchronisation, e.g. automatic request acknowledgements	

2.4.5 Meta operations

Another possibility is to define operations which operate not on the data structure in the tuples but metadata associated to the tuples. This includes tuple ids, tuple types and timestamps (when a tuple was generated in the space).

CORSO has a test primitive which tests an object (tuple) against its timestamp. This acts as a cheaper version of rd as the system does not need to examine the data in the object.

Another operation is ID passing. In CORSO a process can only access an object by reference, hence apart from being the creator of the object or knowing some superobject of the given object processes require a means to have references passed to them from other processes.

Pros	Cons
Typically cheaper than operating on the (more complex) tuple content	Retrieval by ID goes against Linda principles (associative addressing)

2.4.6 Event reactivity

Both JavaSpaces and TSpaces allow for the specification of listeners which observe events in the dataspace and react to particular types of event in particular ways.

In JavaSpaces, the operation notify [6] waits on events where data is inserted into the tuplespace that matches a particular template and notifies the client which requested the notification.

LIME [31] has an operation reactsTo(s,p) where s is a block of code that is executed when a tuple is found in the dataspace that matches template p.

Pros	**Cons**
Adds additional interaction pattern to the core Linda coordination model	A notification request has an unclear semantics with respect to its persistence
Makes the unordered semantics of out as expressive as the ordered semantics	It has been argued that Linda with non-blocking read is more expressive than Linda with notification [6]
Saves clients sending repeated retrieval requests to the space	

It should be noted that [6] acknowledges that while in, out, inp is strictly more expressive than in, out, notify the difficulty of inp implementation in distributed versions of Linda makes the use of notification potentially a good compromise between easiness of implementation and expressive power.

2.4.7 Added programmability

Coordination languages such as TuCSoN [15], Law-Governed Linda [29] and MARS introduce programmability of the coordination model.

One advantage of programmability is to allow for secure communication in the tuplespace, where communicating clients are transient and potentially unidentified. For example, in Law Governed Linda [30] controllers exist between each client and the tuplespace and operate according to a law which specifies how communication should take place (i.e. specifies which operations can be permitted with data matching some given templates).

In TuCSoN there are two levels of perception of the space: how the client sees the space and how the system itself interacts with the space. Hence a coordination primitive reaction(Operation,Body) is used to map from the Operation that is the clients perspective to one or more system operations (the Body).

LuCe [18] uses tuples not only to contain the data to be shared but also to describe tuplespace behaviour such tuples are termed specification tuples. Clients can use the Linda operations to interact normally with the tuplespace but can also specify the behaviour of the tuplespace using the ReSpecT language [17], a logic-based language which allows the specification of reactions in response to interactions in the dataspace. Reactions are sequences of reaction goals which can check properties of the interaction event, perform simple tests on terms and alter the space through the primitives out_r, rd_r, in_r and no_r (a test for absence succeeds if no tuple matching the given template is found). This also allows transactionality as a reaction as a whole succeeds only if all its reaction goals succeed. As a result, the effect of a single interaction is no longer limited to a single insertion, reading or removing of a tuple but can be made as complex as desired by specifying reactions within the tuplespace.

Other coordination languages such as Gamma [2] introduce the chemical reaction metaphor. In this metaphor, tuples are seen as molecules which move in a chemical solution and which react in a certain way when they come into contact with one another. In the coordination language, conditional rewriting rules are defined in the form (R,A), where R is a reaction condition (a Boolean over a data multiset) and A is a rewriting action (a function from a multiset to another multiset).

Pros	**Cons**
Introduces dynamic extensibility to the coordination model	Greater complexity in the tuplespace implementation to include program engines
Can be a means to implement access control features	Security issues in executing external code
	Need to define a programmatic API for the space

3 Analysis of Linda extensions

3.1 Introduction

A number of tuplespace platforms for the coordination of semantic information have been presented in the literature in recent years. These platforms make some initial decisions regarding an API for tuplespace access which could be relevant for the Triple Space. An overview of these platforms and their characteristics can be found in [20]. In this section we review these APIs in the context of making a decision for Triple Space. We draw upon experiences made in these activities to help us identify which coordination language primitives prove relevant or not to a Linda-based communication platform for Semantic Web data.

3.2 Triple Space Computing (TSC)

Triple Space Computing is based on the ideas of [21] which saw tuplespace computing as a potential solution to current state of communication in Web Services.

API call	**Description**
write (URI ts, Transaction tx, Graph g): void	Write one or more triples in a concrete Triple Space identified by a URI.
take (URI ts, Transaction tx, Template t): NamedGraph	Returns a named graph (or nothing) that matches the template (that can be expressed using a formal query language) and deletes the matched named graph from the concrete Triple Space that is identified by the URI ts.
take (URI ts, Transaction tx, URI ng): NamedGraph	Returns the named graph (or nothing) that is identified by the URI ng and deletes the named graph from the concrete Triple Space that is identified by the URI ts.
waitToTake (URI ts, Transaction tx, Template t, long timeout): NamedGraph	Like take but the process is blocked until a matching graph is detected or the given timeout expires.
read (URI ts, Transaction tx, Template t): NamedGraph	Like take but non destructive.
read (URI ts, Transaction tx, URI ng): NamedGraph	Like take but non destructive.
waitToRead (URI ts, Transaction tx, Template t, long timeout): NamedGraph	Like read but the process is blocked until a matching graph is detected or the given timeout expires.

query (URI ts, Transaction tx, Template t) : Graph	This is a query over all triples in a space and independent of the written graphs; it returns a number of matching triples and returns them as a newly created RDF graph.
waitToQuery (URI ts, Transaction tx, Template t, long timeout) : Graph	Like query but blocking.
count (URI ts, Transaction tx, Template t) : long	Return the number of triples that match template t.
update (URI ts, Transaction tx, NamedGraph ng) : boolean	Alters a given named graph if it exists or throws an exception otherwise. Update is internally handled as a transactional take and write however these operations are hidden from external users to avoid the creation of named graph identifiers outside the space.
subscribe (URI ts, Template t, Callback c) : URI	A consumer (agent) expresses its interest in triples that match with template t in a concrete Triple Space. Any time that there is an update in the Triple Space, the subscriber receives a notification via callback interface that there are triples available that match the template. The operation returns an URI that identifies the subscription.
unsubscribe (URI ts, URI subscription) : boolean	A consumer (agent) deletes its subscription, and no more related notifications are received. The operation returns a set of URIs of subscriptions deleted
advertise (URI ts, Template t) : URI	A producer shows its intention to provide triples that match t. Advertisements provide information to the system that can be used in advance to improve the distribution criteria of data and participants. The operation returns an URI that identifies the advertisement created.
unadvertise (URI ts, URI advertisment) : boolean	A producer shows its intention to do not provide more triples that match t.
createTransaction () : Transaction	Ask the TSC infrastructure to create a new transaction and to start it; the operation returns the transaction id.

commitTransaction (Transaction tx) : boolean	Make permanent a set of changes defined inside of a transaction txn.
abortTransaction (Transaction tx) : boolean	Undo a set of changes defined inside of a transaction tx.
createSpace (URI ts) : void	Creates locally a new Triple Space or joins a given Triple Space. This is a management operation to configure a local TS Kernel.
leaveSpace (URI ts) : void	Enables a TS Kernel to leave a Triple Space, i.e., the kernel stops to replicate data for the given space ts.

In the continuation of this section the various TSC interaction operations are discussed and explained in more detail. TSC is an Austrian-funded small FIT-IT project with the goal to develop a first prototype for Triple Space in the area of Semantic Web services [22].

As the minimal data unit in TSC is envisioned to be an RDF graph the signature of the write, as well as of all other operations supports Graph objects. This is however only a generalization and not a restriction, as nothing prevents a publisher from writing RDF graphs containing only single triples. It is however argued that for many use cases the relevant information consists of more than one triple and that handling those triples jointly makes life much easier for the user, as well as for the space management components.

In direct consequence the read and take (in Linda rd and in) operations return whole graphs; in fact rather so-called Named Graphs [11]. A Named Graph is a URI + RDF Graph pair: ng = (uri,graph). The Named Graphs are created by the space implementation upon a write operation and space users (neither producers nor consumers) have no influence on the creation of the graph identifiers. This has two particular reasons: (1) the URIs are created by a trusted entity (the TS Kernel) and invalid URIs are not created, and (2) URI should only be parsed by the entity that actually defined it, and hence we have no problems in case that the URI contains information that is not purely related to identification of resources. Moreover, the authors argue that it is not foreseen that users should get the right to identify single chunks of information.

The reason why the read operations are limited to the retrieval of whole Named Graphs, i.e. to the chunks of triples that were written at once, is the fact that the take and read operation should have the same semantics (besides being either destructive or not). Major problems might arise if a user can take (remove) parts of a graph that a producer defined to be a single unit. As an example one could imagine an order or a service description that only makes sense when all triples are interpreted together; if however some triples are removed this would alter the information content. Hence, only the removal of whole graphs is allowed, and it is in the responsibility of the producer to modularize its information.

The same time one needs to pay tribute to the big advantages that the Semantic Web provides: seamless merging of graphs, nesting and interlinking of information, inference of knowledge and so on. This strengthes of the Semantic Web that shall be taken over to space-based computing are obviously largely limited when only manipulating whole objects. The TSC project therefore agreed on the compromise to introduce a query operation. A query call to the space searches for matching triples in the space without the restriction of having to belong to the same Named Graph.

A last remark to the fundamental read and take operations: TSC not only proposes the template-based operations as they are known from Linda-like systems, but also a Web-like interaction pattern that makes use of the graph names to identify information. Hence, it is not only possible to find relevant chunks of information by template but also by URI. This has several advantages: (1) a given chunk of information can be communicated at once by the reference of the corresponding URI, (2) it allows fulfilling the initial wish for identifiers per triple expressed in [7], and (3) the URI provides directly an index for the available information, which allows for optimization of triple storage and access.

Additionally to the already discussed access patterns via read, take and query the TSC project specified blocking versions. The idea is, in accordance to for example read and readIfExists in JavaSpaces, or read and waitToRead in TSpaces, to allow a consumer to specify if the space infrastructure shall return with no results or wait until there are some matching RDF triples available, i.e. a particular Named Graph that matches the template.

The next operation in Table 3.1 is count. This method was also inherited from the TSpaces API. In fact there is no way to reasonably support this method in open and dynamic environments and the authors of TSC actually recommend refraining from it. In fact, first of all the implementation needs to ensure that the whole virtual space is considered when determining the number of matches and secondly the semantics is very unclear with respect to timing issues. Triple Spaces are eventually about the concurrent access by different (Semantic) Web services and users and the state of the space changes thus constantly.

Initially in the TSC project it was argued that the World Wide Web and hence also Triple Spaces should be an ever-growing source of data where delete operations are not desired. This is however for many obvious reasons far from reality. The same counts for the desire to update information. TSC therefore introduced the update operation, which basically is a transactional take-write operation. In other words, the update operation removes the chosen Named Graph from the space and feeds the newly defined RDF triples back to the space with the same URI, as the removed one had. This two operations need to be transactionally secured to ensure that the space does not contain contradicting entries with the same URI. An example with omitted transactions shall be given to support the cause:

```
URI myFOAFuri = p1.write(<tsc://myspace.example.org>, myFOAF_2006-06-13);
//myFOAFuri = <tsc://myspace.example.org/123abc>

NamedGraph hisFOAF =
p2.read(<tsc://myspace.example.org>,<tsc://myspace.example.org/123abc>);
//hisFOAF = (<tsc://myspace.example.org/123abc>, myFOAF_2006-06-13);


NamedGraph ng =
new NamedGraph(<tsc://myspace.example.org/123abc>, myFOAF_2006-06-14);
p1.update(<tsc://myspace.example.org>,ng);

NamedGraph hisFOAFupdate =
p2.read(<tsc://myspace.example.org>,<tsc://myspace.example.org/123abc>);
```

Blocking retrieval operations can have some unfortunate implications, certainly in open and dynamic environments, as the time for data discovery is often indeterministic and non-blocking operations can last longer than blocking operations. Another argument against the blocking retrieval is the intended support for the publish/subscribe paradigm in order to ensure clear flow decoupling too. Such a publish/subscribe installation could successfully replace the non-blocking operations without changing the execution semantics. These ideas were adopted from the CSpaces proposal (Section 3.3; it is however not ensured that they will be integrated at all in the first TSC prototype resulting from the FIT-IT project. For further information about the publish/subscribe paradigm in Triple Space Computing the reader is referred to the CSpaces paragraph below.

3.3 CSpaces

Conceptual Spaces (CSpaces) [27] was born as an independent initiative to extend Triple Space Computing [21] with more sophisticated features and to study their applicability in different scenarios apart from Web Services.

API call	**Description**
write(set tuples, URI cs_destination, URI cs_origin):void	Write one or more tuples in a concrete CSpace (cs_destination) identified by a URI. If the tuples are defined in terms of a domain theory stored in a different CSpace then it is specify in the third parameter (cs_origin)
take(Template—Query t, URI cs_destination, URI cs_origin):Tuple	Return the first tuple (or nothing) that match with the template or a query expressed in using a formal query language) and delete the matched tuple from a concrete CSpace cs_destination. If the template of query t is defined in terms of a different domain theory than the one stored in cs_destination, then it is specified in the parameter cs_origin
waitToTake(Template—Query t, URI cs_destination, URI cs_origin):Tuple	Like take but the process is blocked until the a tuple is retrieved
read(Template—Query t, URI cs_destination, URI cs_origin):Tuple	Like take but the tuple is not removed

waitToRead(Template—Query t, URI cs_destination, URI cs_origin):Tuple	Like read but the process is blocked until the a tuple is retrieved
scan(Template—Query t, URI cs_destination, URI cs_origin):Set	Like read but returns all tuples that match with template or query t
countN(Template—Query t, URI cs_destination, URI cs_origin):Long	Return the number of tuples that match template or query t
subscribe(URI agent, Template—Query t, Callback c, URI cs_destination, URI cs_origin):URI	An agent (consumer) expresses its interested on tuples that match with template or query t in a concrete CSpace (cs_origin). Like take, if the template of query t is defined in terms of a different domain theory than the one stored in cs_destination, then it is specified in the parameter cs_origin. Any time that there is an update in the CSpace, the subscriber receives a notification that there are tuples available that match the template. The notification is executed by calling a method/routine specified by the subscriber. The operation returns an URI that identifies the subscription.
unsubscribe(URI agent, Template—Query t, Callback c, URI cs_destination, URI cs_origin):Set	An agent (consumer) deletes its subscription, and no more related notifications are received. The operation returns a set of URIs of subscriptions deleted
advertise(URI agent, Template—Query t, URI cs_destination, URI cs_origin):URI	An agent (producer) 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. The operation returns an URI that identifies the advertisement created.
unadvertise(URI agent, Template—Query t, URI cs_destination, URI cs_origin):Set	An agent (producer) shows its intention to do not provide more tuples that match t. The related advertisements are deleted, and the operation returns a set of URIs deleted.
getTransaction(URI cs):URI	Ask the CSpace infrastructure to create a new transaction and returns its id as a URI.

beginTransaction(URI txn, URI cs):boolean	Identify the beginning of a set of instructions executed under a concrete transaction (identified by a URI). Several processes can execute instructions under the same transaction, and only those processes can see the changes produced in the space before the transaction is committed.
commitTransaction(URI txn, URI cs):boolean	Make permanent a set of changes defined inside of a transaction txn.
rollbackTransaction(URI txn, URI cs):boolean	Undo a set of changes defined inside of a transaction txn.

CSpaces integrates tuplespace and publish-subscribe operations, transaction support and semantic data specification in a new coordination model. The coordination model API for CSpaces is very similar to TSC. Hence in this section we focus only on the differences from TSC in the CSpaces coordination model.

The data model of CSpaces does not commit to a concrete representation formalism, giving to the designers the possibility to use different representation means. The data model requires a specific tuple model which has seven fields. One of the fields stores the data that can be represented by different formalisms (RDF, FOL, OWL etc.). In addition, the CSpaces API also takes into account the organizational model that CSpaces promotes. This organizational model envisions direct acyclic graphs of individual and shared knowledge spaces linked by mapping rules. Thus agents can write information in terms of their logical theories, stored in individual CSpaces or in terms of shared logical theories (stored in shared CSpaces).

In both cases, agents have to specify the logical theory used. This requirement is included in the API specification. As a consequence, middleware infrastructure is aware that data transformation services need to be executed. In real scenarios, in which heterogeneity is present, this capability is very important.- The organizational model of CSpaces leads to a clustered organization specifically valuable for distributed knowledge spaces.

We see that CSpaces has some features which are beyond the scope of TripCom and we can consider the CSpace API as a superset of the TSC API. However CSpaces takes into account issues such as distributed spaces and heterogeneity in its API which may prove relevant in subsequent extensions of Triple Space.

3.4 Semantic Web Spaces

Semantic Web Spaces [41],[40] has been proposed by the Freie Universität Berlin. A conceptual model has been drawn up [39],[42] in which the necessary extensions to the traditional Linda co-ordination model were considered to support a tuplespace exchanging Semantic Web information.

A criticism of Linda has been that the semantics of the co-ordination primitives (in, out, rd) were never formally defined by the creators of Linda. When working with Semantic Web data it is important that the set of co-ordination primitives are clearly defined. In Semantic Web Spaces, two levels of interaction are defined: the data level, where tuples contain data without any formal meaning, and the information level, where RDFTuples are recognized as being special data structures that express formally defined knowledge about concepts. RDFTuples are handled also at two levels: in terms of the abstract syntax and in terms of the formal semantics. These three levels of co-ordination provide an increasing level of expressivity at an increasing cost in computability. The co-ordination primitives of Semantic Web Spaces are listed below.

API call	**Description**
outr(s,p,o):boolean	A Linda out which is only true if tuple is RDFTuple
rdr(s,p,o,id):RDFTuple	A Linda rd which only matches on RDFTuples
inr(s,p,o,id):RDFTuple	A Linda in which only matches on RDFTuples
claim(s,p,o,id):boolean	A Linda out which is only true if the RDFTuple conforms to the RDF Schema which constrains it
claim(sSubspace):boolean	Like claim above but passes multiple RDFTuples as a single subspace
endorse(s,p,o,id):Subspace	A Linda rd which can also read inferred tuples and returns a concise bounded description as Subspace
extract(s,p,o,id):Context	Multiple read version of endorse - finds all matching RDFTuples and places them into a context
retract(s,p,o,id):Subspace	An in which does not remove a matched RDFTuple (which would be akin to negation) but replaces its ⟨s,p,o⟩ values with null values

The first set of operations are data view, i.e. a RDF tuple conforming to RDF syntax is accepted, no check is made against ontological (i.e. RDF schema) information. They serve to allow well formed RDF statements to be placed in the tuple space, and be retrieved destructively or non-destructively. As data view operations, templates act, in the sense of Linda, only on the syntactic level of the tuple, i.e. they can match on the basis of URI syntax matching or literal datatype matching. However, no ontological information is considered. This provides a less computationally complex means to acquire RDF tuples, e.g. retrieval by ID, or by (exact) URI match.

However, Semantic Web Spaces also define operations for the information view, where the RDF statements are also satisfiable according to a constraining schema and retrievable based on ontological information. Information view assertion acts as a test of the consistency of the knowledge asserted in the Semantic Web Space. RDF statements refer to a RDFS or OWL ontology, which describes the vocabulary and the meaning of these ontological constructs. For this reason RDF tuples which do not conform to the corresponding schema will be rejected. Note that RDFS statements about the classes instantiated within the RDF tuple are assumed to be available. If not, the RDF tuple is rejected at the information view level as it can not be checked for consistency. The tuple could still be asserted at the data view level, and maybe asserted again later when RDF Schema information is available.

A *claim* can contain either a single RDFTuple or it can contain a Subspace, which is defined at the client and contains one or more tuples. As well as allowing multiple tuples to be claimed in one operation, it provides the means to make claims which contain blank nodes. Within a Subspace, a blank node with the same identifier will be considered as being the same blank node when tuples are added into the space. A claim carries a truth value, i.e. it is making a statement about something that it purports to be true. An accepted claim exists as a RDF tuple or set of RDF tuples equally in the data view, however its 'truthfulness' is only a property of the tuple at the information view level. If the claim can not be substantiated, then the tuple or subspace is rejected and false is returned to the client. Note that in the case of a subspace, the entire subspace must be satisfiable or it will be rejected.

To read tuples from the information view, we propose the *endorse* primitive. A tuple matching the given template is considered 'endorsed' by the information view, i.e. that it has been found to be consistent with current ontological information. The match is returned as a subspace. This subspace may contain a single matching tuple, however in the case of blank nodes the 'linking' tuples are included in the response, e.g. if the matching tuple has a blank node as its subject, then the set of tuples with the blank node as their object are included (hence a Subspace is like the CBD proposed by Nokia).

Semantic Web Spaces also choose to support a solution for multiple read operations (i.e. get all matching tuples for a template). We propose the begin primitive as a version of the copy-collect primitive which acts on the information view. It works within contexts (a partitioning of tuplespace into URI-identified subspaces) - a context is created by the system into which all matching tuples are copied. A reference to this context is passed to the client who is given alone the right to access the context. The client can then make destructive reads in that context i.e. retract (*,*,*,*) to remove all of the tuples. When the context is empty the system destroys the context.

Note that *endorse* and *excerpt* can match tuples which are not in the data view but exist 'implicitly' in the information view (i.e. as inferrable tuples).

Finally, tuples may be removed from the information view. Yet this operation is questionable, as it is not the same as expressing negation (which is not supported in the Semantic Web Space). Rather, one is removing a statement from the set without denying its truthfulness. As a result, Semantic Web Spaces proposes the retract primitive which, if a matching tuple is found, replaces its subject, predicate and object in the information view with the empty value rather than removing it completely. The tuple remains in the data view. Hence its reference continues to exist but the assertion of knowledge that the reference makes is lost. As a result, all inferrable tuples from that retracted tuple are no longer available. Note that this operates the same as endorse, in that a subspace is returned and blank nodes will cause additional tuples to be retracted.

The information view primitives are, as with all Linda operations, blocking in order to support the co-ordination model of Linda. However, we do not consider this binding in an implementation of a Semantic Web Space. There may be cases in which it is preferable to have non-blocking uses (if a match is not found, a null object is returned) or blocking with timeout.

Semantic Web Spaces also answers the issue of synchronization between data and information views. Fundamentally, the information view is a RDF schema-consistent view of a RDF graph built from RDF tuples in the data view. A claim made in the information view will also be added as a RDF tuple in the data view. However, a retraction in the information view does not affect the data view. Conversely, the assertion of a RDF tuple in the data view does not affect the information view – it is not considered as a knowledge claim.

The destructive reading of a RDF tuple in the data view does however alter the information view. The claim is then retracted, with the related consequences. Hence, in any application of a Semantic Web Space, care must be taken in terms of which clients could destructively read which tuples.

Two issues arise in this modelling that a Semantic Web Space must handle. One is changes in the data and information view affect not only the tuple itself but also all tuples that are connected to it (e.g. exist in the sub-tree of the RDF graph for which that tuple is the root) or inferrable from it. Hence a single destructive read can have much larger consequences for the RDF graph in the Semantic Web Space. Similarly, changes in the ontologies being used to determine the satisfiability of RDF graphs may also cause tuples to be retracted. We do not consider these issues further in Semantic Web Spaces, but acknowledge that clients should be aware of the 'destructiveness' of single operations and that in cases restrictions may be advisable on client operations to avoid such destructive operations.

3.5 Comparison and evaluation

We can identify which types of Linda operation listed in Chapter 2 are supported by the semantic tuplespace systems listed in this chapter.

Linda extension	**cSpaces**	**TSC**	**Semantic Web Spaces**
External Transactions	get/begin/commit/rollbackTransaction	create/commit/abortTransaction	
Internal Transactions		update	
Counting operation	countN	count	
Non-blocking retrieval	take, read	take, read	
Operation with timeout		waitToTake, waitToRead	
Test for absence			
Multiple read	scan		extract
Retrieval with multiple templates			
Dispatch-obtain			
Update		update	
Rhonda			
Meta operations			

Notification	(un)subscribe, (un)advertise	(un)subscribe, (un)advertise	
Reaction			
Added programmability			

In Chapter 2 we also briefly identified pros and cons for the types of Linda extension from the point of view of the tuplespace model. This chapter has introduced three prototypical semantic tuplespace systems that are in development and some experiences gained from their implementation and testing. We can summarize our findings here in comparison with the expected pros and cons - whether extensions bring the expected benefit in a semantic computing scenario or if they raise new or additional complications.

3.5.1 External transactions

Both cSpaces and TSC follow the JavaSpaces approach of acquiring a transaction object. In cSpaces one must then explicitly begin the transaction and all following operations in the process are considered part of the transaction until the transaction is explicitly committed or rolled back. In TSC a reference to the transaction is used as a parameter in operations which is clearly more flexible but makes the coordination language slightly more complex . We have also noted that adding transactionality to a tuplespace raises important issues in terms of locking the space. TSC uses "optimistic transactions" to avoid locking the space during a transaction.

In TSC transactions can only be local. In the case of distributed transactions, we will need to consider how we offer guarantees and fault tolerance. This includes determining how long distributed transactions last, how and who uses these transactions and where the anonymity is if a client has to agree with communication partners on the applied transactions. One could use the space partitioning to locate a transactional conversation at a knowable address (e.g. querying partition metadata) while maintaining agent anonymity (knowing "where" but not "with whom").

3.5.2 Internal transactions

An alternative which we found in TSpaces was to allow arrays of tuples/templates in an operation and to handle the set as a transaction internally at the system end. TSC does this with its update operation, which is a transactional read and write handled internally by the system. In terms of replacing external transaction operations, an internal transaction operation, e.g. which takes a set of operations as a parameter, does not appear to us to bring any semantic advantages, rather the client loses the opportunity to reference current or even past transactions. The particular case of updates is considered in its own section.

3.5.3 Counting operation

Both cSpaces and TSC have supported the count operation. TSC has recommended refraining from this operation. The usual purpose of count is to allow clients to know how many times they should make a destructive read on a particular space in order to have retrieved all matches, as opposed to simply repeating the operation until it blocks. Of course, both cSpaces and TSC have non-blocking retrieval which effectively allows a client to know that no further matches are available, and seems to make count superfluous. In Semantic Web Spaces, the extract operation places all matching tuples into a seperate context from which they can be removed. A count may be useful here to know how many times the context can be destructively read. Hence the count operation may be useful in the case of copy-collect (multiple read) but can be rendered superfluous in a coordination model with non-blocking retrieval.

3.5.4 Non-blocking retrieval

Both cSpaces and TSC have supported non-blocking rd and in. The choice between blocking and non-blocking operations is dependent on which communication patterns are to be supported. Of course, if one relies on non-blocking retrieval one no longer uses the benefits of Linda synchronization. Hence they need to be proven necessary in a Linda based system on the basis of requirements to be included in its coordination language.

We also must consider whether we can talk about global operations at all. Linda makes no guarantee that a match is not present. Especially in a globally distributed Triple Space it is unrealistic to think we would ever search the entire space, and if so, we have to take into consideration that during a search, a match can be deleted or a new match added. Hence we agree that we do not consider these operations to make any guarantee at all. Triple Space, as a system whose content will respect the RDF or Semantic Web semantics (Open World Assumption), also raises a question over the meaning of a global retrieval operation. Under the Open World Assumption, the inability of a system to find a match can not be taken to mean a match does not exist, simply that it has not been found.

Restricting such operations to a defined (virtual) partition of the space may be a way to guarantee results respecting both the Linda and Semantic Web approaches similar to the ideas of scoped negation. Another possibility is the use of timeouts, see the following subsection.

3.5.5 Operation with timeout

Rather than have non-blocking retrieval, the consensus seems to be towards adding timeouts to the blocking operations. However we do not claim this as a global operation, rather each kernel will complete the search for a matching tuple according to some system defined policy and then exit and hence a fail will simply mean the search ended without finding a match, rather than that a match does not exist in the space.

TSC chose to include timeouts as parameters for the blocking retrieval operations. In distributed systems, it is clear that processes may be blocked for unknowable amounts of time. Hence timeouts do seem to be a reasonable extension to the coordination model, and operations can still be executed without a specified timeout if desired.

A problem with respect to the duration of the timeout is the amount of effort implied by such a call, e.g. how long an operation needs to determine a response. Imagine that a read is resolved only locally and without success, then we could expect that a system response is almost immediate, i.e. before the timeout is reached. However, if the same read implies queries to other kernels in a distributed system such a read call may last longer, potentially timing out before the response is received from other kernels.

Apart from providing clients with means to determine the length of time that should be allowed to ensure that the operation can be resolved, operations may time out without any indication of whether no response could be found or the time allowed for finding a response was insufficient.

3.5.6 Test for absence

Semantically this is simply the opposite of the non-blocking retrieval operations (or a count operation returning zero), hence we do not find a good reason to require this in a tuplespace API.

3.5.7 Multiple read

Multiple read is supported in cSpaces by a scan operator which returns a Set containing all matching tuples. In Semantic Web Spaces the operation is named extract and is based on the copy-collect primitive. Matching tuples are copied into a context and a reference to the context is returned to the client. The operation is necessary to overcome the Linda multiple read problem (that multiple rd operations may receive the same tuple) where clients need to identify all matching tuples in the space at a given point in time. We note however that it shares the semantic problems of the non-blocking retrieval operations of respecting both Linda and Semantic Web approaches.

Multiple read, if supported in the API, would have the same semantic as the non-blocking/timeout retrieval discussed above matches are collected from the kernels without guarantees of completeness or correctness, hence without freezing the search space. An open question is whether reading from a controlled partition of the space may be handled differently.

3.5.8 Retrieval with multiple templates

None of the systems attempt retrieval with multiple templates. It seems to be merely a shorthand for a transaction consisting of a number of retrieval operations.

Another interpretation of this would be a join between multiple templates, however queries with joins might be expressed better within a query parameter rather than as an additional operation in the API.

3.5.9 Dispatch-obtain

No system has supported dispatch-obtain. In this case however we must raise the question of whether the API developers were aware of this approach, which allows parallel retrieval whilst retaining the Linda semantics of blocking operations. It has demonstrated performance benefits and should thus be seriously considered, with the caveat that we have no experience in implementing it.

3.5.10 Update

TSC has an update operation which is implemented as a transactional single destructive read followed by a write. We have noted that an update only makes sense semantically in comparison to an *in* followed by an *out* where the reference to the tuple is maintained. Hence update in Triple Space is dependent on whether we have references for tuples and we want to maintain them when changing content in the space.

3.5.11 Rhonda

Rhonda is not followed by any of the systems under consideration. Semantically, it encapsulates in a single operation a notification with the predefined reaction to that notification. Hence we can reconsider the value of this operation in the notification/reaction sections.

3.5.12 Meta operations

Meta operations appear valuable but clearly become superfluous in a semantic system as we can describe the system, its structure and tuples semantically and hence include this metadata information in the space itself as tuples. So metadata operations become possible as normal retrieval operations in which the templates match on metadata stored as tuples in the space. Explicit metadata operations do not seem necessary.

3.5.13 Notification

Both cSpaces and TSC have chosen to support notification as well as a similar operation called advertise.

Given there is a need for notification as an interaction pattern, we can add that a notify operation can replace non-blocking retrieval and does not carry the same semantic problems. However notifications tend to operate only on the changes in the tuplespace after the notification becomes active, though we could change this semantic to include an initial restrictive multiple read just as a RSS reader first loads the last x items from the feed before monitoring for any new items.

3.5.14 Reaction

This operation type is a notification with a pre-defined reaction to an incoming notification such as changing the tuple and re-inserting it into the space. The clearest use of this would be in updates (remove the matched tuple, inserted an altered version of this tuple) and in Rhonda-like synchronization (respond to matched tuple with another tuple). Reactions instictively appear useful for Triple Space communication, though it may be preferable to support specific cases of reactions such as updates and rhonda rather than approve a general update operation and some specification of how updates are then specified. It is unclear if such expressivity is necessary. Both update and rhonda can of course also be expressed as an in/out and a rd/out pair respectively, though an alternative would be to consider the more expressive versions of notification.

By allowing reactions as extended notifications which exist in the Triple Space we can make clients lighter. It is open how a reaction will be expressed so that it can be interpreted by the system, possibly by calls to other services.

3.5.15 Added programmability

None of the systems under consideration provided added programmability. From a client perspective it is unclear if there would be any benefit from permitting this. Making the implementation of the space much more complex could prove a greater disadvantage than any advantage that might be gained.

3.5.16 Others: structural related operations

Finally, we have found some operations which depend on the tuplespace structure (WP2). In the case of multiple tuplespaces, there are operations to create and leave spaces (TSC). Likewise in TSC as the interaction structures are named graphs there is a distinction between read/take which operates on matching graphs and the query operation which matches on tuples independent on which graphs they exist in, returning that set of tuples as a newly built graph. Likewise, in Semantic Web Spaces a distinction was made between operations on a Data View and on an Information View. As the structure of Triple Space will be agreed on at the same time as the Triple Space API we do not yet need to decide on structure-related operations. Rather, we will introduce the structure of Triple Space in Chapter 6 (as requirements arising from WP2) and hence specify the necessary structural operations in Chapter 7.

3.5.17 Conclusion

Our consideration of these prototypes evaluated together with implementation experiences and our overview of Linda extensions in Chapter 2 have brought us to an initial evaluation of the Linda extensions for our purposes in Triple Space. Clearly this needs to be further refined by consideration of the interaction patterns and the requirements arising from the other workpackages, e.g. the tuplespace structure (WP2), integration with Web Services (WP4) and the use cases (WP8). However we can organize the extensions initially under the categories of necessary, optional and unnecessary for Triple Space. It is understandable that almost no extension is already stated as necessary, this will become clearer from the identification of requirements on Triple Space. However we can already see a set of extensions which are to be considered seriously (optional) against others which already seem to be out of scope for our purposes (unnecessary). Some choices are also clearly related to one another, e.g. between non-blocking retrieval, multiple read and notification.

Necessary	**Optional**	**Unnecessary**
Operation with timeout	External transactions	Internal transactions
	Count operation	Non-blocking retrieval
	Multiple read	Test for absence
	Dispatch-obtain	Retrieval with multiple templates
	Reaction	Meta operations
	Rhonda	Update
	Notification	Added programmability

4 Requirements from TripCom

4.1 Introduction

Until now, we have assessed the possible Triple Space API on the basis of what can be derived from the literature about the value of particular extensions to the Linda coordination model. In this chapter we turn to more specific factors: the concrete requirements upon the Triple Space API that can be derived from other workpackages in TripCom. In particular, five workpackages in TripCom are seen as relevant to deciding requirements upon the final Triple Space API:

- WP2 which deals with specifying the structure of tuples and tuplespaces in the Triple Space
- WP4 which deals with specifying how Triple Space will be integrated with the present (Semantic) Web services infrastructure
- WP5 which deals with specifying security and trust solutions for Triple Space
- WP8a which deals with specifying an Enterprise Application Integration (EAI) use case for Triple Space
- WP8b which deals with specifying an eHealth use case for Triple Space

4.2 Requirements from the tuple and tuplespace structure

The tuplespace model in TripCom foresees three fielded tuples, corresponding to the subject, the predicate and the object of an RDF statement. Tuples are identifiable by means of an URI, which are allocated to instances of a Tuple class in a tuplespace ontology. Tuples can be grouped into graphs, which are identifiable by means of URIs as well. Hence the prospected Triple Space API should adapt the signatures of the co-ordination primitives to these structures: it should allow agents to insert single statements or sets of statements to the tuplespace and to refer to them using identifiers or templates to be matched to their content.

Hence the tuple structure makes the following requirements upon the coordination operations:

- Tuple operations *must* accept as a parameter either three RDF resources which signify the subject, predicate and object of a RDF statement or a RDF statement itself. Considering the need to bind the coordination language to client programming languages and network protocols, we prefer to allow three parameters which can be expressed at the client as URIs rather than one parameter which must use a dedicated tuple datatype.
- Tuple operations *must not* accept as a parameter an identifier for a tuple or a named graph in the space. This is in line with the principles of tuplespace computing (associative addressing rather than by reference). Unique identifiers are handled by the space itself and expressed through the tuplespace ontology, hence operations acquiring IDs for tuples/graphs or using them in retrieving data can be expressed through the existing coordination model and the appropriate RDF queries expressed by the tuplespace ontology.

- Tuple operations *could* accept as a parameter a multiset of RDF statements, i.e. a RDF (named) graph. This makes the most sense in tuple emission as a form of multiwrite which transactionally places all or none of the statements in a named graph into the space. It is more questionable in the case of retrieval, as it raises issues of graph matching as a form of RDF retrieval and may be replacable with the use of an appropriate query language.

- Tuple operations *could* return as a response a multiset of RDF statements, i.e. a RDF (named) graph. This makes the most sense in non-destructive tuple retrieval as a form of multiread which places all matched statements in a named graph and makes it available to the client.

As regards the tuplespace organization, the co-ordination primitives should take into account the tree-like organization of the tuples in spaces, which might be further divided to (sub-) spaces. The spaces are identified by means of URIs as well. Accordingly the co-ordination model should allow agents to perform operations on pre-defined spaces, and to create or destroy these.

Hence the tuple structure makes the following requirements upon the coordination operations:

- The ability of agents (with appropriate authorization?) to create and possibly destroy a space. The operation could allow the agent to provide a means of identification for the space (an URI) or returns an unique identifier if ID allocation is handled by the space itself. We tend towards the latter option in order to maintain unique URIs.

- The ability of agents to identify which space another space is the child of. Rather than allow the movement of spaces, we consider this a part of the initial creation operation.

- The ability of agents to specify in which space they wish to interact. This *must* be a parameter on the coordination operations. We can assume agents can identify which space they want to use through querying metadata in the tuplespace ontology or some other means outside of the coordination model.

- The ability of agents to interact in multiple spaces may be expressable by the support for nesting spaces, so that agents can always switch activity to a parent space in order to be able to additionally access tuples in any of its child spaces. We either expect that the parent space can be identified by querying the tuplespace ontology or we could imagine a ”parent” keyword with that semantics.

- The support for scopes [28] (virtual overlapping views on a space which can group tuples from different, unrelated subspaces) is currently undefined but we note if scopes are identified by URI then agents can use the subspace parameter to also express a scope URI. The system would have to be able to differentiate if a scope or subspace is being referenced by a certain URI.

- However as the effect on the structure of the space is different with contexts as with subspaces we would need separate creation and possibly destruction operations for scopes. Scope creation would need to be based on some filter which defines which tuples are members of the newly created scope. We note that where this filter would be a template, we have the same semantics as the multiple read operation "copy-collect" [37]

- Additionally, scopes can have additional operations applied on them (e.g. to join them, or to access their intersection). One means to do this would be to extend the scope creation parameter to express these operations and allow scopes to be created as joins, intersections etc. of other scopes.

A final issue for the coordination model is whether we allow spaces to be first class objects in the Triple Space. It seems that we will indeed be able to refer to spaces as they are identified by unique URIs in the TS ontology, and hence authorized agents could be able to add and remove statements about the spaces, which could affect equally the actual logical structure of the Triple Space. Rather, we prefer dedicated create/destroy space operations because they would encapsulate a set of system-internal insertions and deletions of statements in the tuplespace ontology (not only an instantiation statement but also the set of metadata associated to that instance). This protect accesses directly on the tuplespace ontology, and prevents other operations which we decide to disallow in Triple Space such as copying or moving subspaces.

4.3 Requirements from Web service communication

We distinguish between two kinds of requirements on the Triple Space API. The "low-level functionality" describes direct requirements on the coordination primitives, while the "higher-level functionality" describes requirements which result from integrating TripCom and Web services.

4.3.1 Low-level functionality

For retrieval, we find classical Linda "rd" is necessary and "in" is optional. We question whether a destructive read operation is necessary, as Triple Space aims for persistent publication and the statefulness of Web service interactions requires that earlier data exchanges are sometimes referred back to, in fact this is a shortcoming of the current message-based Web service communication that Triple Space should solve.

Three ways of identifying the tuples to be retrieved from the space are needed:

- based on the tuple content: The tuple content is identified via a template or query (e.g. SPARQL-like). This is required to realize discovery mechanisms and facilitates retrieval operations like "Get the graph that matches with template t"

- based on unique triple/graph identifier: The tuple is uniquely identified via appropriate metadata, thus allowing distinction of tuples with identical content and simplifies the implementation of interaction patterns (e.g. request/response, WS-Addressing: RelatesTo-header). This can be used e.g. for "Get the graph that contains resource r"

- based on context: The tuple is identified via tuple context (metadata). Possible retrieval operations are e.g. "Get the graph published by user X"

In fact, most of the examples are solved by using the TS ontology to express an appropriate RDF query and do not require any extension of the classical Linda retrieval operations. Only the first example "Get the graph that matches with template t" raises the question if this is expecting the semantics of multiple read (all tuples matching a template), getAny (all tuples matching any of a set of templates) or getAll (a set of tuples matching all of a set of templates). We note that where multiple read generates scopes, a getAny could be modelled by creating a scope as a join between the scopes created by a set of multiple read operations.

Furthermore two kinds of retrieval operations have to be supported:

- blocking operations: To support polling for tuples (e.g. through WS-Polling) timeouts for blocking operations are needed. This also allows to natively support certain interaction patterns, e.g. master-worker.
- non-blocking operations: If global view operations are supported, then polling can also be realized via non-blocking operations.

The count operation can be used to enable monitoring the number of graphs matching a certain template.

Both the rhonda operation and Bonitas dispatch/obtain are optional in WP4.

The write / out operation will be used to publish an RDF graph G. Two alternatives are possible:

- The client knows, in which sub-space G is to be stored ("Publish the RDF graph G at sub-space X")
- The space should decide in which sub-space G is to be stored ("Publish the RDF graph G")

To enable messaging over the triple space, a message channel between communication partners has to be provided. This can be achieved by publishing to and reading from the same sub-space. This requires ordered delivery of messages, which can be supported in the Web services layer. Building upon these semantics allows us to realize a virtual message channel over the space and permits us to replace existing EAI messaging technologies.

The users should be able to subscribe to one or more available channels. For the channels a user will subscribe to, will be notified for any message published under that channel.

- "Please subscribe me to subscription channel x"
- "Please unsubscribe me to subscription channel x"
- "Please let me know about all the messages published that matches with template t", i.e. template-based notification.

In-place updates through the 'update' operation may be useful to realize stateful resources (with frequent state changes) in the space and could be used in the way of "Update the RDF graph G1 at sub-space X with the new graph G2"

4.3.2 Triple Space Management

- Create a new space
- Remove a space
- Renaming space

Spaces can have a recursive structure, i.e. a space can contain spaces (sub-spaces).

4.3.3 Transactions

WP4 requires two different types of transactions:

- local transactions: Local transaction are needed e.g. for transactional storage of multiple tuples on one node. This way, the TSpace-like multiwrite operation can be emulated. Support for different isolation levels might be useful to support higher concurrency on a Triple Space node.
- distributed transactions: To encapsulate multiple service invocations into one single transaction, we need distributed transactions to coordinate service requesters, service providers and the space. Even a single space-based service invocation has at least three participants, thus requiring 2PC between the partners.

¿From the clients view, both types of transactions are used through the same interfaces. The space decides, which transaction type has to be applied.

4.3.4 Event reactivity

To support the publish/subscribe pattern, event-reactivity via notification is required.

4.3.5 Web service interaction patterns

A direct mapping from WS interaction patterns supported by WSDL 1.1 and 2.0 to the Triple Space API primitives has been given in the previous chapter. As a consequence, also more complex composite interaction patterns as proposed in e.g. WS-Coordination, WS-Eventing, WS-Polling, WS-Transaction, WS-Notification etc. should be realizable by either composing the accordant Triple Space interaction primitives or even by employing an alternative TS Coordination primitive which ideally provides improved performance in some particular respect (e.g. response time, fault tolerance, scalability, comprehensibility, etc.).

4.4 Requirements from security and trust

In terms of the security and trust approaches taken in Triple Space, we are interested in those which potentially require certain operations or operational semantics (including parameters for security or trust information). We consider the possible requirements on the API from the model for tuple space security presented in D5.1, section 3.4 However we should also note that primarily the aspects of security and trust will be implemented outside of the Triple Space API, e.g. as part of the communication protocol or expressed to / applied to data contained within the tuples.

We foresee a "protocol authentication process" (PAP) to which a client presents its credentials and from which it receives an authentication token. Whether authentication has some effect on the coordination model will depend on two design decisions that are yet to be made:

- whether the PAP is external to the Triple Space kernel or is also accessed through interaction with the Triple Space (possible request for authentication operation)
- whether the authentication token is passed by the client in every interaction with the Triple Space (necessary additional parameter in operations)

As we prefer to avoid additional complications of the Triple Space coordination model, we currently assume an authentication service which operates independently of the Triple Space but is trusted by it. Once an agent is accredited, the authentication service notifies the Triple Space "agent X has been authenticated" and hence the Triple Space allows operations by agent X upon it according to its access policies. Note that it has not yet been clarified by what means agent X is recognized as such by the Triple Space in a secure fashion: hence we keep open the possibility that authentication tokens will be a necessary additional parameter in Triple Space operations, while seeking an alternative means which can avoid extending the coordination model footprint, e.g. identification by IP address.

Another means to protect data being exchanged over the Triple Space would be to allow agents to share key pairs (encryption-decryption) so that one agent may publish data encrypted by its key in the space and only authorized agents who have the corresponding decryption key may be able to access that data. Given the tuplespace requirement of agent anonymity we could assume that keys are exchanged in some way through interactions with the Triple Space. Of course, an alternative is that keys are allocated by another service outside the Triple Space, possibly combined with the authentication service discussed above. The latter option would also keep security aspects outside of the Triple Space API (keys remain at agents and do not appear in the space where other agents might be able to access them), but would be affected by the structure of the operation parameters:

- if operations use three parameters for s, p and o then we could support encryption at the resource level, but possibly not at the tuple level
- if operations use a single parameter for a triple then we could support encryption at the tuple level, but possibly not at the resource level

Finally, some meta-information would be necessary to guide agents with a certain decryption key to correspondingly encrypted tuples. However, rather than place such metainformation as a parameter in the operation, we can consider either using agreed subspaces for secure conversations between some agents or expressing this metainformation as tuples (property-value pairs on the encrypted tuple or graph as subject).

Another case for metainformation is the access rights of an agent. This can be expressed in some formal way, such as by Access Control Lists (ACLs). The major question here is if security metainformation should be handled in the Triple Space over the Triple Space API or through dedicated interaction with a security component which may be part of the Triple Space. If we use the Triple Space API, can we re-use the Linda (extension) operations with dedicated RDF vocabularies (place security information into the tuples rather than into the API) or do we add new operations/parameters?

In the former case, there is not only an open question of which vocabularies exist to be used but also how the data can be securely manipulated in the space. As it would be part of the TS ontology, we could require that interactions with TS ontology-based information is handled in a separate way to interaction with the TS content, e.g. access is restricted to the corresponding TS administrator with the correct authentication and encryption. However we do not need to require that new operations are defined, but that the access layer would have to analyse tuple content to identify TS ontology operations.

Currently we prefer the first option in order to retain simplicity in the coordination model.

Finally, trust information may be as much part of the space as security information. However, we do not see any requirements from representing trust in the space which impact on the coordination model.

Finally, Triple Space should provide for security-related error codes such as "you are not allowed to perform this operation", or "the owner of this data is ...", or "this result is incomplete as you are allowed only to access to partial information".

4.5 Requirements from the EAI use case

Some extension that different APIs can extend Linda language can be:

- Usual operations: The typical operation of rd, out and in are accepted by this scenario. The communication among different actors in this scenario will be performed through TS. So negotiation, a service request and so on, consist on writing TS in order to ask, offer, etc something, and read from the TS to know what happened and write again to answer it.

- Transactions: Transactions that involve acting atomically on a group of data perhaps don't have a lot of sense for DAM use case. This can be because this use case considers negotiation among different roles all the time, which imply a lot of messages interchanged between them, and that a concrete actor doesn't do a lot of operation continuously. So that all the extension that take into account, at this first stage of the description, can be not needed.

- Blocking: Taking into account that the goal of this scenario is to have an asynchronous communication among different roles, and that several entities can respond to some data, it makes sense that while some entities are emitting data to the TS, others will wait to get this data.

- Count: The count operation returns an integer which is the number of tuples in the tuplespace which match a given template. This could be interesting for this scenario when it's wanted to obtain some information from TS that match a concrete query or template.

- Subscribe: The subscription (subscription and unsubscription) involves receiving a notification when a concrete template has been matched. This can be useful in our scenario, for the service provider in the case it is waiting an answer from several content providers.

4.6 Requirements from the eHealth use case

The eHealth use case is centered on the exchange of data in the form of an European Patient Summary (EPS) between heterogeneous, distributed health systems.

4.6.1 Transaction operations

The storage needed by the EPS may be seen as a big virtual storage where the summaries of each citizen are stored (no matter for now where the summaries are stored). Each summary is made up of several sections (allergies, medical histories,) and each section includes several entries which report the specific basic information (the citizen is allergic to penicillin, the citizen was recovered on day X in place Y due to problem Z). The EPS scenario requires that a care giver creates one or more entries in one or more sections of the same summary in an ACID transaction.

This can also be supported by multiwrite which is a form of local transaction. However this operation only supports the case of one agent performing all write operations. We expect actually that support for distributed transactions will be necessary as different health systems will share in writing or retrieving some data about a patient to or from the space. Locking (in the case of transactions) needs to be implemented for sections of the EPS, not the EPS as a whole which we should expect will be accessed by over a million users. Each patients records should constitute a protected section. Most accesses should be transactional, not single triples.

4.6.2 Global View operations

In the EPS scenario, there is the need to verify if the summary of a specific citizen exists in the TS. There is the need to verify the presence or count the data on a specific summary without blocking any access to other summaries. Hence blocking operations may be complemented by non-blocking variants which are used when retrieving from a partition of the space (a single EPS).

4.6.3 Multiple Read

The EPS scenario requires that a specific summary doesnt change its observable state while an external component retrieves it. Of course, the other summaries in the EPS space can modify their state. We could argue that we only multiple read within protected partitions of the space e.g. each patient summary could be clearly delimited in the space as a single named graph.

This wouldn't solve immediately multiple read across EPSes - guarantee of correctness would be a precondition in this scenario. This might be a case of scopes rather than an operation in the API.

4.6.4 Retrieval operations

As these operations dont provide added capabilities to the Triple Space but are simply alternatives, they are very welcome but optional.

4.6.5 Meta operations

The EPS scenario doesnt require this operation directly.

4.6.6 Event reactivity

This is a very important operation for the EPS scenario. For example, its possible that some external application subscribes itself in order to be notified when some data is added in the TS.

4.6.7 Added programmability

Any read or write operation on a summary must be tracked by the system, together with some information about the user who is performing the operation, the date-time and the changes of the state. This programmability is needed for the internal operations in the TS but, at this stage of the development of the use cases, there is no requirement for added programmability at the external level.

4.7 Summary

It is now possible to draw up which operations and parameters are necessary in the Triple Space API from the other workpackages defining aspects of the tuple(space) structure, its security and trust infrastructure and use cases for the implemented Triple Space. We can divide requirements between those relating to the operations (Linda and its extensions) and those relating to parameters of the operations .

4.7.1 Operations

The table shows for each workpackage presenting requirements on the Triple Space API operations which of the extensions considered in this deliverable are superfluous (empty box), optional (O) or necessary (X). We have noted that external transactions in these scenarios tend to be necessarily distributed (transaction shared between different agents). Furthermore, we divide retrieval with multiple templates into getAny (all tuples matching any of the templates) and getAll (a set of tuples which matches all of the templates) to more clearly differentiate their semantics.

Regarding the core Linda operations in, rd and out we note there was consensus from all workpackages to use these for classical Linda-style coordination.

Linda extension	**wp4**	**wp8a**	**wp8b**
External (Distributed) Transactions	X	X	
Internal Transactions (multiwrite)	X		X
Counting operation	O	O	O
Non-blocking retrieval	O		O

Operation with timeout	X		
Test for absence			O
Multiple read	X	X	X
getAny	O		
getAll	X		
Dispatch-obtain	O		O
Update	O		O
Rhonda	O		O
Meta operations			
Notification	X	X	X
Reaction			
Added programmability			

In addition to this list we can add the operations on tuplespace structure which are needed to support nested spaces.

- *Space creation* with a parameter containing the URI of the parent space. If no parameter is given, the parent space is the global Triple Space. An URI is returned which uniquely identifies this space.

- *Space destruction* with a parameter containing the URI of the space to be destroyed. We expect that this option will only be open to authorized agents, e.g. the creator of the space. All tuples in the space are destroyed, including those in all child spaces of the space. This operation is effectively the same as destructively reading all tuples from the space, as well as removing references to the destroyed space(s) from the tuplespace ontology.

We do not explicitly define at this stage the operations for scopes, as it is currently undefined if this will be supported in Triple Space. We note however that one can use the multiple read operation ”copy-collect” as a scope creation operation, using a template as the filter. Extensions might be necessary to support other means to build scopes such as joins, intersections etc.

4.7.2 Parameters

All operations which require a tuple or template will use as parameters three URIs representing the subject, predicate and object of a RDF statement. The third field, representing the object, can alternatively contain a RDF literal, i.e. a value in some simple datatype such as string or integer. A template will use a variable of type URI or other datatype rather than a concrete value. A multiset of RDF statements, i.e. a RDF named graph, can be a parameter for the tuple emission operation, transactionally placing all or none of the statements in the space. A multiset of RDF statements could also be used for tuple retrieval (like getAny and/or getAll [13]), returning a multiset of RDF statements as answer in the form of a RDF named graph. Here, a getAny type operation could return a newly constructed graph and a getAll type operation could return a reference to a matching named graph already in the space.

Finally, all operations on the space should support an extra parameter which takes an URI identifying the space on which the operation will be applied. If no parameter is given, the operation applies to the global Triple Space.

In terms of security and trust, we attempt to avoid additional parameters on operations which contain security/trust information. Rather, we consider much of the security/trust infrastructure as being orthonogal to the access layer. It may be that security/trust interactions can be expressed within the Triple Space API, using dedicated tuple structures for the security/trust information (or triples based on some RDF vocabularies for expressing that information) and possibly associated to a separate security space (using the space URI parameter on the operations).

Finally, it has been proposed that Triple Space supports error reporting. On the other hand this places the task of error handling into the API, and an alternative is to use the existing error handling capabilities of the protocols and programming languages used by Triple Space, e.g. exceptions are part of most object oriented languages. An alternative is to define abstractly meaningful error codes that the TS API implementation should ideally return so that clients have an indication of the reason for a failure in their interaction with a space.

4.8 Conclusion

In this chapter we considered what requirements arise from workpackages defining other aspects of Triple Space such as its structure and security/trust infrastructure as well as those aiming to realise specific communicative scenarios through Triple Space (Web services, EAI, eHealth). This has helped us to further see which operations are important as part of the coordination model of the Triple Space as well as to refine which parameters should be supported by the operations. In the following chapter we present the API of Triple Space.

5 Triple Space API

5.1 A semantic Linda

Having completed a full and thorough state of the art and requirements analysis, we are able to develop a specification for the Triple Space API. This is not a single action, but rather a progressive development based on the available information (which is given in the previous chapters) and partner discussions. As a result, first of all we present issues that were identified and the contents of our discussions on those issues. Subsequently, there is a first analysis of the operations to be supported by the API. A brief overview of how our specification of the API evolved across the given discussions precedes the agreements on the parameters to be supported and a final API specification. This API is frozen for the first implementation of a Triple Space prototype.

5.2 Issues and discussions

This section summarizes the content of discussions we had on some key open issues relevant to the API. The full discussions are referenceable in the Semantic Linda open issues document in TripCom CVS and the TripCom WP3 mailing list.

5.2.1 Transactionality

Database transactions are generally short-duration and hence preserve ACID properties (as by, for example, 2PC) by locking the resources involved in the transaction. Web service transactions can be long-running and hence relax ACID properties - by each interaction in the transaction, the storage is updated and visible (as with a commit), however in the case of a rollback, "compensation" mechanisms are used to return each storage to its state before the transaction.

It seems unrealistic to lock resources in the space, preventing access by other processes, particularly as both the scale of the resources and the duration of the transaction are unknowable and can be large.

The middle approach here would be to commit each step of the transaction in a private snapshot of the resources being interacted with, so that the public storage remains visible and in the state before any (not yet committed) transaction. Once the transaction is committed, the public storage is updated with the final state of the private snapshot. However, maintaining snapshots of each data set being transactionally used may become very large and with RDF there is the question of defining the bounds of this data set.

Here it is important to check if data consistency is violated when the current state of the data is updated with the snapshot. Updates performed by other processes could be lost. Doing this, we will quickly end up with optimistic transactions, i.e. inconsistencies may occur during transaction processing and are only detected at commitment time. This way, you don't need to block resources for a longer period of time.

An "Optimistic Transaction" does not guarantee the execution of the changes in a transaction upon its committal. Instead a callback is used to indicate whether the commit has succeeded or failed. In other words, no lock is made on the data until the commitTransaction() is called. Hence optimistic transactions are useful when long-running parallel transactions are not operating on the same data. Optimistic transactions do not use any locks, but rely on resolving conflicts at the time of committal.

5.2.2 Synchronicity

It was decided that an non-blocking operation will never be equal to a timeout of zero, and should thus neither be modeled as such. We suggest to simply provide blocking operations with timeout, where the local call returns after the timeout. Although some implementations use a timeout parameter 0 to indicate non-blocking behaviour, this is misleading.

5.2.3 Guarantees

The idea to use optimistic concurrency control would be, as in transactions above, to take a snapshot of the searched space at the time of execution and to apply the global operation to this snapshot. Of course, by completion of the operation, the original data set may have altered and the operation's result is not longer correct. However, just as differences between the snapshot and actual data need to be checked at the time of transaction committal in an optimistic scenario, maybe the difference could be checked at completion of the operation and its result modified accordingly.

However it must also be noted that Triple Space does not necessarily have to guarantee correctness on a global operation, and the feasability of taking a snapshot is highly dependent upon the system's ability to determine the bounds of the client query.

5.2.4 RDF specific constructs

Blank nodes are handled according to the RDF semantics, and to avoid problems with clients losing references to linked tuples through the existence of a blank node in the query result, we choose to support query response of a closed graph (no blank nodes at any open node). In other words, for a template (?s,p,o) and a matching tuple (s,p,o), for o being a blank node, we return also all tuples with that blank node as subject, and for s being a blank node, we return also all tuples with that blank node as object. Thus we propose a similar approach to that suggested by Nokia in step 2 of Concise Bounded Descriptions [1]. (see below)

Containers/collections are decomposed into their triples. RDF APIs tend to support them as specific constructs and this is the best way to support them in Triple Space.

Reification is supported as the RDF semantics expect. A client can build a reification of a RDF statement in a RDF graph (4 triples) and knowing the statement's URI any client can add statements about that statement.

5.2.5 Richer retrieval

In the parallel work in TripCom D3.2 we will first propose a query language and in the prototype of this activity, we choose to support simpler retrieval, i.e. Linda templates

[1]`http://www.w3.org/Submission/2004/SUBM-CBD-20040930/`

of (s,p,o) matching on RDF tuples. This can be altered to support the test cases from the scenarios, e.g. if we want to match on conjunctions or disjunctions of templates.

5.2.6 Removal or invalidation of tuples

Triple Space is based on exchange of knowledge and on persistent publication. While data has only the property of existence (either a resource is available at an URL or it isn't), knowledge has the property of truth (If I publish my telephone number on a website, removing it later does not mean it is no longer my telephone number). In logical models operating under the Closed World Assumption, the deletion of a statement in a factbase could be understood as negation (”negation as failure”). In the Semantic Web, which operates under the Open World Assumption, the non-existence of a fact is interpreted rather as ”unknowableness” - its truth value is unaltered. Hence classical destructive read in Linda systems - removing the tuple and all references to it from the tuplespace - does not seem to be applicable to a semantic tuplespace of knowledge statements rather than data tuples.

We will support destructive read initially, however we could extend this model later to indicate that a tuple has been invalidated, rather then removed. Invalidation simply means that the tuple is no longer valid for the agents accessing the (sub)space; in a sense this means we take a subjective view on the world of knowledge (no expectation of absolute truth). Rather, each agent can express what is truth for it in the space, however authorized agents can equally invalidate those truths (”for this subspace, this is not valid”) to other agents. For example, subspace administrators will probably tend to act as authorities for the ”truth” expressed within their subspace. However, it would not be right for them to act as authorities for truth in the global space, as no agent can decide what is ”truth” for any other agent.

5.3 Agreeing on the operations

Firstly, we are agreed on retaining the classical Linda operations of out, in and rd as these form the core co-ordination model of tuplespace computing. We note that the EAI use case prefers an ordered semantics for out, however it is to be examined whether this can be satisfactorily modelled with an out followed by a (blocking) read on the outed tuple. Additionally, the semantics of in are deliberately left unaddressed. It is possible that rather than destructively reading a tuple from the space, and hence making all information about it unavailable to all other processes, that in may act as a sort of invalidation on the tuple. While classical Linda returns tuples in a non-deterministic fashion, a semantic Linda may see valid tuples and ignore invalidated ones.

The review of Linda extensions and their application to semantic tuplespace systems gave us this table:

Necessary	Optional	Superfluous

Operation with timeout	External transactions Count operation Multiple read Dispatch-obtain Reaction Rhonda Notification	Internal transactions Non-blocking retrieval Test for absence Retrieval with multiple templates Meta operations Update Added programmability

For each of these extensions we return to the requirements of the workpackages from the previous chapter. We will make a final decision for each extension based on the combination of its recommendability in terms of tuplespace computing, interaction patterns and workpackage requirements. That decision can either be ”core” (C), ”optional” (O) or ”unnecessary” (U).

Linda extension	**Attr.**	**Discussion**
External (Distributed) Transactions	C	This is important for ensuring fault tolerance in a set of interactions with the space. Given that Triple Space will be used by different agents to co-ordinate knowledge-based tasks, transactionality is vital. Its importance is already clear in the Web services paradigm which Triple Space will support. Distribution of transactions is however an open implementation issue.
Internal Transactions (multiwrite)	C	While not needed in general, writing a set of tuples in one atomic operation is a specific case of an internal transaction operation which Triple Space needs to support. As its internal data model is individual triples, this provides a means to emit a Named Graph to the space in a single operation.
Counting operation	O	Seen as rather trivial from the tuplespace computing and interaction pattern perspectives, however noted as useful by all Triple Space scenarios. It is to be tested how much a count operation is actually needed and whether multiple read followed by reading from its results wouldn’t be sufficient in most cases.
Non-blocking retrieval	U	We prefer to use a timeout parameter (see below).
Operation with timeout	C	Timeouts can help avoid that processes block for excessive periods of time, as is possible in open distributed systems.
Test for absence	U	Not seen as adding much to a coordination model which already supports non-blocking retrieval.

Multiple read	C	Linda has a problem with reading multiple tuples with the same template, in that it can't guarantee that you don't get the same tuple multiple times. Multiple read is a solution to this, and seen as necessary for supporting interaction patterns as well as Triple Space scenarios.
getAny	U	Effectively a merge of several multiple read operations. Would be replacable by merging result sets or a query operation using a dedicated query language which can express this operation.
getAll	O	Effectively a join of several multiple read operations. Hence, we could also replace it by a join operation on result sets. However, a possible alternative semantic for this operation would be graph matching like in the TSC system, where a Named Graph in the space is returned whose contained tuples match all templates in the operation. This is a different interaction than multiple read as all matches must be found in the same graph, rather than constructing a set of results from all matches in the searched space.
Dispatch-obtain	U	No necessity is seen for this operation, but its performance results from Bonita suggested it may be worth examining if performance becomes an issue in the prototype. However, there is possibly little difference to making a notification request and canceling the notification after the first matching tuple.
Update	O	No necessity is seen for this operation at the tuple level. However it may be useful for updating Named Graphs directly and hence avoiding in(Graph) followed by an out(Graph) which fails and leaves no graph remaining in the space.
Rhonda	U	No necessity was identified for this operation.
Meta operations	U	Given that metadata will be expressed in tuples themselves, meta operations can be modeled by standard operations containing tuples/templates with metadata
Notification	C	Notification is seen as an important extension to the Triple Space coordination model necessary to support certain forms of interaction pattern as well as all Triple Space scenarios.
Reaction	U	No necessity was identified for this type of operation.

Added programmability	U	No necessity was identified for this type of operation.
Create space	C	Given multiple nested spaces, a means is necessary to create a new child space.
Destroy space	C	Likewise, it is necessary that the space creator may destroy (remove all references) to their space. The space will cease to exist while the tuples will retain a memory of once existing in that space.

5.4 Evolution of the API

Through discussions by mail, conference call and F2F, a number of further refinements of the proposed API (which was based on the set of core operations in the previous table) took place:

1. We simplify the types of parameter taken by the operations to Template and Graph. A binding of the API to a programming language will need to be able to define datatypes for both. We drop Tuple (as a Graph can be defined to have one triple) and Set (as a set of RDF triples can always be seen as a graph).

2. A Graph is a RDF graph which can optionally be associated with an URI identifier (and is then considered a Named Graph, see http://www.w3.org/2004/03/trix/)

3. The original Template is a data construct consisting of three ordered fields which model a RDF triple (including type constraints). The fields can contain variables as well as concrete values and hence the Template models the classical Linda template constrained to match on RDF triples.

4. We consider Template to signify an abstract datatype, meaning it is open to further redefinition in the API, e.g. to include complex template types formed as conjunctions or disjunctions of atomic templates, or eventually to be a query expression based on a RDF query language. In other words, as we extend semantic matching and hence the type of template (query) supported, we do not need to alter the API but only the definition of what a Template contains. Also, this removes the need for a separate query primitive as query support would be provided by the type(s) of Template permitted.

5. We provide abstract error types for the API and expect concrete error reporting to be handled by the programming language and transport layer.

6. We removed references to scopes as there are no plans to support them in WP2 at present.

7. We removed out for a single triple as this can be achieved with out(Graph) where the graph is a single triple.

8. We renamed rdall as it does not guarantee to read all results to simply rd (multi-read) and added in (multi-in), renaming the original rd and in operations to rda and ina to indicate they return a single match. rd and in is preferred for the multiple match operations as we expect them to be more commonly used in RDF data retrieval.

9. We replaced getAll with rdg (graph read) in order to make operation naming more consistent

10. In order to ensure that Named Graphs can be manipulated the same as other Graphs, we introduced ing (graph in) as well

11. Notify is renamed as subscribe and unnotify as unsubscribe in order to better reflect that we named the resulting processes Subscriptions

12. We removed count as it can not make guarantees (no locking of the space) and one can use rd and count the triples in the graph locally

13. Update is left out of the core API. It is the same as graph-in/graph-out within a transaction.

5.5 Agreed parameters

Summarizing our discussion on parameters, we note:

- we attempt to specify this API in a programming language neutral way, so that it can be mapped into different host languages. However we must name the parameter and return value types. Here we use Jena names but note that other data structures could be used to model this API.

- a tuple is a triple encapsulating subject, predicate and object in one (programming language) data structure. It would be the decision of a concrete API mapping to a programming language to determine how a triple is represented, e.g. a three fielded ordered array, or as a RDF Statement.

- each subject, predicate and object can be an URI or blank node (objects can also be Literals as defined in the RDF spec).

- a template is originally a tuple in which any value may be a variable rather than a concrete value.

- a template in the API may be extended in due course (e.g. various subclasses in an OO language) to support more complex data structures such as conjunctions or disjunctions of single templates or richer query expressions.

- operations with sets of tuples use RDF (Named) Graphs. A programming language would need to decide how this is represented in the host data structures. An extension to Jena exists to handle Named Graphs.

- tuple operations also take an optional parameter which refers to a space by its URI

- tuple retrieval also take an optional parameter which specifies a timeout for the operation. No value is classical Linda blocking (wait indefinitely until a match is found).
- we add a transactionID parameter which allows associating an individual API operation with an active transaction.
- we avoid placing security and trust information in parameters.
- we provide an abstract set of error messages, and expect that Triple Space supports the error notification capabilities of the transport layer and host programming languages.

5.6 The Triple Space API

In this section we present the Triple Space API which resulted from the analysis, requirements collection and work package discussion outlined in this deliverable. Optional parameters are in [square brackets].

We leave the set of error messages outside of the table. These are listed afterwards with an indication of which operations may return them.

Operation	**Parameter**	**Returns**	**Additional notes**
out	Graph g, URI space, [URI graphID, URI transactionID]	void	Multiwrite operation. Inserts either all or none of the tuples included in the graph. If the graphID parameter is specified, creates a new Named Graph in the named space with the specified identifier which contains all tuples of the graph. A transactionID can be specified if the operation is to belong to a given active transaction.
rda	Template t, [URI space, URI transactionID, integer timeout]	Graph g	The timeout parameter takes an integer which is to be understood as a time period expressed in seconds. Returns a single matched tuple together with any tuples bound to it following the approach of handling RDF constructs.
rd	Template t, [URI space, URI transactionID, integer timeout]	Graph g	The timeout parameter takes an integer which is to be understood as a time period expressed in seconds. Returns as many matches as possible found prior to timeout together with any tuples bound to them following the approach of handling RDF constructs. If no timeout is given, returns matches after the global space has been searched, as determined by the Triple Space.

rdg	Template t, [URI space, URI transactionID, integer timeout]	Graph g	Returns the entire content of a Named Graph from the space which contains a match for the template.
ina	Template t, [URI space, URI transactionID, integer timeout]	Graph g	As rda, but destructively reads the returned tuple and its bound tuples in the space. What a destructive read means in a Triple Space shall be further specified.
in	Template t, [URI space, URI transactionID, integer timeout]	Graph g	As rd, but destructively reads all matched tuples and their bound tuples in the space.
ing	Template t, [URI space, URI transactionID, integer timeout]	Graph g	As rdg, but destructively reads the graph from the space.
subscribe	Template t, URI space, Listener l, [URI transactionID]	URI subscription	Notification in the space is established by providing a template and a listener object. Listeners are for example found in the Java language. A listener object is able to monitor a communication channel, and when a tuple matching any template specified in the template set is emitted in the space, the listener is notified. The operation returns an URI identifying the subscription if it is successfully registered in the space.
unsubscribe	URI subscription, [URI transactionID]	Boolean result	This cancels the active subscription with the given ID. It returns true if the subscription has been successfully cancelled.
createTransaction	String type	URI transactionID	Type can be “local” or “shared”. The idea is to support both transactions which are local to a client (URI is only known to the client) and are shareable with others (for distributed transactions). The means for clients to acquire references to shared transactions is outside of the API.
getTransaction	URI transactionID	Boolean result	Used to join in a shared transaction with other clients. If true is returned, the client now shares in this transaction once it is begun until it is committed or rolled back. If false is returned, the system has not been able to share this transaction.

beginTransaction	URI transactionID	Boolean result	Begins the transaction. All subsequent interactions by all agents sharing this transaction referring to this transactionID are handled transactionally, i.e. 'all or nothing'. If false is returned, the transaction could not be begun by the system.
commitTransaction	URI transactionID	Boolean result	commits all interactions made within this transaction in the space. If false is returned, the transaction could not be committed by the system.
rollbackTransaction	URI transactionID	Boolean result	rolls back all interactions made within this transaction in the space. If false is returned, the system could not roll back the transaction.
create	URI parentSpace, [URI transactionID]	URI space	This creates a new space in Triple Space as a child of the space given by the parentSpace URI. If no parent space is specified, the new space is directly a child of the global Triple Space. The system allocates a new URI to this space and returns it to the client.
destroy	URI space, [URI transactionID]	Boolean result	This destroys the space identified by the given URI. We expect this also destroys all child spaces of this space.

As previously mentioned, some abstract error types are provided here which can be used in a TS API implementation to provide a Triple Space client with clear information about the reasons for failure of an operation. Given that a secure system should not provide clients with too much information (as failure of an operation can also indicate an attempt at inproper access) we restrict ourselves to error types so that a client can differentiate between possible reasons for failure. This is intended as an initial list to cover all possible error situations. In concrete implementations, the programming language and protocol used will generally support most of these error types, removing the requirement to specify TripleSpace-specific errors (e.g. Java's MalformedURIException replaces the abstract InvalidURIError).

- *InvalidOperationError* indicates that the provided operation is not supported by the TS API.
- *InvalidParameterError* indicates that the parameters provided by the operation do not correspond to what has been defined in the API. This may be due to additional or missing parameters, or the use of the wrong datatype with a parameter.
- *UnauthorizedOperationError* indicates that the client does not have authorization to execute this operation on the Triple Space. This can be caused by the

client not being authenticated at the space, or that the authenticated agent's allocated rights does not include the given operation.

- *InvalidURIError* indicates that an URI provided in a parameter of the operation was not a syntactically valid URI.
- *InvalidGraphError* indicates that the RDF graph provided as a parameter of the operation is not valid RDF.
- *InvalidSpaceError* indicates that the Space URI provided as a parameter of the operation is not valid. The URI is either already being used for an existing space (in the case of "out") or the client does not have the rights to retrieve data from this space (in the case of "rd" or "in") or the space does not exist.
- *InvalidTransactionError* indicates that the Transaction URI provided as a parameter of the operation is not referring to a currently active transaction, or that the client is not participating in that transaction.
- *InvalidTemplateError* indicates that the Template provided as a parameter of the operation is not syntactically valid, according to the Template type (e.g. a SPARQLTemplate throws this error if it does not contain a valid SPARQL Query).
- *InvalidTransactionError* indicates that the given transaction type in the "createTransaction" operation is not recognised by the Triple Space.

In the case of timeouts, we consider any positive integer value as valid, a zero value as indicating no timeout on the operation (indefinite blocking) and ignore negative values.

5.7 Comments on operation semantics

Further to the API, it is important to clearly specify the behaviour of the Triple Space API operations.

5.7.1 out operations

'out' adds a tuple into the tuplespace. That tuple is visible to clients who are requesting data in the space to which the tuple belongs, or in parent spaces of that space. Tuples are unique, while the contained RDF statement may already exist in the space (so in the RDF level, the client may see a single statement). The tuple may also belong to a particular Named Graph, meaning it is now a member of a graph formed by the sum of all tuples in the space and its parent spaces which are marked to belonging to a Named Graph of the same URI.

5.7.2 rd (non-destructive read) operations

The non-destructive read operations are blocking and can be given a timeout parameter.

First case is that if the read operation is applied on a space, the metadata from the TS ontology is used to identify the physical data which represents the (virtual) space, e.g. to know which persistent stores are relevant. Otherwise, the operation applies to the global space, e.g. all data in persistent storage is queried. How to access those stores in order to only query over the data within the specified space will be dependant on the data structures used in the back-end storage, which is preferable to loading all tuples listed as belonging to a space in the TS ontology into an in-memory RDF graph. Then all read operations are passed to the matcher. Generally, the form of the Template will be used to determine which semantic matching approach can be taken. Different Templates can be defined in the API to use different semantic matching techniques, e.g. with or without inference, different levels of reasoning, which will be specified in T3.4. An open issue is how the matching step acquires the ontologies it needs - generally, we expect to query an inference model and that the appropriate ontologies can be identified and loaded into a reasoner to generate a complete inference model of the data. For now, we could expect to find an ontology for any RDF resource through a HTTP GET at its URI though in practice this is not so simple. A register of ontologies may be used to ensure that ontologies are only loaded when necessary.

Results of retrieval operations (rd/rda) may be extended by other triples according to rules applying to specific RDF constructs. Rather than return a RDF triple with a blank node on its own, other triples with the same blank node (in the subject position, where it was the object of the original statement, or in the object position, where it was the subject of the original statement) will also be returned with it.

Timeouts: both rda and rdg return a 'single' answer (a triple or a named graph, respectively) which must be returned within the specified timeout. Otherwise, by the timeout, the kernel needs to cancel internally the running query process, which also means informing the client of operation 'failure' (no tuple match found in time) and if relevant the storage layer if the query is being executed there. By rd the storage layer returns matches as they are found and a result graph is built by the kernel until the process finishes, either by no more matches being found or the timeout being reached. If there is no timeout, the process is active until the query returns a result (rda/rdg) or the query ends because there is no more data to search - this is determined by the query implementation.

Blocking: if the operation ends without a match being found (before the timeout), the process is blocked until a match is available (until the timeout is reached). The process is registered at the process manager with its Template. Only newly inserted tuples need to be examined for matches on the Template, though we need to consider that in the time since the query was made until it ended without a result, matching tuples may have been added. Hence a cache at least of recently added tuples in the kernel may be usable with a local reasoner to allow the process manager at intervals to check if blocked processes can find matches and hence be unblocked. By rda, the single match is returned, by rdg the entire named graph of the match needs to be retrieved, and by rd all matches within the cache at that interval are returned in a graph.

5.7.3 in (destructive read) operations

The only difference with 'in' operations should be that the results are also deleted from the tuplespace.

Both ’rd’ and ’in’ can retrieve inferred tuples, hence querying is also executed over an inference model. As inferred tuples do not exist explicitly in the tuplespace, their deletion is not possible and hence ’in’ acts just like a ’rd’ (no actual change in tuplespace). The client may not know that the returned tuple is inferred however, and it may be an option to allow Templates which specifically act only upon explicit data in the tuplespace in cases where clients want to avoid this issue (which could be called the ’multiple in’ problem, i.e. as inferred tuples aren’t actually removed, repeated ’in’s with the same template could keep on returning the same inferred tuple).

5.7.4 tuplespace operations

A ’create’ operation is handled by the Triple Space Manager, and depends on how the management layer is implemented. If the representation of the tuplespace is handled through the Triple Space metadata describing the structure of the tuplespace, this representation is altered to allow for a new (sub)space. A register of which spaces exist at the kernel is needed to validate out/rd/in operations applied to named spaces, as the space may not exist. This could be an internal query on the meta-pool for the existence of a particular space instance.

A ’destroy’ operation removes the named space from the tuplespace structure representation in the kernel, e.g. the statements about that space from the Triple Space metadata. Tuples in a destroyed space and its children will also be deleted (in the sense of an ’in’ operation on them).

5.7.5 transactional operations

Transactions are handled by a dedicated manager which ensures transactional properties on a set of operations sharing the same transaction (identified by URI). For Triple Space we do not expect full ACID-ity. The kernel maintains a register of active transactions and can allocate URIs to new transactions and begin them as an internal process of the transaction manager. Clients can access this register and see those transactions marked as shared, and hence acquire their URI and add themselves to that transaction process, allowing for the relevant security and trust aspects that have been applied. Through the transaction manager, the other kernel components see a single transaction process, even if multiple clients are concurrently executing operations in that transaction. Hence we expect that parallel operations in the same transaction need to be serialized. Finally, a transaction can either be committed or rolled back.

If we take optimistic transactions, to preserve concurrency in the kernel, the changes are only made on the public tuplespace together at transaction commitment so during the transaction other clients outside of the transaction do not see any changes being made (each transaction process may need its own snapshot of the portion of tuplespace being changed) and rollbacks are hence very inexpensive. It may be we can define different levels of transaction to allow for different cases, such as transactions in which changes are seen before committal or in which no snapshot is made (so that no failure can be identified until committal time).

5.7.6 subscription operations

Subscriptions are handled by the process manager which maintains a register of active subscriptions and can allocate URIs to new subscriptions. We expect it will function like blocked processes, optionally making already a query over the local cache once a subscription is added, and then checking at intervals if any matches have been inserted into the tuplespace. Unlike blocked processes, a subscription process ends only once an 'unsubscribe' is received.

6 Semantic Linda and interaction patterns

A Triple Space will serve as a means to connect heterogeneous applications and services [43]. In this context, heterogeneity not only refers to the usage of different data formats but also to the integration of systems of different kinds of communication styles, e.g. Linda coordination, message queuing, and service oriented architectures (c.f. [16, 12]). As a consequence, the Triple Space must provide an interface powerful enough to adapt to and integrate these different styles without constraining the possible interactions.

The aim of this chapter is to tie together semantic Linda, presented in the previous chapter, with typical interaction patterns which clients will use in Triple Space. As the primary driving force behind Linda is the idea of coordination, many interaction patterns may be ideally be replaced by coordination mechanisms. Interaction patterns must be closely related to the features of the tuplespace model. Hence, we will identify which patterns are relevant for Triple Space and note if the expressiveness of semantic Linda is sufficient to support them.

6.1 Interacting Partners

Interaction patterns are realised by interacting partners. TripCom strives for a potentially global infrastructure which is necessarily distributed and structured. This means that we have to consider at least two types of partnerships.

- **Triple Space clients interacting with Triple Space kernels**: A Trile Space client invokes requests and uses the result for its own purpose.

  Reflecting on the use cases presented in WP8B, an Triple Space client could be a hospital application that connects to the Triple Space that serves the European Patient Summary (EPS). But it could also be a special browser that supports the underlying Triple Space protocol and allows to retrieve and to present semantic information in a generic, application independent way.

  Triple Space kernels embody the Triple Space and its capabilities: A Triple Space kernel manages semantic content.

- **Triple Space kernels interacting with other Triple Space kernels**: We envisage the whole Triple Space to be composed of many Triple Space kernels that form a Triple Space "fabric". The structure and the rules governing this fabric are not defined at this point in time. Basically, the communication between Triple Space kernels serves to distribute information or to aggregate information from/to a single client - to/from multiple client.

  Distribution and aggregation of information in a fabric of Triple Space kernels can follow two basic strategies which are both used today by Web servers on intranets as well as on the public internet:

  – The aggregation is performed by the user via redirection or computed/scripted redirection. A very popular and sophisticated form of this aggregation type is the so-called "mashup" technology that combines the content from multiple Web content providers.

- The aggregation is performed through a hierarchical arrangement of portal and portlet servers. The Triple Space client contacts only the top-level portal server. This type is popular in enterprises where security and controlled/audited access is of primary concern.

In analogy to the WWW, Triple Space will probably use both basic strategies to operate a distributed fabric of Triple Space kernels, e.g. we might expect clients to access multiple Triple Space kernels as well as Triple Space nodes to engage other clients in the execution of a function.

6.2 A Short Comment On Architecture

From the point of view of the architecture, the following requirements can be added:

- Whatever the type of interacting partners and the interaction patterns used, we would request that the communication interfaces remain essentially the same. The same interface of a Triple Space kernel should be able to handle the communication with another Triple Space kernel and with a Triple Space client.
- Interaction patterns should leverage on the advantages offered by the Linda principles. This means, that complex interaction patterns based on message exchanges should be replaced by coordination and cooperation mechanisms based on the application of Linda principles.

6.3 Semantic Linda and Interaction Patterns

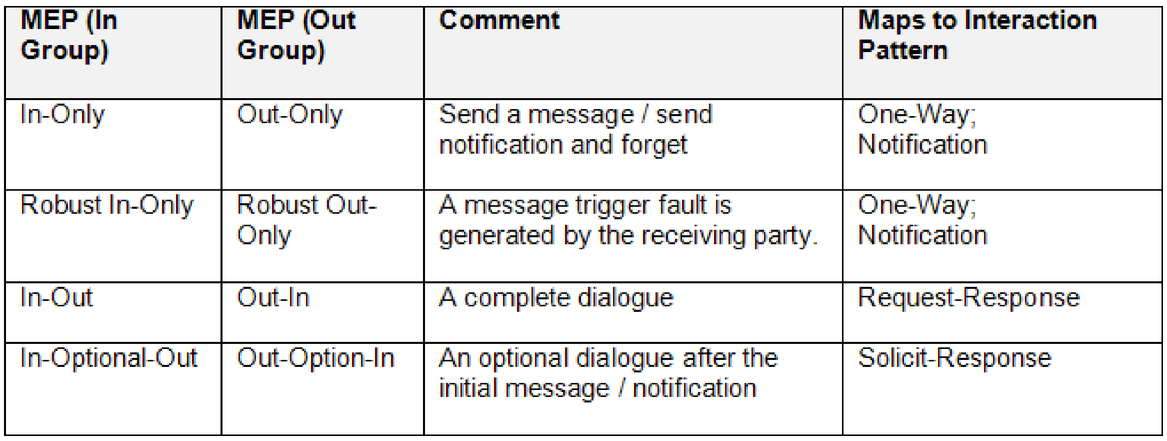
Chapter 5 introduced the extension of Linda for Triple Space which we have named semantic Linda.

In order to tie this coordination model and interaction patterns together, we need to introduce communication as the vehicle that connects the interacting partners. One way of doing this is to take the fundamental Web service interaction patterns defined in the Web service Description Language (WSDL). While version WSDL 1.1. [44] only supports four basic types of interaction patterns (one-way, request-response, solicit-response and notification), WSDL 2.0 [45] supports arbitrary complex interaction patterns called Message Exchange Patterns (MEPs), which are referred by an unique identifier[1]. MEP's are specific enough to describe communication while remaining sufficiently abstract allowing to avoid the presumption of specific communication mechanisms (like transports, bindings, interfaces, etc.)

The standard MEP's defined in the WSDL-2.0 specification are divided in two symmetric groups depending on which partner initiated the communication in Figure 6.1.

In Figure 6.2, we map Linda primitives to MEP's and to interaction patterns.

[1]see `http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327` for details)

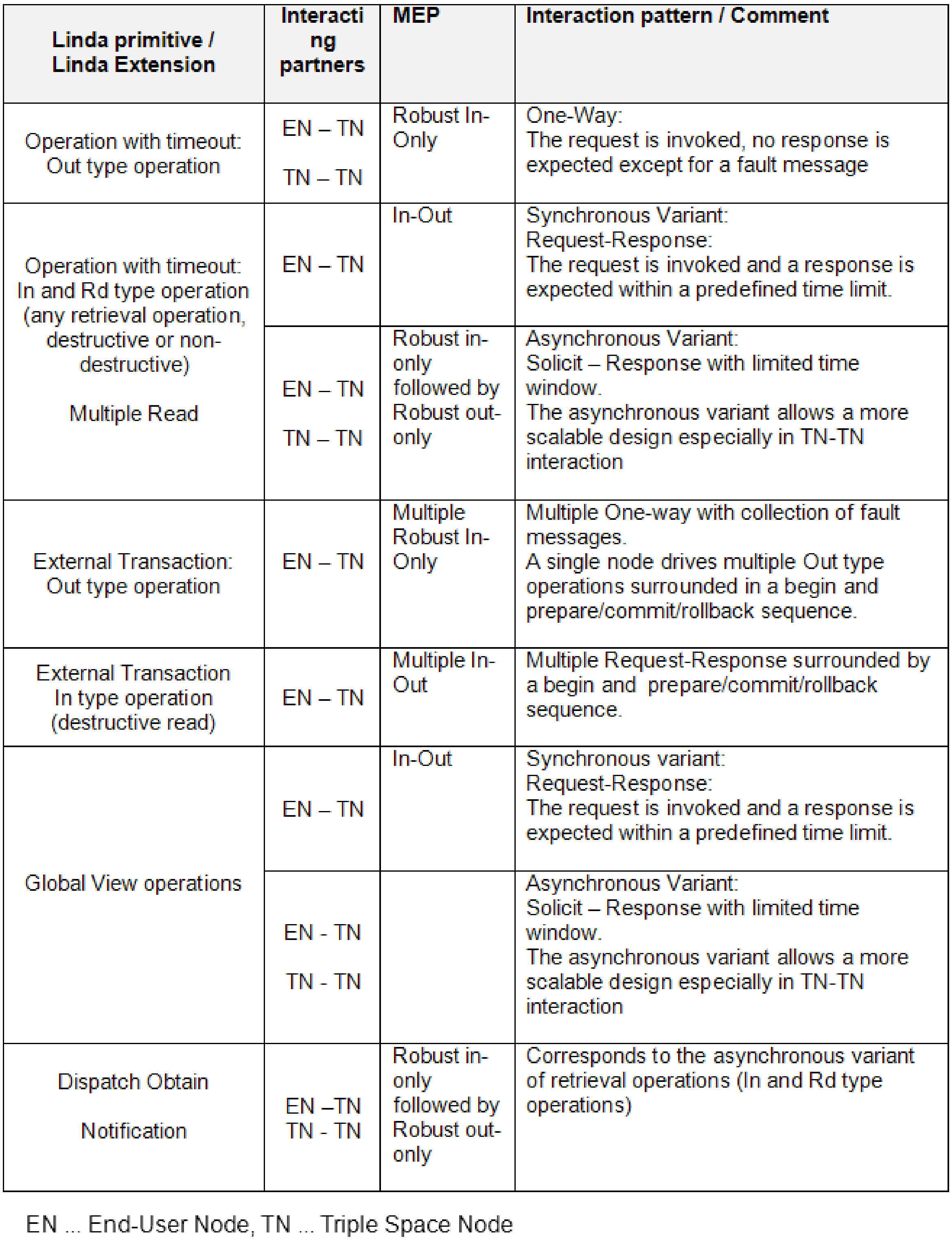


MEP (In Group)	MEP (Out Group)	Comment	Maps to Interaction Pattern
In-Only	Out-Only	Send a message / send notification and forget	One-Way; Notification
Robust In-Only	Robust Out-Only	A message trigger fault is generated by the receiving party.	One-Way; Notification
In-Out	Out-In	A complete dialogue	Request-Response
In-Optional-Out	Out-Option-In	An optional dialogue after the initial message / notification	Solicit-Response

Figure 6.1: Message Exchange Patterns of WSDL 2.0

6.4 Conclusion

Looking at table 6.2, we can draw the following conclusion: the semantic Linda API as proposed does allow clients to use the interaction patterns given in the WSDL 2.0 specification. Given our expectation that these patterns represent the vast majority of communication styles used by Triple Space clients (which will also typically be Web services), semantic Linda is an expressive enough coordination model for Triple Space.



Linda primitive / Linda Extension	Interacti ng partners	MEP	Interaction pattern / Comment
Operation with timeout: Out type operation	EN – TN TN – TN	Robust In-Only	One-Way: The request is invoked, no response is expected except for a fault message
Operation with timeout: In and Rd type operation (any retrieval operation, destructive or non-destructive) Multiple Read	EN – TN	In-Out	Synchronous Variant: Request-Response: The request is invoked and a response is expected within a predefined time limit.
	EN – TN TN – TN	Robust in-only followed by Robust out-only	Asynchronous Variant: Solicit – Response with limited time window. The asynchronous variant allows a more scalable design especially in TN-TN interaction
External Transaction: Out type operation	EN – TN	Multiple Robust In-Only	Multiple One-way with collection of fault messages. A single node drives multiple Out type operations surrounded in a begin and prepare/commit/rollback sequence.
External Transaction In type operation (destructive read)	EN – TN	Multiple In-Out	Multiple Request-Response surrounded by a begin and prepare/commit/rollback sequence.
Global View operations	EN – TN	In-Out	Synchronous variant: Request-Response: The request is invoked and a response is expected within a predefined time limit.
	EN - TN TN - TN		Asynchronous Variant: Solicit – Response with limited time window. The asynchronous variant allows a more scalable design especially in TN-TN interaction
Dispatch Obtain Notification	EN –TN TN - TN	Robust in-only followed by Robust out-only	Corresponds to the asynchronous variant of retrieval operations (In and Rd type operations)

EN ... End-User Node, TN ... Triple Space Node

Figure 6.2: Linda to MEP mappings

7 Triple Space Prototype

In order to validate the Triple Space API specification, we implement a prototype of the Triple Space kernel which supports the specification. In this chapter, we outline the prototype implementation agreed upon by task partners and describe the decisions made regarding the implementation of the API operations and open issues. Finally, we describe the current implementation.

7.1 The prototype implementation

The current prototype is regarded as a proof-of-concept for the Triple Space Service Technology and will lack several general features which are to be determined later on in the project course:

- The usage of the full TS ontology being developed parallel in WP2;
- Distribution of kernels and underlying storage, to be developed in WPs 1 and 2;
- Storage and matching of semantic information beyond that of RDF(S) expressiveness, to be defined in WPs 2 and 3;
- Security and trust aspects to be specified in WP 5.

Rather the prototype will serve as a first stage testbed and demonstrator for the initial specifications produced in the TripCom project, which will cover:

- The Triple Space API and coordination model as defined by T3.1
- The representation of RDF data in tuples as defined by T2.1
- The structure of RDF-containing tuplespaces as defined by T2.1
- The storage model and architecture based on persistent RDF stores as defined by T1.2

Finally, this work will occur parallel to T6.2 and is interdependent with it. Through this implementation plan, we need to identify the components of the Triple Space architecture relevant to the Triple Space kernel and storage (WPs 1, 2 and 3). Through the implementation work, we expect to revise and refine these components, particularly in terms of our requirements and their responsibilities. Hence throughout the implementation task we communicated with T6.2 in a bidirectional manner, and input our component descriptions to D6.2.

7.1.1 High level design

The diagram presented in Figure 7.1 provides a high-level definition of the different components in the prototype. It is deliberately kept simple, as we focus in this first step on a single (non-distributed), local (non-remote) prototype of the TripCom specifications, rather than an attempt at creating the Triple Space system itself. A Triple Space architecture will be detailed in the Reference Architecture defined in WP6, which is tasked with the implementation of Triple Space.

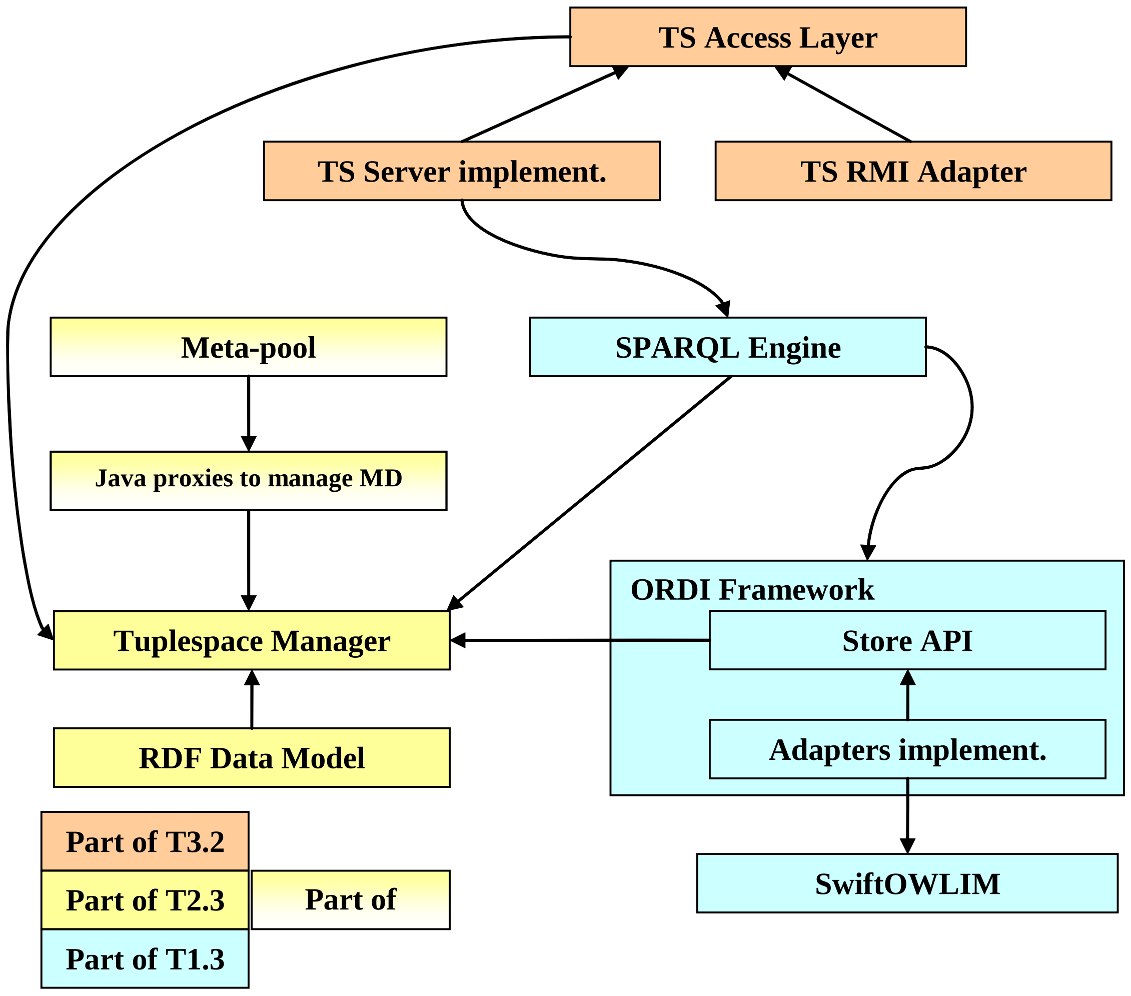


Figure 7.1: High level prototype design

High-level component description:

The TS API provides the public interface of Triple Space to clients and accepts the operations specified in the API given in this deliverable as part of the efforts of WP3. The prototype implementation extends the API interface to realise the operations as they have been defined from the point of view of the client and uses extensive synchronization and is multi-thread safe, i.e. the system can run concurrent operations on the tuplespace in different threads.

Tuple Space Model is the specification of data models for tuples and tuplespaces provided in WP2. It uses the XVSM platform from TU Vienna as a base tuplespace implementation. RDF data types are supported and are transparently mapped to the XVSM Entry Java objects. XVSM NamedContainers are mapped to the Spaces defined in the tuplespace model. A full description is available in TripCom D2.1, Implementation section.

ORDI Framework, developed by Ontotext, is the proposed basis for data storage as being developed in WP1. It is a black-box with a well-defined interface for providing persistent back-end storage of RDF data and a SPARQL query engine to allow for RDF-based querying over the data set. An adapter is implemented on top of ORDI for synchronization of the data between the storage and the tuplespace model (XVSM) as well as to handle all query requests and results using the XVSM co-ordination mechanisms.

The arrows indicate the general data flow between these three high-level components. The reader should also refer to D6.2 where the components of Triple Space are described in further detail, as part of the Triple Space Reference Architecture.

7.2 The Prototypical Implementation

A first Triple Space prototype implementing the API specification outlined in this deliverable is available at `http://sourceforge.net/projects/tripcom`. The prototype is local, running in a single JVM, with querying and storage through integration with the ORDI framework provided by Ontotext and tuplespace co-ordination and management provided by the XVSM platform of TU Vienna.

For the prototype we realized the following functionality:

- Storage of RDF triples and (named) graphs
- Management of RDF data in tuples and spaces
- Support for the Triple Space API
- Semantic matching of RDF through a SPARQL engine, using triple patterns as Templates in the API operations

7.2.1 Storage of RDF triples and (named) graphs

The core prototype will use Sesame openRDF for the RDF data representation in the Tuplespace Manager and store the tuplespace data in XVSM as Java objects. We do not query over this data as XVSM does not incorporate inherent support for semantic data: it may be an option to retrieve RDF data objects from XVSM and then create inference models from them with a reasoner component. However, it is more efficient to reason directly at the storage layer. Hence for the prototype, we store RDF in both XVSM and ORDI: a listener component in the store catches events in XVSM (addition or removal of triples) and synchronize the persistent storage.

7.2.2 Management of RDF data in tuples and spaces

XVSM provides support for tuple and tuplespace management including partitioning of the global tuplespace. When persistent storage is available, we may access data in the persistent storage directly. XVSM is usable then as a local storage for the TS metadata (meta-pool), the kernel cache (recent operations over the kernel) and other kernel data (e.g. the security and distribution data). Management will then be largely querying the TS metadata over XVSM to determine how to carry out operations, which is the direction taken in the Reference Architecture of WP6.

7.2.3 Support for the Triple Space API

We build on top of the relevant (XVSM, ORDI) access methods, adding Triple Space specific code, e.g. each operation spawns a thread within the kernel which is monitored and blocking the thread until a result is found or timeout is reached. Operations are applied to XVSM and 'listened to' by the storage framework, so that e.g. SPARQL queries are executed and the results returned to the access layer.

The proposal for the implementation of timeouts is that they begin when the operation begins to be handled in the kernel (a thread is created). When the timeout is reached while the thread is still active, the kernel terminates the thread and sends a timeout error to the client. In a RMI implementation, we define the API methods as throwing TimeoutExceptions. Note that rd and in operate slightly differently, when the timeout is reached the results collected so far (through result "streaming" from the ORDI framework) are returned to the client and then the thread is terminated without an exception.

The implementation of optimistic transactions over the global tuplespace is a non-trivial research challenge which has been identified by the implementation task. While possibilities to realize such transactional functionality in Triple Space will continue to be discussed in a separate document, with the intention to add such transaction support to later prototypes and then to the Triple Space implementation in WP6, for the next phase of the prototype we will add transactional support on top of a transaction management API being provided in the XVSM system.

7.2.4 Semantic matching of RDF

The Template passed in the retrieval operation is mapped into a SPARQL query and executed by the SPARQL query engine across a RDF data set in the persistent storage. In the prototype, two types of Template are defined: a single triple pattern template which maps to a simple SPARQL query or a SPARQL template which accepts any valid SPARQL query as a string.

8 Conclusion

This deliverable reports on the results of the tasks T3.1 and T3.2 in TripCom.

Firstly, after a complete and thorough study of the state of the art and requirements analysis in the Linda co-ordination language and combined with requirements collection from the TripCom scenarios, we were able to draft an API specification for Triple Space. This specification, following discussions on open issues and a resulting evolution of the specification, has been finalized for the first implementation of a Triple Space prototype. having been validated against a standard set of interaction patterns.

The prototype is available as an open source project in Sourceforge. We reported on the implementation decisions made and how we chose to handle particular issues, such as the semantics of the coordination operations and other open issues. The prototype will be further developed in the implementation workpackages (WP2 and WP3). Particularly, distribution and richer query support are planned for the next phase of the implementation, as well as the incorporation of transactions. The storage component will be further developed in parallel in WP1. Dedicated protocols for remote access to the Triple Space will be specified in due course in WPs 4 and 6 to allow Web service and Web-style remote access. Security and trust is specified orthogonally in WP5. Finally, these implementations are inputs to the Triple Space architecture, which will be implemented in WP6.

The implementation work is done in close co-ordination with the other workpackages to ensure full uptake of our results and will be evaluated and iteratively defined in the next phases. As a result, it serves as the first core specification and prototype of Triple Space, and an important basis for the work to come.

References

[1] B. Anderson and D. Shasha. Persistent linda: Linda + transactions + query processing, 1991.

[2] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.

[3] Nadia Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro. Coordination models: a guided tour. pages 6–24, 2001.

[4] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[5] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Comparing three semantics for Linda-like languages. *Theoretical Computer Science*, 240(1):49–90, 2000.

[6] Nadia Busi and Gianluigi Zavattaro. On the expressiveness of event notification in data-driven coordination languages. *Lecture Notes in Computer Science*, 1782:41–55, 2000.

[7] Chris Bussler. A minimal triple space computing architecture. In *Proc. of the WIW 2005 Workshop on WSMO Implementations*, 2005.

[8] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[9] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. XML dataspaces for mobile agent coordination. In *SAC (1)*, pages 181–188, 2000.

[10] Nicholas Carriero and David Gelernter. Case studies in asynchronous data parallelism. *Int. J. Parallel Program.*, 22(2):129–149, 1994.

[11] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *J. Web Sem.*, 3(4):247–267, 2005.

[12] Dario Cerizza, Emanuele Della Valle, doug foxvog, David de Francisco, Reto Krummenacher, Henar Munoz, Martin Murth, and Elena Paslaru-Bontas Simperl. State of the art and requirements analysis for sharing health data in the ts, 2007.

[13] Paolo Ciancarini and Davide Rossi. Jada - Coordination and Communication for Java Agents. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 213–228. Springer-Verlag: Heidelberg, Germany, 1997.

[14] Paolo Ciancarini, Robert Tolksdorf, and Fabio Vitali. The world wide web as a place for agents. In *Artificial Intelligence Today*, pages 175–193. 1999.

[15] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Coordination and access control in open distributed agent systems: The tucson approach. In Porto and Roman [34], pages 99–114.

[16] David de Francisco Marcos, Daniel Wutke, Daniel Martin, Andreas Harth, and Martin Murth. Requirements analysis and architecture profile for eai applications, 2007.

[17] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *Selected Areas in Cryptography*, pages 169–177, 1998.

[18] Enrico Denti, Andrea Omicini, and Vladimiro Toschi. The luce coordination technology for mas design and development on the internet. In Porto and Roman [34], pages 305–310.

[19] eva Khn, editor. Nova Science Publishers, 2001.

[20] L.J.B. Nixon F. Martn-Recuerda and E. Paslaru Bontas. Knowledge Web Deliverable D2.4.8.1 Technical and ontological infrastructure for Triple Space Computing, January 2006.

[21] Dieter Fensel. Triple-Space Computing: Semantic Web Services Based on Persistent Publication of Information. In Finn Arve Aagesen, Chutiporn Anutariya, and Vilas Wuwongse, editors, *Proc. of the IFIP Int'l Conf. on Intelligence in Communication Systems*, volume 3283 of *Lecture Notes in Computer Science*, pages 43–53. Springer-Verlag, November 2004.

[22] Dieter Fensel, Reto Krummenacher, Omair Shafiq, eva Kühn, Johannes Riemer, Ying Ding, and Bernd Draxler. TSC - Triple Space Computing. *e&i Elektrotechnik und Informationstechnik*, 124(1/2), Febuary 2007.

[23] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[24] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[25] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[26] David Gelernter and Suresh Jagannathan. *Programming linguistics.* MIT Press, Cambridge, MA, USA, 1990.

[27] F. Martin-Recuerda. Towards cspaces: A new perspective for the semantic web. In *1st International IFIP WG 12.5 Working Conference on Industrial Applications of Semantic Web (IASW '05)*, Jyvaskyla, Finland, 2005. Springer IFIP Book Series.

[28] Iain Merrick and Alan Wood. Coordination with scopes. In *Proceedings of SAC'00*, pages 210–217. ACM Press, 2000.

[29] Naftaly H. Minsky and Jerrold Leichter. Law-governed linda as a coordination model. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 125–146, London, UK, 1995. Springer-Verlag.

[30] Naftaly H. Minsky and Jerrold Leichter. Law-governed linda as a coordination model. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 125–146, London, UK, 1995. Springer-Verlag.

[31] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, July 2006.

[32] Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications.* Springer, 2001.

[33] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *761*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

[34] António Porto and Gruia-Catalin Roman, editors. *Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Limassol, Cyprus, September 11-13, 2000, Proceedings*, volume 1906 of *Lecture Notes in Computer Science*. Springer, 2000.

[35] A. Rowstron and A. Wood. Bonita: a set of tuple space primitives for distributed coordination. In *Proc. HICSS30, Sw Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[36] Antony I. T. Rowstron. WCL: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.

[37] Antony I. T. Rowstron and Alan Wood. Solving the linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[38] D. Shasha and J. Turek. Beyond fail-stop: Wait free serializability and resiliency in the presence of slow-down failures. Technical Report 514, Sep l990.

[39] R. Tolksdorf, L. Nixon, and E. Paslaru Bontas. A Conceptual Model for Semantic Web Spaces. Technical Report TR-B-05-14, Free University of Berlin, September 2005.

[40] R. Tolksdorf, L. Nixon, and E. Paslaru Bontas. Towards a tuplespace-based middleware for the Semantic Web. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)*, 2005.

[41] R. Tolksdorf, L. Nixon, E. Paslaru Bontas, D. M. Nguyen, and F. Liebsch. Enabling real world Semantic Web applications through a coordination middleware. In *Proceedings of the ESWC05*, 2005.

[42] Robert Tolksdorf, Elena Paslaru Bontas, and Lyndon J. B. Nixon. A coordination model for the semantic web. In *Proceedings of the 2006 ACM Symposium*

on Applied Computing (SAC), Dijon, France, April 23-27, 2006, pages 419–423, 2006.

[43] Tripcom. Triple space communication, Annex I. – Description of Work. FP6 - 027324, November 2005.

[44] W3C. Web services description language (wsdl) 1.1. W3C Note, March 2001.

[45] W3C. Web services description language (wsdl) version 2.0 part 1: Core language. W3C Candidate Recommendation, March 2006.

[46] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel Alexander Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.