Triple Space Computing: Adding Semantics to Space-based Computing

Johannes Riemer[1], Francisco Martin-Recuerda[2], Ying Ding[3], Martin Murth[1], Brahmananda Sapkota[2], Reto Krummenacher[2], Omair Shafiq[2], Dieter Fensel[2] and Eva Kühn[1]

[1] Institute of Computer Languages, Vienna University of Technology, Vienna, Austria
{mm,jr, eva}@complang.tuwien.ac.at

[2] Digital Enterprise Research Institute, University of Innsbruck, Innsbruck, Austria
{francisco.martin-recuerda, brahmananda.sapkota, reto.krummenacher, omair.shafiq, dieter.fensel}@deri.org

[3] Electronic WebService GmbH, Innsbruck, Austria

Abstract. Triple Space Computing (TSC) is a very simple and powerful paradigm that inherits the communication model from Tuple Space Computing model and projects it in the context of the Semantic Web. In this paper, we propose Triple Space Computing as a new communication and coordination framework for Semantic Web and Semantic Web Services. We have conducted wide-covered state of the art studies in related fields and identify the current status and the value added by TSC. Based on this, we propose the overall architecture of TSC and the interactions among different components.

1 Introduction

Triple Space Computing [1] is a powerful paradigm that inherits the communication model from Tuple Space Computing and projects it in the context of the Semantic Web. Instead of sending messages forward and backward among participants, like most of today's web service-based applications do, triple-based applications just use a simple communication based on reading and writing RDF triples in a shared persistent and semantically described information space. Triple Space Computing as a new paradigm for coordination and communication compliant with the design principles of the Web, thus provides a major building block for the Semantic Web and for interoperation of Semantic Web Services.

The current communication paradigm of Web Services is message-oriented. SOAP as a communication technology for XML implies messaging, WSDL defines messages that a Web Services exchanges with its user, and literally all ongoing research efforts around Semantic Web Services rely on these technologies. Although most message-based technologies have reached a level of maturity, there are still some open issues. For instance, messaging technologies might not be scalable. Triple Space

Computing (TSC) aims to overcome the deficiencies of message-based communication technologies by adding semantics to Tuple space computing. TSC is based on the evolution and integration of several well-known technologies: Tuple Space Computing [2], Shared Object Space, Semantic Web and in particular RDF Schema. However, [3] reports some shortcomings of the current tuple space models. They lack any means of name spaces, semantics, unique identifiers and structure in describing the information content of the tuples. This tuple space provides a flat and simple data model that does not provide nesting. TSC takes the communication model of Tuple Space Computing, wherein communication partners write the information to be interchanged into a common space and thus do not have to send messages between each other, TSC enhances this with the semantics required for Semantic Web enabled technologies.

The basis for prototype development is based on Corso (Coordinated Shared Objects) system [4]. Corso is a platform for the coordination of distributed applications in heterogeneous IT environments that realizes a data space for shared objects. Corso offers maximum scalability and flexibility by allowing applications to communicate with one another via common distributed persistent „spaces of objects". For testing and validating the TSC technology with special attention to the support for Semantic Web Services, we will integrate this system in the Semantic Web Service Environment WSMX[1], which is the reference implementation of the Web Service Modeling Ontology WSMO[2]. Thereby, the TSC technology will be aligned with emerging technologies for Semantic Web Services. By providing the basis for a new communication technology for the Semantic Web, TSC will provide a significant contribution to international research and development efforts around the Semantic Web and Semantic Web Services.

In this paper, we report some of the progresses. First, we provide the summarized current state of the arts of Space-based Computing. Based on this, we present the overall architecture of TSC which is mainly focusing on introduction of different components involved in a Triple Space environment and the connections and interaction among these components. Finally we mention some potential future works.

2 State of the art of Space-based Computing

Although the space-based computing paradigm is a relatively new concept, a considerable amount of technology has already been created to support such paradigm. In the following we briefly present the state of the art in this field.

Linda was developed by David Gelernter in the mid-80s at Yale University. Initially presented as a partial language design [2], it was then recognized as a novel communication model on its own and is now referred to as a *coordination language* for parallel and distributed programming [5]. Coordination provides the infrastructure for establishing communication and synchronization between activities and for spawning new activities. There are many instantiations or implementations of the

[1] www.wsmx.org
[2] www.wsmo.org

Linda model, embedding Linda in a concrete host language. Examples include C-Linda, Fortran-Linda and Shared-Prolog. Linda allows defining executions of activities or processes orthogonal to the computation language, i.e. Linda does not care about, how processes do the computation, but only *how* these processes are created. The Linda model is a *memory* model. The Linda memory is called *tuple space* and consists of logical tuples. There are two kinds of tuples. Data tuples are passive and contain static data. Process tuples or "live tuples" are active and represent processes under execution. Processes exchange data by writing and reading data tuples to and from the tuple space.

Multiple extensions of Linda have been proposed, most of which introduce new operations. **Multiple tuple spaces** have been proposed to improve modularity in Linda [6]. Collect [7] and copycollect [8] have been proposed to solve the multiple rd and related problems. **Notifications (notify)** have been introduced by JavaSpaces[3] to embed reactive programming into the shared data space. **Transactions** are proposed by PLinda [9] to unify two different kinds of parallelism. **Asynchronous process-to-space communication** is proposed in Bonita [10] to improve performance compared to prior Linda implementations.

JavaSpaces technology from Sun Microsystems provides a platform for simple development of distributed, Java based systems. It is designed to help the programmer to solve two related problems: distributed persistency and the design of distributed algorithms. Using JavaSpaces, distributed processes can communicate, share objects and coordinate their activities by reading and writing tuples (called entries) from and to persistent spaces. JavaSpaces are based on Jini[4], an open software architecture that enables the creation of network-centric solutions which are highly adaptive to change. JavaSpaces make use of some Jini technologies such as *entries*, *transactions*, *leases* and *events*.

TSpaces[5], developed at the IBM Almaden Research Centre, is "a network communication buffer with database capabilities which enables communication between applications and devices in a network of heterogeneous computers and operating systems". It is a tuple space implementation in Java, which supports, besides the basic Linda like tuple space operations for reading and writing tuples, database services, URL-based file transfer services, access control and event notification services. Since TSpaces is implemented in Java, it automatically possesses network ubiquity through platform independence. The small memory foot print makes TSpaces attractive for small, embedded systems and ubiquitous computing.

GigaSpaces[6] offer a family of products around the flagship "GigaSpaces Enterprise Application Grid". All of them are based on a space-based, distributed shared memory core. The main concept of this core is a space, which can be accessed via the JavaSpaces interface. Additionally there are some extensions to the JavaSpace API, alternative non-JavaSpace APIs and clustering features supporting replication, failover and load balancing. GigaSpaces provide a persistent JavaSpaces implementa-

[3] http://java.sun.com/products/komo/specs
[4] http://www.jini.org
[5] http://www.almaden.ibm.com/cs/TSpaces/html/ProgrGuide.html
[6] http://www.gigaspaces.com

tion with some extensions, most of which fall into the categories batch processing (e.g. write multiple entries at once), inplaces updates (updates in the space without having to take and re-insert an entry to the space) and administration activities (e.g. starting and stopping spaces).

The difference to tuple spaces is that Corso do not work the tuples but with data objects. Efficient and physically distributed data structures can be built up in the Corso space using unique object references (OIDs). Objects that are written into the space can be arbitrarily structured (e.g. records, arrays, tuples, etc.) and sophisticated replications techniques keep the space consistent. The Corso system also provides notification facilities. It is able to notify to the corresponding users of an object that a resource is available. Acting as a virtual shared memory, Corso is used for communication between and synchronization of parallel and distributed processes. Corso is not only a pure memory model. It also supports complex coordination patterns on the shared objects through an advanced transaction and process model. Transactions and processes define the recoverability of the shared data objects and of the computations on them. The Corso middleware hides the heterogeneity and puts a strong focus on security. The virtual space can be divided into secure subspaces connecting trusted partners. It provides secure distributed data spaces as well as individual security and privacy policies by using flexible plug-in mechanisms like login authentication, access authorization, and communication encryption. Corso has already been successfully applied in several industry projects.

As a relevant part of the *Service Oriented Architecture* (SOA), *notification* is expected to play an essential role in the development of asynchronous, loosely-coupled and dynamic systems, where entities receive messages based on their registered interest in certain occurrences or situations. TSC project will provide extensions for tuple space coordination model to support publish-subscription capabilities.

The publish-subscribe paradigm is an asynchronous, many-to-many communication for distributed systems [26]. The model defines two main roles for participants: **source**, which generates notifications; and **sink**, which expresses its interest in concrete event notifications or pattern of event notifications. Typically a source can act as a producer and as a publisher. **Producers** encode information into notification messages, while **publishers** make accessible these notifications. Similarly, a sink can act as a subscriber and as a consumer. **Consumers** express interest in concrete notifications and consume those notifications when corresponding information is published by a source. **Subscribers** are responsible for registering the consumer' interests.

In the case of a loosely-coupled configuration (independent producers, publishers, subscribers and consumers), the publish-subscribe model de-couples the processes involved in information exchange in four orthogonal dimensions (partially adapted from [26]):

- **Space decoupling**: Producers and consumers can run in completely different computational environments as long as both can make access to the same event service, i.e., space-wise the processes are completely de-coupled
- **Reference decoupling**: the processes that interact through an event service do not need to know each other (anonymous). The notifications published by publishers are accessed by consumers indirectly. In general, notifications do

not include references to concrete consumers, and similarly consumers do usually not include specific references to producers.

- **Time decoupling**: the processes that interact through an event service do not need to be up at the same time (asynchronous). In particular, producers might generate some notifications while related consumers are not connected with the event service, and the other way around, consumers might get notifications while the original producers are not online.
- **Flow decoupling**: participants are not block while producing/receiving notifications. Consumers can receive a notification while performing some concurrent activity (i.e. through a 'callback'). Producers can produce notifications and continuous with their execution flow. In other words, main flows of producers and consumers are not affected for the generation or reception of notifications.

There is a long tradition in the use of these kinds of technologies in the areas of distributed objects, message oriented middleware (MOM) and Peer to Peer systems. Recently, two new specifications, **WS-Notification** [11] and **WS-Eventing** [12] bring the publish-subscribe communication paradigm once again into the fore. The publish-subscribe paradigm is an asynchronous, many-to-many communication paradigm for distributed systems [13]. A fundamental problem of most publish/subscribe systems, is how to match the interests of consumers with the available notifications generated by producers. Simple strings such as "Weather/Warnings" or complex XPath or SQL queries do not provide enough expressivity to perform a sophisticated matching of interests and data. However, in [14], a proposal for a Semantic Message Oriented Middleware based on DAML+OIL is presented to overcome this limitation. Some concrete implementation based on publish-subscribe paradigm are Gryphon [15], TIB/RV [16], Scribe [17], SIENA [18], Hermes [19] and so on. The relation of the TSC, Web and Tuple Space paradigms is shown in Fig. 1.

3 TSC Architecture

Semantically enabled tuplespaces can offer an infrastructure that scales conceptually on an Internet level. Just as Web servers publish Web pages for humans to read, tuplespace servers would provide tuplespaces for the publication of machine-interpretable data. Providers and consumers could publish and consume tuples over a globally accessible infrastructure, i.e., the Internet. Various tuplespace servers could be located at different machines all over the globe and hence every partner in a communication process can target its preferred space, as is the case with Web and FTP servers. This highlights many advantages for providers and consumers. The providers of data can publish it at any point in time (time autonomy), independent of its internal storage (location autonomy), independently of the knowledge about potential readers (reference autonomy), and independent of its internal data schema (schema autonomy) [20].

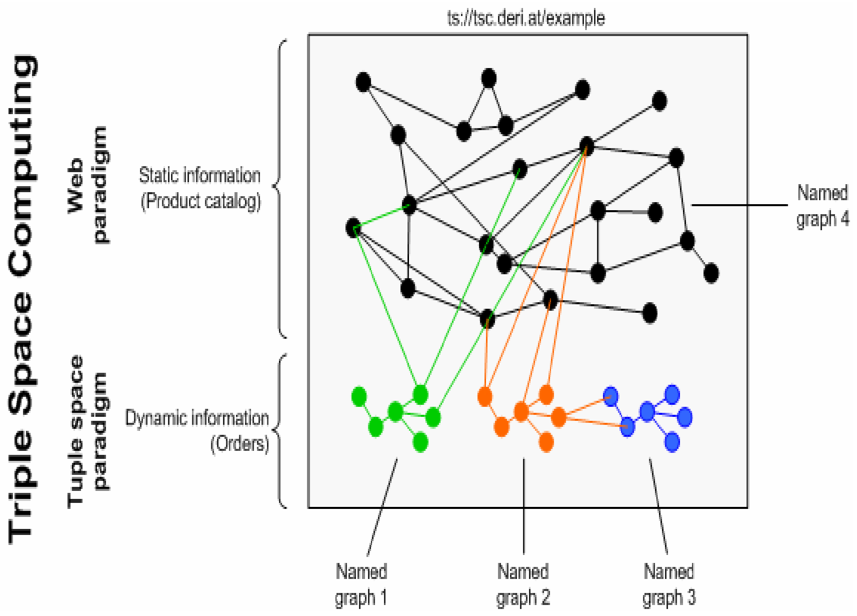

Fig. 1. TSC and related paradigms

3.1 Overall architecture

Like the Web, TSC aims to build a Triple Space Computing infrastructure based on the abstract model called REST (Representational State Transfer) [21]. The fundamental principle of REST is that resources are stateless and identified by URIs. HTTP is the protocol used to access to the resources and provides a minimal set of operations enough to model any application domain [21]. Those operations (GET, DELETE, POST and PUT) are quite similar to Tuple-Space operations (READ, TAKE and WRITE in TSpaces). Tuples can be identified by URIs and/or can be modeled using RDF triples. Since every representation transfer must be initiated by the client, and every response must be generated as soon as possible (the statelessness requirement) there is no way for a server to transmit any information to a client asynchronously in REST. Furthermore, there is no direct way to model a peer-to-peer relationship [22]. Several extensions of REST, like ARRESTED [22], have been proposed to provide a proper support of decentralized and distributed asynchronous event-based Web systems.

The limitations of REST to model asynchronous interaction motivated us to pay attention to Peer-to-Peer systems, and in particular super-peer configuration. P2P are decentralized, distributed, self-organized and capable of adapting to changes such as failure [19]. Although there are several open issues regarding scalability, shared resources management, security and trust, current efforts in the field (for instance [23]) are progressively overcoming these problems. Hybrid architectures that combine pure P2P and client/server systems are called super-peer systems [24]. This configuration drives into two-tiered system. The upper-tier is composed of well-connected and powerful servers, and the lower-tier, in contrast, consists in clients with limited computational resources that are temporarily available. Three kinds of nodes are identified in Triple Space architecture:

- **Servers** store primary and secondary replicas of the data published; support versioning services; provide an access point for light clients to the peer network; maintain and execute searching services for evaluating complex queries; implement subscription mechanisms related with the contents stored; provide security and trust services; balance workload and monitor requests from other nodes and subscriptions and advertisements from publishers and consumers.

- **Heavy-clients** are peers that are not always connected to the system. They provide most of the infrastructure of a server (storage and searching capabilities) and support users and applications to work off-line with their own replica of part of the Triple Space. Replication mechanisms are in charge to keep replicas in clients and servers up-to date.

- **Light-clients** only include the presentation infrastructure to write query-edit operations and visualize data stored on Triple Spaces.

Servers and heavy-clients are in charge of running Triple Space kernels (TS kernels). A TS kernel provides interfaces for participant to access Triple Spaces and implements storing of named graphs [27], synchronization of participants via transactions and access control to Triple Spaces. A Triple Space can be spanned by one or multiple TS kernels. If multiple TS kernels are involved, the kernel exchanges named graphs of the space in a consistent way. Participants are users and applications which use the Triple Space in order to publish and access information and to communicate with other participants via the Triple Space. Applications can be run in servers, heavy clients, and light clients. Since light clients are running in small devices like mobile phone, only small applications are expected to be run in those devices.

3.2 Triple Space Data Model and Operations Summary

A Triple Space contains data in form of RDF triples. Non-overlapping sets of triples are grouped into *named graphs*. Processes (participants) read, write and take named graphs to and from the Triple Space in the same way as tuples are read, written and taken in Tuple Spaces. The essential difference is that named graphs can be related to each other via the contained RDF triples. For example the object of a triple in one named graph can be the subject of a triple in another named graph. This way named graphs are not self-contained (as tuples in Tuple Spaces), but can build arbitrary RDF graphs to represent information. To make use of such nested triples, the TSC interaction model allows, in addition to the already mentioned operations, a query operation. This operation allows creating new RDF graphs out of the named graphs in a given space.

Named graphs can be used for several purposes in a Triple Space. One use is to publish semantic information to the Triple Space like information is published in the (semantic) Web. Named graphs can constitute ontologies (RDF vocabularies) in the Triple Space, e.g. a product catalog containing product descriptions and prices. Another use of named graphs is to let processes communicate and synchronize each other. For example processes can write named graphs representing orders to the space, which are picked up by processes which process the orders. Since named graphs can be inter-linked via RDF, orders can be related to ontologies, for example to a product catalog, published in the space, and thus be semantically enriched.

The operations available in TSC support both scenarios. *Read operations based on templates* can be used to discover and navigate through the information contained in the Triple Space. Mediation [20] based on RDF Schema allows to overcome heterogeneity of data. *Transactions enable* processes to be coordinated consistently. The overall, global Triple Space is partitioned into Virtual Triple Spaces, which are identified by a Triple Space URI. All Triple Space operations are performed against a certain Virtual Triple Space by specifying its URI. To publish information, the following operation is used:

write(URI ts, Graph graph): URI

As a result a named graph is inserted to the Triple Space *ts*, consisting of a generated name, which is a URI, and the given *graph*. The *generated name* is returned. Information contained in a Triple Space can be retrieved by *templates*, e.g. via the following operation:

read(URI ts, Template template): NamedGraph

It returns one *named graph* existing in the Triple Space, which matches *template* [20]. Further retrieval operations allow query data stemming from multiple named graphs (*query*), to block until intended data becomes available (*waitToRead*, *waitToQuery*) or to remove the retrieved data from the space (*take*). Concurrent access to a Virtual Triple Space by multiple participants needs to be managed in order to preserve the intended semantics of interactions [9]. Participants can agree on *common rules* which guarantee consistency. For example certain kinds of information may be *known as* being immutable. Other kinds of information may be agreed to be *locked* if some conditions hold. This kind of interaction between participants is referred as *cooperative parallelism*. Interacting participants must know and agree on certain rules.

Another way to handle consistent concurrent interactions is *transactions*. Transactions allow participants to concurrently access the Triple Spaces without having to agree on explicit rules. This kind of parallelism is called *competitive parallelism* and is well known, e.g. from databases. TSC supports transactions and provides operations to create, commit and abort transactions. A transaction identifier is used to bind a TSC operation to a certain transaction. A typical sequence of operations of a participant could be:

```
1  transactionID = createTransaction
2  namedGraph1 = read(tripleSpaceURI, transactionID, namedGraphURI)
3  namedGraph2 = take(tripleSpaceURI, transactionID, tempalte)
4  graph3 = //process graphs
5  nameURI = write(tripleSpaceURI, transactionID, graph3)
6  commitTransaction(transactionID)
```

The sequence of operations above creates a transaction (1), reads a named graph identified by *namedGraphURI* (2), consumes another named graph retrieved by *template* (3), processes the received graphs and calculates a new graph *graph3* (4) and writes the new graph (5). Finally the transaction is committed (6). By enclosing the read, take and write operations with a transaction, it is guaranteed, that either all the take and write operations are applied to space identified by tripeSpaceURI, or the operations do not have an effect at all. Further it is guaranteed that the graph observed in (2) still exists and has not been modified by a concurrent operation of another participant.

3.3 Triple Space Kernel Overview

The TS kernel is a software component which can be used to implement both Triple Space servers and heavy clients. In the former case it also provides a proxy component, which allows light clients to remotely access the server. The TS kernel itself consists of the multiple components shown in Fig. 2.

TS operations and security layer accepts Triple Space operations issued by participants via the TSC API. **Heavy clients** run in the same address space as the TS kernel, and the TS kernel is accessed by its native interface. **Light clients** use **TS proxies** to access the TS kernel of a server node transparently over the network. As a variation a light client can access a TS kernel via a standardized protocol, e.g. HTTP. In this case a server side component, e.g. a **servlet**, translates the protocol to the native TS kernel interface. The execution of a TS operation includes verification of security constraints, maintaining state of blocking operations and invocation of the underlying coordination layer. The **security management API** is used to define and change security configurations such as access control for spaces or named graphs. The **coordination layer** implements transaction management, i.e. the creation, commit and abort of a transaction and guarantees that concurrent operations are processed consistently. It accesses the local *Data Access Layer* to retrieve data from a space and to apply permanent changes to a space [28]. Furthermore, if a Space is spanned by multiple TS kernels, the Coordination Layer is responsible for inter-kernel communication to distribute and collect data and to assure that all involved kernels have a consistent view to a space. The **mediation engine** resolves heterogeneity issues by providing mappings for possibly occurring mismatches among different RDF triples. It is due to the possibility that different participants may have different RDF schemas while communicating via triple space. Mapping rules for mediation are provided to the mediation engine at design time and are processed during run time in order to resolve heterogeneities by identifying mappings. The **mediation management API** provides methods to turn on/off the usage of mediation engine, to add, remove and replace mediation rules. The coordination layer is based on the middleware Corso, which is used to replicate named graphs to all involved TS kernels and guarantees consistency via built-in transactions. YARS [25] is used to realize the data access layer. Distributed kernels can be used to realize architectures containing all kinds of nodes. A Triple Space can also be realized by a single TS kernel, which is accessed by remote participants in Client/Server style. For non-distributed TS kernels the task of the coordination layer boils down to transaction management, which can efficiently be done by a traditional RDF database. Compared to a distributed implementation a centralized implementation however does not scale as well with the number of participants and geographical distribution of participants.

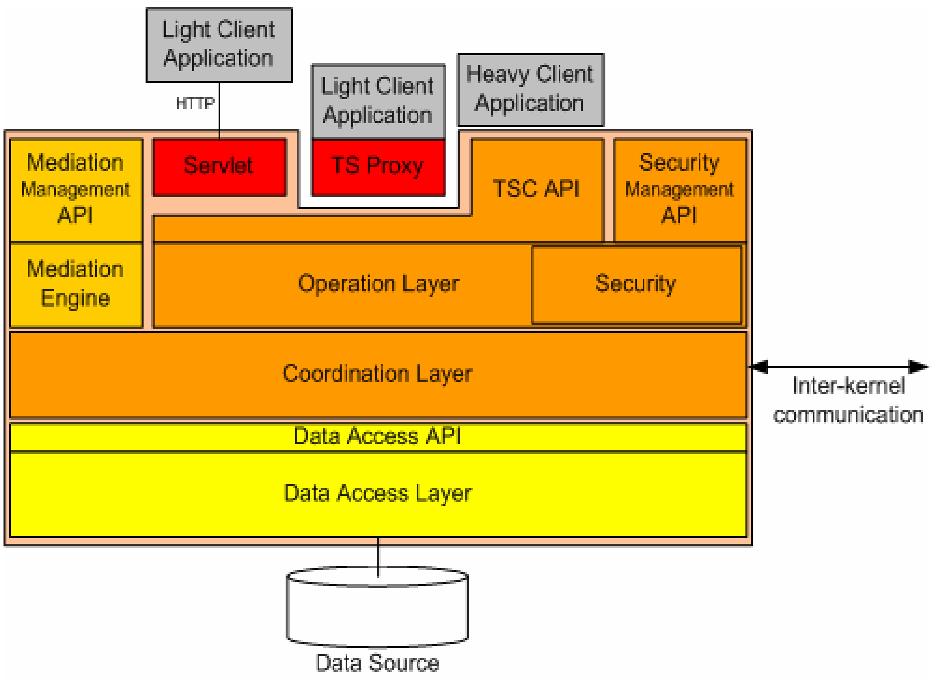

Fig. 2. The TS Kernel

3.4 Coordination Layer

Since the coordination layer is the core part of the architecture, we illustrate it in more details. Basically the coordination layer has three responsibilities, as shown in Fig. 3. Firstly, local TS operations, such as reading and writing named graphs (Fig. 3, 1a), are executed by accessing the local data access layer (Fig. 3, 1b) and by propagating changes to other involved TS kernels (Fig. 3, 1c). Other local operations include transaction and Triple Space management. Secondly, changes of a space originating from other TS kernels are recognized (Fig. 3, 2a) and applied to the local data access layer (Fig. 3, 2b). Thirdly, remote TS kernels involved to span a certain space are discovered automatically in the network (Fig. 3, 3).

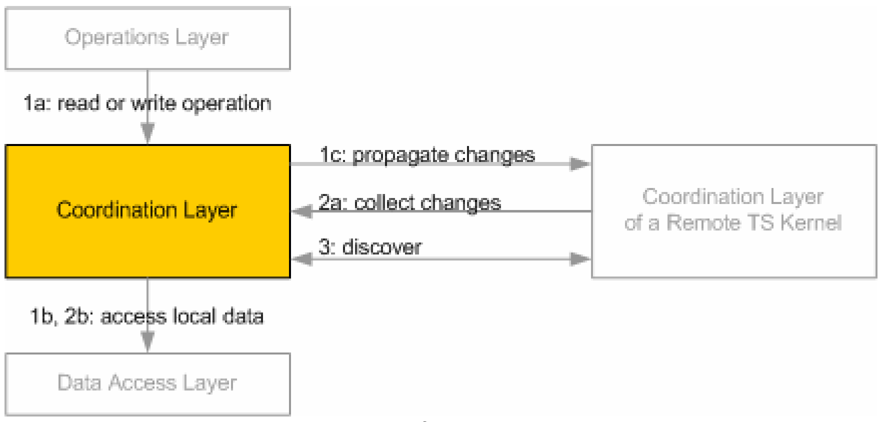

Fig.3. Interactions of the Coordination Layer

Triple Space management includes creating new as well as finding, joining and leaving existing Triple Spaces. To create a Triple Space, the access layer is used in order to initialize the new space and create necessary meta-data [20]. To find a space for a given URI other TS kernels may need to be contacted. By joining an existing space the TS kernel will be notified by all other involved TS kernel whenever the space changes. A TS kernel automatically joins a space as soon as that space is accessed via the kernel by any TS operation. Leaving a space stops a TS kernel from being notified about changes. Further, replicated data of the space may be deleted at the leaving TS kernel.

Consistent concurrent access to named graphs is provided via transactions [20]. In principle both optimistic and pessimistic transactions are applicable for TSC, however they are not exchangeable due to differences in their semantics [29]. We decided to support optimistic transactions, as supported by Corso, because they provide a higher degree of concurrency, if read operations are more frequent than write operations, which results in a higher throughput, because they are free of deadlocks without the introduction of additional, semantically sophisticated timeout parameters [20]. Finally, because they enable a pragmatic integration of a data access layer, which itself does not support a transaction interface. A property of optimistic transactions is that a commit of a transaction may fail even if all previous operations using that transaction succeeded. In practice this can mostly be overcome by repeating the whole transaction. Regarding transactional semantics, all TS operations fall in one of the following categories:

- **Blocking retrieval operations**: The semantics of a transactional read operation is that at commit time of the transaction the named graph must be in the same state as observed at read. Otherwise the commit fails. The same is true for take operations; additionally, if the commit succeeds, the named graph is removed from the space. An update has the semantics of a take followed by a write. An important property of this category of operations is that the transactional semantics only depend on the affected named graph. Intuitively this is clear: a named graph A can be read or updated independently from another named graph B. As a result no global view of a space is needed for transactional operations of this category. For example, a named graph can be taken even if some of the participating TS kernels are offline, as long as the coordination layer guarantees the above semantics.

- **Non-blocking retrieval operations**: If a non-blocking operation returns a named graph, the semantics are as described for the blocking read operation. If, however, no named graph is returned, because no existing named graph matches the given template, different semantics are possible. *Strict semantics* would guarantee that at commit time still no matching named graph exists. This, however, requires a global view to the whole Triple Space, i.e. it must be guaranteed that at no other TS kernel a matching named graph has been written - after the non-blocking operation and before the commit. This is infeasible if TS kernels should be allowed to be disconnected from the network and should thus be able to be used for writing new named graphs to a space. That's why *relaxed semantics* are more reasonable: If a non-blocking read or take operation does not return a named graph, it means that currently no matching named graph could be found, however no guarantees are made that in fact no matching named graph *exists* (anywhere in the distributed Triple Space).

- **Write a graph to a space**: Writing a graph to a space can always succeed, i.e. a write operation never causes a transaction to fail. This is especially useful if a heavy client wants to produce data even if it is offline. If, however, the transaction used for a write fails for some other reason, the written graph will be discarded. This is especially useful if a heavy client wants to produce data even if it is offline.

- **Other global operations**: As non-blocking read and take operations, the count operation needs a global view to the Triple Space for strict transaction semantics. Relaxed transaction semantics are not applicable since this would be equivalent to using no transaction at all.

The effects of all operations do not become globally visible until the used transaction has been committed successfully. After a successful commit all effects need to be propagated to all other involved TS kernels and to the local data access layer. An important requirement for the coordination layer is to guarantee atomicity, consistency and durability even in the presence of (partial) failures. As mentioned above, the transaction semantics of most operations depend on the affected named graph only. That's why operations, which address named graphs directly, such as write and the variants of read and take operations, which use a named graph URI parameter, can be realized comparable easily. In a first step, the affected named graph is associated with the transaction by the coordination layer. This usually involves communication with other TS kernels in order to collect information needed at commit time. Secondly, the data access layer may be used to retrieve the contents of named graphs. Finally, when the transaction has been committed successfully, the effects are propagated to the data access layer.

Operations which take templates are more complicated. In a first step the data access layer is used to find out about the affected named graphs, e.g. to find a named graph which matches a given template. In a second step the affected named graphs could be bound to the transaction - however this is too late, since it cannot be guaranteed that the named graphs observed by the data access layer have not changed between step one and two.

Multiple TS kernels may be used to span a distributed Triple Space. To realize a distributed Triple Space, the coordination layers of all involved TS kernels communicate in order to distribute and access data. The physical addresses of the kernels are in general independent from the URIs used to identify a Triple Space. For example, two TS kernels deployed on 192.168.0.1 and 192.168.0.2 can span the Triple Space *example/abc*. The coordination layer is responsible to discover all TS kernels spanning a Triple Space.

4 Conclusion and future work

In this paper, we describe the current status of the development of Triple Space Computing as a novel communication and coordination framework that combines Semantic Web technologies and tuple space computing for Semantic Web Services. We have conducted current state of the art studies in related fields and identify the value added by TSC. Based on this, we propose the overall architecture of TSC and explain the interactions among different components.

TSC is an Austrian national funded project which still has two years to go. During these two years, we will provide a consolidated TSC architecture and interfaces for cooperation among the components and for the TSC infrastructure as a whole, especially design mediation and query engine components for TSC. Furthermore, we will focus on data replication, security and privacy mechanisms in TSC, to investigate the relation between WSMO[7] and TSC conceptual model and to find out how standard architectures (REST, SOA) can be better applied in TSC. In the end, a running prototype will be provided and the usability will be tested via a case study on how TSC can enhance message communication and process coordination in WSMX[8].

[7] http://www.wsmo.org
[8] http://www.wsmx.org

References

1. Fensel, D.: Triple-based Computing. Digital Enterprise Research Institute (DERI) Technical Report DERI-TR-2004-05-31 (2004)
2. Gelernter, D.: Generative Communication in Linda. ACM Trans. Prog. Lang. and Sys. 7 (1985) 80-112
3. Johanson, B., Fox, A.: Extending Tuplespaces for Coordination in Interactive Work-spaces. J. Systems and Software, 69(2004) 243-266.
4. Kuehn, E.: Fault-Tolerance for Communicating Multidatabase Transactions. Proc. Of the 27th Hawaii Int. Conf. on System Sciences (HICSS), ACM, IEEE (1994)
5. Carriero, N., Gelernter, D.: How to Write Parallel Programs: A First Course. MIT Press (1990)
6. Gelernter, D.: Multiple Tuple Spaces in Linda. Proc. Of PARLE, Springer Verlag (1989)
7. Butcher, P., Wood, A., Atkins, M.: Global Synchronization in Linda. Concurrency Practice and Experiences 6(1994) 505-516
8. Rowstron, A., Douglas, A., Wood, A.: Copy-Collect: A New Primitive For the Linda Model. YCS 268 (1996)
9. Anderson, B., Shasha., D.: Persistent Linda: Linda + transactions + query processing. In: Banatre, J.P., Le Metayer, D. (eds): Research Directions in High-Level Parallel Programming Languages, Lecture Notes in Computer Science, Vol. 57. Springer-Verlag, Berlin Heidelberg New York (1991) 93—109
10. Rowstron, A., Wood, A.: Bonita: A Set Of Tuple Space Primitives For Distributed Coordination. Proc. Of the 30th Hawaii Int. Conf. on System Sciences (HICSS), IEEE Computer Society, Washington (1997)
11. Graham, S., Niblett, P. (eds): Web Services Notification (WS-Notification). Technical Specification (2004). http://www-106.ibm.com/developerworks/library/specification/ws-notification
12. Geller, A. (ed): Web Services Eventing (WS-Eventing). Technical Specification (2004). http://www-106.ibm.com/developerworks/library/specification/ws-eventing
13. Virgillito, A.: Publish-Subscribe Communication Systems From Models To Applications. PhD Thesis. http://www.dis.uniroma1.it/~virgi/virgillito-thesis.pdf
14. Li, H., Jiang, G.: Semantic Message Oriented Middleware For Publish/Subscribe Networks. Proc. Of. The SPIE, 5403 (2004) 124-133
15. Strom, R., Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D., Ward, M.: Gryphon: An Information Flow Based Approach to Message Brokering (1998)
16. Oki, B., Pfluegel, M., Siegel, A., Skeen, D.: The Information Bus – Architecture For Extensive Distributed Systems. Proc. Of the 1993 ACM Symposium on Operating Systems Principles (1993)
17. Castro, M., Druschel, P., Kermarrec, A., Rowston, A.: Scribe: A Large-scale and Decentralized Application Level Multicast Infrastructure. IEEE J. on Selected Areas in Communications 20 (2002)
18. Carzaniga, A.: Architectures For An Event Notification Service Scalable to Wide-Area Networks. PhD Thesis. Politecnico di Milano. (1998)
19. Pietzuch, P., Bacon, J.: Hermes: A Distributed Event-based Middleware Architecture. Proc. Of The International Workshop on Distributed Event-based Systems (DEBS) (2003)
20. Krummenacher, R., Martin-Recuerda, F., Murth, M., Riemer, J.: TSC Framework. TSC Project Deliverable (2005)

21. Fielding, R. T.: Architectural Styles And The Design of Network-based Sofware Architectures. PhD Thesis, University of California, Irvine (2000)
22. Khare, R., Taylor, R. N.: Extending the Representaional State Transfer Architectural Style For Decentralized Systems. Proc. Of The International Conference on Software Enginneering (ICWE), Edinburgh, Scotland (2004)
23. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: The OceanStore Prototype. Proc. Of The 2nd USENIX Conference on File and Storage Technologies (FAST) (2003)
24. Yang, B., Garcia-Molina, H.: Designing a Super-Peer Network. IEEE International Conference On Data Engineering (2003).
25. Harth, A.: Optimized Index Structures For Querying RDF From The Web (2005). http://www.deri.org
26. Eugster, P. T., Felber, P.A., Guerraoui, R., Kermarrec, A. M.: The Many Faces of Publish/Subscribe. ACM Computing Survey (2003)
27. Carroll, J. J., Bizer Ch., Hayes, P., Stickler, P.: Named Graphs. J. of Web Semantics 3 (2005)
28. Kuehn, E.: Virtual Shared Memory for Distributed Architecture. Nova Science Publisher (2001)
29. Gruber, R. E.: Optimism vs. Locking: Study of Concurrency Control for Client-Server Object-Oriented Databases. Technical Report, MIT/LCS/TR-708 (1997)

Acknowledgement

The work was supported by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT Semantic Systems project Triple Space Computing (TSC 809249). The authors would like to thank the people who are involved in TSC project and the funding support from the Austrian government.