EXTENDING TUPLE SPACES TOWARDS A MIDDLEWARE FOR eCOMMERCE

Studienarbeit von Philipp Obreiter

Betreuer: Guntram Graef
Telecooperation Office (TecO), Universität Karlsruhe
Vincenz-Prießnitz Str. 1, 76131 Karlsruhe, Germany

In Zusammenarbeit mit: Shanghai Jiaotong University
# Contents

I. INTRODUCTION ........................................................................................................... 3

II. TUPLES PACES ........................................................................................................... 4  
- TERMINOLOGY ........................................................................................................... 4  
- SIGNATURES ................................................................................................................ 4  
- PARALLEL COMPUTING .......................................................................................... 6  
- OPEN DISTRIBUTED COMPUTING ........................................................................... 6  
- CLASSIFICATION ........................................................................................................ 8

III. TUPLES PACES IN AN ECOMMERCE ENVIRONMENT ............................................ 10  
- REQUIREMENTS ......................................................................................................... 10  
- COMPLIANCE ............................................................................................................ 11

IV. CONCEPTION AND IMPLEMENTATION OF XTSPACES ........................................ 14  
- SCOPE ....................................................................................................................... 14  
- FAILOVER SAFETY .................................................................................................... 14  
- SCALABILITY ............................................................................................................. 19  
- ARCHITECTURE ......................................................................................................... 23  
- IMPLEMENTATION ..................................................................................................... 26  
- PERFORMANCE ......................................................................................................... 27

V. FUTURE WORK FOR XTSPACES .............................................................................. 29  
- EXTENSIONS ............................................................................................................. 29  
- PREDICTION SERVICES BASED ON SIGNATURES .................................................. 29  
- UNRESOLVED PROBLEMS ....................................................................................... 32

VI. CONCLUSION .......................................................................................................... 33

VII. LITERATURE ............................................................................................................ 34

A. APPENDIX: USAGE OF XTSPACES ......................................................................... 37  
- INSTALLATION .......................................................................................................... 37  
- RUNNING THE SYSTEM ............................................................................................ 37

B. APPENDIX: PERFORMANCE TESTING .................................................................... 42  
- BENCHMARK .............................................................................................................. 42  
- TESTING STRATEGY .................................................................................................. 45
I. Introduction

In its original conception a decade ago, the World Wide Web (Web) was a distributed system for knowledge interchange. The use of the Web has gradually changed since and has moved beyond its anticipated scope. Nowadays, the Web is seen as a new application platform which offers most services needed for a distributed operating system. This trend has eased the conception and deployment of distributed applications on the Web. Web applications take advantage of the simple implementation model of the Web. They rely on existing dedicated communication components, that is, the web servers and the client frontends (browsers). However, the development and evolution of Web applications appears highly complicated and non-trivial (Gaedke 1998). Therefore, a new discipline named Web Engineering (Gaedke and Graef 2000) has emerged to deal with these demands. Web Engineering is the application of Software Engineering concepts (Fairley 1985) to the development and evolution of Web applications in order to achieve both a cost reduction and an increase in quality.

More recently, organizations have begun to offer Web-based products and services on the Web, they offer eCommerce services. A more general definition defines eCommerce as "any form of business transaction in which the parties interact electronically rather than by physical exchanges or direct physical contact" (ECOM 1998). Normally, an eCommerce service can't be thought of involving the services of only one organizational entity. In general, it is composed of application systems of different business units or even of different enterprises. That is why eCommerce involves coordination of businesses. Unfortunately, businesses are based on a variety of information systems. This heterogeneity manifests in networks, operating systems, data representation and semantics, business processes and business software. eCommerce applications are conceived as provider of eCommerce services. Therefore, they do not only inherit the problem domain of Web Applications. In addition, they have to provide a means to coordinate and combine different business services.

It seems to be a promising idea to use a dedicated technology that forms a coordination middleware (Bernstein 1996). Such a middleware should abstract from the heterogeneity of the participating information systems. In addition, it should come along with a convenient model for coordinating these information systems, even though they are distributed and loosely coupled applications. As the coordination middleware, (Gaedke and Turowski 05/1999) suggests a federation layer that is built upon the tuplespace concept (Gelernter 1985). However, the original tuplespace concept has to be adjusted in order to meet the specific requirements of an eCommerce environment.

In the following sections, we first introduce tuplespaces and their state-of-the-art. Afterwards, we will formulate the requirements of a coordination middleware from the viewpoint of eCommerce applications. The existing tuplespace technologies will be checked against these requirements. We will see that none of them is qualified to form an adequate coordination middleware. Therefore, we will conceive a tuplespace technology named xTSpaces that meets the two central demands found. Later, we will point out how xTSpaces can be extended to meet a larger number of requirements and which problems are still to be solved while achieving this. Lastly, we will conclude on an outlook.
II. Tuplespaces

TERMINOLOGY

In the following, basic terms which are used throughout this work are introduced. The terminology is based on the Linda 1 proposal (Gelernter 1985).

A field consists of a type and optionally a value. If a value is assigned, the field is said to be actual. An actual field A matches an actual field B if and only if the types and values of A and B are equal. A field is said to be formal, if no value but only a type is assigned. A formal field A matches an actual or formal field B if and only if A and B have the same types. In object-oriented languages, an actual field is an object and a formal field is a class. Therefore, the definition of matching is adjusted. A formal field A matches an actual or formal field B if and only if B's type is equal to A's type or its subtype by inheritance. An actual field has to implement a method that takes as parameter a field and returns whether it matches it. It is assumed that such a method is reflexive and transitive, in order to implement reasonable matching semantics. Hence, an actual field A matches an actual field B if and only if A's matching method returns true on a call with B as parameter.

A tuple is an ordered collection of actual fields.

An actual template is an ordered collection of actual fields. A formal template is an ordered collection of fields with at least one of the fields being formal. An actual or formal template matches a tuple if and only if each of their fields match, taken into account the ordering. A semantic template is able to match a tuple structurally. E.g. there exists exactly one semantic template that matches all tuples. Until recently, it was the only semantic template supported by implementations of tuplespaces. Therefore, semantic templates are not discussed in detail in the following sections. Some terminologies (Gaedke and Turowski 06/1999) do not distinguish between tuples and templates; tuples and actual templates become actual tuples, formal templates and semantic templates become formal tuples and semantic tuples. In the following, the term template will be primarily used to stress that something is tried to be matched again tuples.

A tuplespace is a logically shared associative memory that is used to store and retrieve tuples through the tuplespace primitives. In a tuplespace, a tuple may exist more than once and usually there is no ordering of the tuples. Since Linda 3 (Gelernter 1989), a tuplespace consists of multiple spaces containing tuples. An entity using a tuplespace in order to store and retrieve tuples is called a tuplespace client. The provider of the tuplespace is called the tuplespace server.

SIGNATURES

The structure of tuples and templates induces their signature. A formal definition follows: \(\tau\) is the set of active, formal and semantic tuples. \(\sim \subset \tau^2\) and is defined by \(\forall t,s \in \tau: [(t \sim s) \iff ((t \;matches\; s) \wedge (s \;matches\; t))].\) \(\sim\) is reflexive, symmetric and transitive, so that it is an equivalence relation. \(\tau^\sim = \tau / \sim = \{t^- \mid t \in \tau\}\) is \(\tau\) partitioned in equivalence classes. Each equivalence class is a signature, in other words \(\sigma\) is a signature if and only if \(\sigma \in \tau^\sim\). A tuple t has the signature \(\sigma\) if and only if \(t^- = \sigma\). It is possible to extend the definition of actual, formal and semantic to signatures. Such a definition is well-defined.
Signatures are structured hierarchically. The matching relation on signatures \( \leq \subset \tau^2 \) is defined by \( \forall t^-, s^- \in \tau^* : [ (t^- \leq s^-) \iff (t \text{ matches } s) ] \). A signature \( \rho \) is matched by the signature \( \sigma \) if and only if \( \sigma \leq \rho \). \( \leq \) is reflexive, anti-symmetric and transitive. Therefore, \( \leq \) is an ordering relation and \((\tau^-, \leq)\) is a partially ordered set. \( \bot \) is the signature of the semantic tuple that matches every other tuple. \( \top \) is the signature that is matched by every signature. Hence, \( \bot \) is the minimal element of \((\tau^-, \leq)\) and \( \top \) is the maximal element of \((\tau^-, \leq)\).

The ancestors of \( \sigma \) are \( \sigma_{\backarrow} := \{ \rho \mid \rho \leq \sigma \} \). The descendants of \( \sigma \) are defined as \( \sigma_{\nearrow} := \{ \rho \mid \sigma \leq \rho \} \). If \( \xi \subset \tau^- \), \( \xi \) has the set of ancestors \( \xi_{\backarrow} := \{ \rho \in \tau^- \mid \forall \sigma \in \xi : \rho \leq \sigma \} \) and the set of descendants \( \xi_{\nearrow} := \{ \rho \in \tau^- \mid \forall \sigma \in \xi : \sigma \leq \rho \} \). \( \bot \in \xi_{\backarrow} \) and \( \top \in \xi_{\nearrow} \), so that \( \emptyset \notin \{ \xi_{\backarrow}, \xi_{\nearrow} \} \). Then, two functions \( \lor, \land : \tau^2 \rightarrow \tau^- \) are defined as \( \sigma_1 \land \sigma_2 := \land(\sigma_1, \sigma_2) \in \{ \rho \in \{ \sigma_1, \sigma_2 \}, \land \forall \sigma \in \{ \sigma_1, \sigma_2 \}, \land : (\sigma \leq \rho) \rightarrow (\rho = \sigma) \} \) and \( \sigma_1 \lor \sigma_2 := \lor(\sigma_1, \sigma_2) \in \{ \rho \in \{ \sigma_1, \sigma_2 \}, \lor \forall \sigma \in \{ \sigma_1, \sigma_2 \}, \lor : (\rho \leq \sigma) \rightarrow (\rho = \sigma) \} \). Therefore, \((\tau^-, \lor, \land)\) is a lattice.

If \( \xi \subset \tau^- \), then \( \xi_{\downarrow} := (\lor_{\sigma \in \xi} \sigma) \) and \( \xi_{\uparrow} := (\land_{\sigma \in \xi} \sigma) \). Therefore, \( \xi \) is extendable to a set of signatures \( \xi_{\downarrow} := \xi_{\downarrow} \cup \{ \bot \} \) with a minimal element, and to a set of signatures \( \xi_{\uparrow} := \xi_{\uparrow} \cup \{ \top \} \) with a maximal element.

Figure 1 gives an example of signatures and their structure induced by matching. It is a multi-inheritance tree.

![Signature multi-inheritance trees](image)

Figure 1: Signature multi-inheritance trees

Even if a tuplespace consists of multiple spaces, the concept of signatures is appropriate. In the following, it is shown how a multiple space tuplespace can be reduced to a single space tuplespace. \( \pi \) is the set of valid spaces. \( \tau \) is defined as known. Then define \( \tau' \subset \tau \) as follows:
\[
\tau' := \{ e \in \tau \mid \text{head}(t) \in \pi \land \text{tail}(t) \in \tau \}
\]
with head giving back the first field and tail giving back the rest of the tuple. In consequence, signatures and the matching relation are defined on \( \tau' \) as before on \( \tau \). Due to \( \tau' \subset \tau \), it appears that a single space tuplespace is more general than a multiple space tuplespace. On the other hand, it is trivial to show that a multiple space tuplespace is more general than a single space tuplespace. It is concluded that they are equivalent. Therefore, studies on signatures in other chapters will assume the single space tuplespace case.
PARALLEL COMPUTING

The first coordination language (Gelernter and Carriero 1992) based on tuplespaces was Linda (Carriero and Gelernter 1989). It became very popular in the field of parallel programming. Linda extends an arbitrary host programming language, like C, C++, Prolog, Lisp, by adding a set of minimal primitives. They are:

\begin{align*}
\text{out} & (ts, \text{tuple}) \\
& \text{Write tuple into the tuplespace ts.} \\
\text{in} & (ts, \text{template}) \\
& \text{Read and remove a tuple matched by template from the tuplespace ts.} \\
\text{rd} & (ts, \text{template}) \\
& \text{Read a tuple matched by template from the tuplespace ts.}
\end{align*}

The upper primitives are synchronous, so that reading a tuple will block if no matching tuple is found. In order to overcome this restriction, the asynchronous non blocking primitives \text{inp} and \text{rdp} were added afterwards (Carriero 1987).

Most of the early research work in the tuplespace domain was done in the parallel programming community. Throughout that work, many new additional primitives and features have been proposed for extending Linda. They are targeted at providing new functionality and better performance. For functionality extensions, the most accepted proposals are the introduction of multiple spaces per tuplespace (Gelernter 1989) and the introduction of bulk primitives which are able to work on more than one tuple per primitive (Anderson et al. 1991, Jensen 1993). For performance amelioration, most of the proposals are targeted at optimizing the compilation process (Wells et al. 1995, Carreia et al. 1994). The variety of extensions has resulted in different Linda dialects and tuplespace implementations, such as Bauhaus Linda (Carriero et al. 1995), Piranha (Gelernter et al. 1993), MTS-Linda (Jensen 1993) and Melinda (Hupfer 1990). In most scenarios of parallel programming, the tuplespace clients are already known at compile time. In this case, these programs are said to be closed implementations. Therefore upper tuplespace implementations come along with optimization using compile-time analysis (Carriero et al. 1990).

OPEN DISTRIBUTED COMPUTING

Until the mid 90s, the tuplespace concept was seldom known outside the parallel programming community. The emergence of distributed applications, especially internet applications, has since triggered the use of tuplespaces beyond their original scope. Tuplespaces became a coordination mechanism for open distributed computing, that is defined to be those systems that are "dynamically composed from non-dedicated hardware and software components" (Tolksdorf 1995). In consequence, a tuplespace can be used as brokering mechanism between service providers and service clients or between peer-to-peer users. That is why tuplespaces go beyond the client-server paradigm, because service providers and clients are both tuplespace clients. Service providers and clients do not directly communicate, but they are decoupled in time and space. They even do not have to know about each other. New design patterns and paradigms have arisen, the best known are the blackboard (Luger et al. 1993) and master-worker (Freeman et al. 1999) patterns.
The scope of tuplespaces in open distributed computing (open tuplespaces) has drifted from its origins. Tuplespace clients are deployed in networks of uncertain connection stability and latency. Furthermore, some distributed applications are planned to last very long, however without any loss of data allowed. In addition, it is possible that tuplespace clients crash without the tuplespace server knowing about it. This demands for adjusting the original tuplespace concept.

Therefore, functionality extensions of open tuplespaces aim at guaranteeing that the ACID demands are fulfilled. The atomicity and isolation of the tuplespace’ interaction with a client is achieved by the means of transactions. Three additional primitives are thus added, they are primitives for transaction begin, commitment and abort. The durability of a tuplespace is achieved by logging or by coupling the tuplespace to a heavywight database. In this case, tuplespaces act as a communication buffer from the tuplespace clients’ viewpoint and as a database cache from the the database’s viewpoint. However, only few tuplespace implementations are truly durable (Anderson et al. 1991). The consistency of the tuplespace has to be ensured by the tuplespace clients themselves. This is because the tuplespace doesn’t understand the semantics of data stored in tuples.

Performance enhancements of open tuplespaces are achieved by offering a variety of new primitives. Their aim is to minimize communication over the network. Their drawback is an inevitable interface blow.

Recent open tuplespace implementations are Jada, JavaSpaces and TSpaces. They are all implemented in Java. In the following, a short introduction is given:

PageSpaces (Ciancarini et al. 1996) is a high level framework for distributed applications and services. Agents incorporate generic roles within such a framework. PageSpaces is based on Laura (Tolksdorf 1994) and ShaDE (Castellani et al. 1995). Laura uses Linda in order to broker services and ShaDE introduces object-orientation to Linda. As a result of the PageSpaces project, Jada aims at providing basic coordination for PageSpaces, that is the agents’ interaction. Apart of the basic tuplespace primitives in, rd and out, Jada also supports bulk primitives. Due to its simple nature, Jada is very lightweight.

JavaSpaces (Sun 2000) comes with several differences. Tuples are called entries, formal fields are called wildcards. The primitives in, inp, rd, rdp and out are called take, takeIfExists, read, readIfExists and write. Tuples are serializable objects and their matching takes class hierarchies into account. JavaSpaces is embedded into Jini and uses its features in order to provide a richer set of primitives. Tuples can be forgot over the time (leasing), a tuplespace client can be notified on changes in the tuplespace (events) and a transaction mechanism based on the two-phase commit protocol is available. According to JavaSpaces’ design goals, durability is not a concern. JavaSpaces lacks bulk primitives and is based on the Remote Method Invocation (RMI). Therefore, performance of JavaSpaces applications suffer. Lastly, JavaSpaces is tightly bound to Jini. As a result, JavaSpaces is not as lightweight as Jada is.
TSpaces (Wyckoff et al. 1998) takes a different approach. It is directly based on TCP/IP and therefore cannot rely on RMI for marshalling. The basic primitives are called waitToTake, take, waitToRead, read and write. As in JavaSpaces, tuples are serializable objects and their matching takes class hierarchies into account. Furthermore, clients’ notification on events (callback) and coarse-grained space wide transaction are supported. On top of this, there are primitives for bulk operations (scan) and for rendez-vous (rhonda). It is possible to define queries that go beyond the original matching concepts of tuplespaces. New but merely mature features are provided, too. Spaces within the tuplespace can be owned and access to them can be restricted. Logs are used to ensure durability. Finally, custom operators can be uploaded from the client for the purpose of server side processing of the tuplespace. From its conception, it is possible to use IBM’s DB2 for ensuring persistency and durability. But by the moment of writing, this feature is not available yet. As a result, TSpaces is much bigger and more complex than Jada and JavaSpaces. On the other hand, TSpaces comes along with better performance and a wider range of functionality.

CLASSIFICATION

How do tuplespaces compare to other similar cooperation and communication concepts? In the following, the tuplespace concept is classified among those. Their commonalities and their differences are described. Even though the concepts are all applicable in many problem domains, their scope is quite different. Language interoperability and performance issues are kept aside during this classification.

Generally spoken, the cooperation and communication concepts are applied on systems with space or time remoteness. Their goal is to provide a transparent view upon the remoteness. This is their point in common. However, they considerably differ in what kind of remoteness they hide in which degree. Remoteness of space means that two or more entities are located in different places, so that their interaction has to go through system borders. Remoteness of time means that two or more entities are active at different moments, so that their interaction has to survive longer than its participating entities (lifetime borders). Another difference is the coupling of participating entities. Among the concepts there are some that totally decouple the entities, so that all interaction is indirect. Meanwhile, other concepts provide mechanisms for direct interaction between the participating entities. In the following, three concepts are examined:

The first one is to provide an infrastructure for calls on remote entities. The infrastructure is provided by a middleware. It places a stub on the caller’s side and a skeleton on the callee’s side. Therefore, the system border is crossed by the means of the stub-skeleton interaction. This concept abstracts from space remoteness. However, it doesn’t deal with time remoteness, since it will only work if both, caller and callee, are active at the same time. So to speak, the entities’ lifetime borders cannot be crossed. Among the middlewares using this concept is CORBA (Ben-Natan 1995), RMI (Sun 1997), EJB (Sun 1998) and DCOM (Microsoft 1996). Still, they all differ in their brokering mechanism of a call from the caller’s stub to the callee’s skeleton.
Messaging systems go further in decoupling participating entities. A sender puts his message in a queue on the messaging system. Those entities who have registered to this kind of message before, will then receive the message. The sender does not know which entities are the receivers of his message, and the receivers do not necessarily know about whom sent the message. Therefore, interaction between participating entities is indirect. In addition, more complex communication schemes like broadcasts are readily supported. However, a receiver will not get the message if he has not registered to it as a listener. Hence, time remoteness cannot be gapped. An example for messaging systems is JMS (Van Huizen 2000).

In contrast to upper two concepts, the focus of databases is to gap time remoteness. Objects are persistently stored after insertion, so that their retrieval is time independent. Therefore, databases allow interaction between the inserter entity and the retriever entity through lifetime borders. This scheme implies that the participating entities are decoupled.

Of course, it is possible to extend databases to gap space remoteness. Messaging systems can be altered, so that they deliver messages to entities who only register after message submittal. However, this section describes the concepts’ orientation and scope. Therefore, the ensuing classification is appropriate:

![Classification of the tuplespace concept](image)

**Figure 2:** Classification of the tuplespace concept

Figure 2 classifies upper three concepts by the means of entity coupling and remoteness transparency. It is an interesting fact that no direct entity interaction is possible if the time remoteness has to be gapped. The reason is that direct entity interaction cannot go through the lifetime borders. If either the caller or callee does not exist, the caller and callee cannot interact by a call.

The tuplespace concept brokers data and no calls, so that entities participate indirectly. Furthermore, the scope of tuplespaces is to bridge time and space remoteness. The receiver of objects has not to exist on object submittal to the tuplespace. On the other hand, the basic design goal of tuplespaces is to enable interaction of space-remote entities. Therefore, the tuplespace concept is situated between the messaging concept and the database concept in this classification.
III. Tuplespaces in an eCommerce Environment

REQUIREMENTS

What is needed, is a coordination middleware supporting eCommerce applications. The tuplespace concept was historically first conceived for parallel computing. It was extended in order to be applicable for open distributed computing. However, the application of tuplespaces in an eCommerce environment is far more exigent. In the following, its catalogue of requirements is compiled. Based on the properties of the original Linda tuplespace, further important properties are identified. They are discernible into absolutely necessary and desirable, thus forming core requirements and optional requirements. At last, it is pointed out which properties are meaningless for the application of tuplespaces in an eCommerce environment. In other words, they are no requirements, since they are orthogonal to the given problem domain.

A middleware supporting eCommerce applications has to be flexible enough to reflect changes within their usage profile. This is especially true for the demand that a tuplespace should scale in accordance to the application’s needs. This includes the size of the tuples to store, that easily exceeds the anticipated size of tuples made up of primitive data type fields. In addition, the number of tuples is expected to rise with the number of the participating entities. This observation leads to the core requirement of scalability (c1) that reflects both, scalability of tuple size and number of tuples. If the performance is independent of the problem size, a tuplespace is fully scalable. If the performance suffers logarithmically to the problem size, a tuplespace is defined as sufficiently scalable. Tuplespaces have to be at least sufficiently scalable in order to meet (c1).

On the other hand, a tuplespace middleware has to comply to the ACID demands already mentioned above. An eCommerce application has to be able to regard the middleware primitives as atomic and isolated (c2). By this, a sequence of primitives either has full effect on the tuplespace or none at all. But until the sequence’s end, no effect is visible to others. Otherwise, clients would have inconsistent views of the data stored within the tuplespace. A means of fulfilling (c2) is to provide primitives with transactional semantics. Durability calls for tuples being persistently stored (c3), regardless of system failures. However, eCommerce applications demand for far more. Their availability is scheduled near to 100%, so that the middleware supporting them inherits this constraint. Therefore, the tuplespace middleware cannot afford a single point of failure. If one entity of the middleware fails to provide its service, the middleware itself should still be working. It is deduced that failover safety (c4) is a core requirement. A tuplespace fulfilling (c4) is robust against failures.

In its conception as a cooperation mechanism for parallel programming, tuplespaces lack of any security concept. Security means that the participating client entities should only be able to do what they are allowed to do. The granularity of security on the coarse level is the access control to the tuplespace. On the fine level, it is defined which spaces or tuples are accessible by which primitives. Therefore, security policies and measures come along with need for administration and user management. A tuplespace middleware is required to provide some security mechanisms (c5). Otherwise, undesirable entities could keep track of the data exchanged by eCommerce applications, obviate or even alter it.
The more diversified the functionality of a tuplespace is, the more complex is the task to administer it. The idea is to be able to monitor, alter and maintain a tuplespace on the fly, without bringing the system down. Therefore, it is desirable to have tool support dedicated to this task (o1). Although this is an optional requirement, only its fulfillment enables effective realization of requirements like (c5).

Yet another desirable feature is the possibility to be notified of changes within the tuplespace on demand (o2). A frequent pattern of the coordination of eCommerce applications is the brokering of services. A service provider stores a tuple made up of a service description into a space and waits for clients of its service. Because the clients will answer with tuples of a certain signature, the service provider has to register on the middleware to be notified when such tuples appear.

The concept of a middleware for eCommerce applications as introduced in (Gaedke and Turowski 06/1999), calls for support of XML-tuples. The idea is that most data interchanged within a eCommerce application is well formed XML (Bray et al. 1997). In addition, XML-tuples provide an inherent integration of semantic tuples. Therefore, the support of XML-tuples (o3) is desirable, too.

A middleware has to abstract from the underlying heterogeneity. On the other hand, middleware components are placed on the client computer in order to give access to its primitives. In addition, it cannot be anticipated that servers of tuplespaces are placed on a homogeneity of computers. Therefore, the middleware should be portable (o4).

If an eCommerce application is written in different programming languages, it will probably have to access and call the middleware’s primitives from a variety of languages. The middleware should support calls on its primitives in whatever language (o5). Therefore, either a language independent calling mechanism of primitives has to be used, or wrappers of the primitives have to be provided for a subset of programming languages.

It is assumed that up to a certain level, middleware components placed on the client’s computer can be exigent to the system on which they are run. This assumption is based on the estimate that eCommerce applications are run on the businesses’ information systems and server machines. Therefore, it is no requirement for the middleware that the clients have to be thin (n1).

Note, that upper requirements specification is not true in other problem domains that could benefit from tuplespaces. E.g. in ubiquitous computing (Weiser 1993), it is a promising idea to use a tuplespace middleware to provide a cooperation means of ad-hoc networks. The requirements of such a middleware are completely different. The clients have to be as thin as possible, so that (n1) would become a core requirement.

**COMPLIANCE**

In the following, it is investigated whether Jada, JavaSpaces or TSpaces comply to the list of requirements compiled in upper section. If they fall short of fulfilling the requirements, tuplespaces would not be applicable in an eCommerce environment, yet. Therefore, further work on tuplespaces would have to be set about.
Jada is a non commercial release dating back to 1995. Therefore, its range of primitives is still close to those common in tuplespace implementations aimed at parallel computing. There is no notion of scalability, transactions, persistency, failover safety, security, administration and events. It is possible to add XML tuple support through customizing. But such an add-in does not exist. Since Jada is written in Java, it is portable. On the other hand, clients have to call the Jada primitives from Java, so that client applications written in other programming languages are hindered to use it.

JavaSpaces is aimed at providing distributed persistency of tuples, but falls short of making them durable. No database backend and no logging mechanism are realized, so that system failures come along with loss of data. It is not failover safe and scalable. However, JavaSpaces is built upon the standards Java, RMI and Jini, which eases an integration with other systems. Direct calls on primitives from other languages as Java is not foreseen. Mature eventing and transaction mechanisms are inherited from Jini. Just like Jada, JavaSpaces comes with no built in XML tuple support, but it can be added through customizing. A framework for security and administration is inherited from Jini, too. However, it is left to the client to implement his security policy and administration tools based on the framework.

TSpaces has no notion of scalability and failover safety. The transaction model is bound to single TSpaces servers. Transactions including more than one server are not intended. The use of DB2 in the backend in order to enable persistency is planned but not yet available. Meanwhile, a logging mechanism is used instead. The security concept is based on an access control with coarse granularity, that is on a space level. The clients are allowed to use certain primitives on a space. Clients can be organized into groups and an administrator is assigned. The concept lacks of administration support and fine granularity access control. On the other hand, TSpaces comes with a built in HTTP server for monitoring and administrating the contents of the tuplespace. Notification on two kinds of events is possible. Support for XML tuples is built in and such tuples can be queried with xPointer. Nevertheless, the implementation is inefficient and it is not taken advantage of the ease to express semantic tuples with XML. Because TSpaces is implemented in Java, it is portable but its clients have to call primitives from Java.

The following figure gives a resumee of the analysis taken:
Figure 3: Compliance of existent tuplespace implementations

It is recognizable that the core requirements of scalability (c1) and failover safety (c4) are not tackled by any of the recent tuplespace implementations. It is an interesting fact, that scalability was an issue in parallel computing in the past (Corradi et al. 1995). However, since then, the scope of tuplespace implementations has changed in its emphasis, so that the scalability has been neglected. On the other hand, failover safety is something new to tuplespaces. Apart of that, aspects like persistency and XML support are not yet solved in a satisfactory manner.
IV. Conception and Implementation of xTSpaces

SCOPE

As shown in prior chapter, there is no available tuplespace implementation that complies to the core requirements of a coordination middleware for eCommerce applications. Transactions, persistency and security measures have been introduced and applied in the two recent implementations JavaSpaces and TSpaces. But the topics failover safety and scalability remain untouched, even though they are a cornerstone and knockout criteria for the adequacy of a tuplespace middleware. Therefore, this contribution focuses on their compliance and the feasibility of such a project.

The tuplespace implementation xTSpaces is conceived and implemented. It should be failover safe and scalable, as far as this is feasible. If the incorporation of other requirements seems to be too cumbersome, they are neglected or ignored. In addition, xTSpaces must not be an isolated approach, that deprecates after evolution of existent tuplespace implementations. xTSpaces shall be based on them, in order to provide its services. Hence, xTSpaces does not compete with existing tuplespace implementations, but it rather complements them.

The conception of xTSpaces is based on the assumption that it is built upon a tuplespace implementation that supports events and that is written in Java. Therefore, JavaSpaces and TSpaces are both applicable as a base. In the following the tuplespace implementation used as a base, is called native tuplespace. If implemented on different native tuplespaces, xTSpaces provides a uniform view and a unified access to them. Furthermore, it could be used as a gateway in a system scape of heterogenous native tuplespaces.

However, the focus in this work is to provide a failover safe and scalable service. Therefore, dedicated mechanisms have to be set in place.

FAILOVER SAFETY

A system is failover safe if and only if it takes failures from at least two entities of the system to cause a system failure. An entity is a system’s part that collaborates with other parts. Depending on the view of the system, an entity can be coarse or fine granular, but it is always an elementary building block of the system. Failover safety aims at guaranteeing high availability rates. If failures of entities are uncorrelated and the entity that failed is replaced quickly, the system is not supposed to fail.

Entities of a tuplespace middleware are tuplespace servers, tuplespace clients and the network between. A tuplespace server runs a tuplespace service. Therefore, its failure will cause the service’s inavailability. A tuplespace client runs a proxy to the tuplespace server. This proxy offers an interface for calling the tuplespace’s primitives. It is assumed that the proxy’s interaction with the server may be interrupted anytime without server corruption. Hence, the failure of a tuplespace client will only cause the failure of the client application and tuplespace proxy, but no system failure. At last, the network should be conceived as failover safe on itself. This is possible, e.g. by ensuring a connectivity degree of two. In conclusion, a mechanism has to be found to run failover safe tuplespace servers. Then the tuplespace middleware will be failover safe, too.
xTSpaces’ scope is conceived as to build upon a service of native tuplespaces. Yet, servers of the native tuplespace are not failover safe, as shown above. Therefore, they have to be combined and coordinated in a certain manner, in order to build a logical server that is failover safe. Hence, a layer (S-Layer) is elaborated. It builds upon a native tuplespace’s services, in order to provide a failover safe tuple space middleware. Coordination is achieved by the means of protocols. The clients’ proxies actively participate in this, because it has been shown that clients do not have to be thin. The term sEntity refers to the combination of a server of the native tuplespace together with server components dedicated at implementing functionality of the S-Layer. The efforts aim at conceiving a logical server that is failover safe. It will be called sServer. In the following, it is shown how an sServer is built up by sEntities and how the shape of the protocol is. It should be pointed out, that this is not the only strategy for a sServer.

Redundancy (Lee et al. 1990) is a promising approach for the conception of a sServer. Static redundancy means that there are at least two sEntities that concurrently run and perform the same task. However, then the consistency of the sEntities’ content is difficult to achieve. On the other hand, the client proxies shall not have to interact with all of the sEntities. This demand comes from performance considerations and from upper assumption, that a proxy’s interaction with the sEntities may be interrupted anytime. Therefore, static redundancy proves to be cumbersome to apply to this problem domain.

Dynamic redundancy means that there is a primary entity (primary) that provides the service. In addition, there is a secondary entity (secondary) which takes over in providing the service if and only if the primary fails (primary failure). Three major problems arise if dynamic redundancy is applied. The first one is, that in case of primary failure, the secondary has to know the last state and content of the primary. Otherwise, tuples would get lost. Therefore, the primary’s content is mirrored or its changes are logged. In both cases, data has to be stored on an entity that still has to be accessible after primary failure. For simplicity and performance considerations, it is assumed that the secondary tackles the mirroring or logging of the primary.

The second problem is how the failure of the primary can be hidden from the client. This includes that primary failure should not evoke a brief unavailability of the sServer’s services. But if logging is applied, the logs have to be processed in case of primary failure. This is expected to last a certain amount of time during which the sServer’s services are unavailable. Therefore, mirroring the primary is indispensable. In addition, only the client’s proxy is able to hide the primary failure from the client applications. Hence, the proxy has to keep track of the primary’s and secondary’s state.

The last problem is how the secondary is informed about the primary failure. Because the clients’ proxies have to keep track of the sEntities’ state anyway, they are used to notify the secondary of a primary failure event. Since all of the three major problems can be solved, xTSpaces applies dynamic redundancy in order to achieve failover safety.

Redundancy is provided by reduplication of sEntities. In order to conceive a protocol that is as simple as possible, xTSpaces’ sServer is made up of two sEntities. The parts of the protocol which are relevant for these two sEntities, are called server side protocol. The design of the server side protocol is rather simple, because it is made use of the symmetric arrangement of the two sEntities. The parts of the protocol which are relevant for the clients’ proxies, are called client side protocol. Its main focus are the state transitions of the proxies. Apart of that, the client side protocol is also involved in coordinating the two sEntities.
A shared memory among sEntities and proxies is needed for coordination. It is realized as one space on the primary and the secondary. Tuples stored in the shared memory describe the state of the sEntities. The tuple (PRIMARY) is stored on the primary, so that it becomes discernible for the proxies which sEntity is the primary. Another tuple (BLOCKED) on the primary is set, if the secondary is actively mirroring the contents of the primary. If the primary's content is changed meanwhile, the consistency of the two sEntities cannot be guaranteed any more. Therefore, the proxies have to wait for the deletion of this tuple before continuing their work. Yet another tuple (STANDBYADDRESS) on the primary points to the secondary's address. Hence, proxies which only know about the primary get aware of the secondary. The last type of tuple (TOBESHUTDOWN) denotes that the secondary wants the primary to be shut down. In this situation, the primary's content is not mirrored any more, so that the proxies should continue their work on the secondary. The tuple (TOBESHUTDOWN) is set by the secondary, if the tuple (PRIMARYDOWN) is written to the secondary's space by a proxy. The idea behind is that the primary's services could be briefly unavailable. In consequence, some proxies may notify the secondary of this situation. Other proxies possibly do not know about the problems on the primary. Hence, even if the primary's service is available again afterwards, these proxies will still know that a change has occurred. It is pointed out, that the protocol entities exchange the control data in-band, that is in form of tuples. Therefore, the S-Layer only relies on the native tuplespace, but no deployment specific services are exploited.
Upper figure illustrates xTSpaces’ S-Layer server side protocol. A secondary’s agent (Synchronizing agent) permanently mirrors the primary’s content. Therefore, the secondary is client of the primary’s tuplespace service. Such an agent is initialized during the booting process of the secondary. It registers to all events on the primary and thus keeps track of the primary’s content. When the secondary is notified of a primary failure, it ceases to mirror. Apart of that, the primary and the secondary have one watchdog agent each. The primary’s watchdog is started while booting the primary. It shuts the primary down, when it is reported to have failed. The secondary’s watchdog is started while booting the secondary and waits for notification of a primary failure. In such a case, it tries to notificate the primary of its failure. Afterwards, it starts a process which will reconfigure the secondary as primary. If primary failure notifications come late, they are ignored. The first step of booting a sEntity is to examine whether it is a primary or a secondary. If it is a secondary, the primary is blocked, in order to start the mirroring and to reconfigure the primary.
The server side protocol ensures that whatever sEntity fails, the other one is always marked and working as primary. By distributing coordination tasks to the participating entities, the S-Layer’s protocol achieves the sServer’s failover safety. Coordination tasks are dedicated to the clients’ proxy, too. They work according to the client side protocol which is shown in lower figure. The term "Toggle" depicts, that the proxy changes its assumption about which sEntity is primary and which sEntity is secondary.

Figure 5: Client side protocol state-chart of the S-Layer

Apart of supporting the coordination of the sEntities, the client side protocol mostly covers the following problem. The client proxy has to keep track of the sEntities’ state, so that it is able to redirect calls on its primitives to the valid primary. Therefore, three super states are discernable in the state-chart. The state of the sEntities are directly modeled by the one server mode and two server mode. During initialization, the proxy examines the number of active sEntities and their working state. It is possible that primary failure has not been discovered by other proxies, so that none of the sEntities are primaries. In this case, the remaining sEntity is notified.
Primitives are performed in two states of the client side protocol. The proxy comes with an agent that blocks primitives, if the primary is reported to be blocked. State transitions between the one server mode and two server mode are conceived as follows: If a secondary registers itself on the primary, the state changes to the two server mode. Through interaction with the primary, the proxy may realize that the primary failed and its service is down (downReport). In this case, the secondary is notified. If the secondary reacts by promoting itself to a primary, the proxy switches to the one server mode.

In conclusion, the S-Layer draws failover safety from dynamic redundancy and its distributed management protocol. As mentioned above, the protocol is based on the assumption, that the clients do not have to be as thin as possible. Otherwise a centralized management protocol would have to be worked out.

SCALABILITY

The chapter of the requirement analysis states, that "the core requirement of scalability (...) reflects both, scalability of tuple size and tuple number". In other words, the tuplespace implementation has to be scalable in regard of the size of single tuples and the number of tuples to be stored. Only the latter issue is addressed by xTSpaces, because it is expected to be the major concern in scalability. Therefore, a single but very large tuple may cause some problems to xTSpaces. Chapter V gives more details about this. In this section, it is discussed how scalability is achieved in regard of \( n \) tuples. The examination of scalability deals with server and client resources, that is computing and memory load, and it deals with the performance of primitives. The primitives are classified into reading primitives \( (\text{rd, in}) \) and writing primitives \( (\text{out}) \). The resources have to scale with \( O(1) \) or \( O(\log n) \), the primitives’ performance with \( \Omega(1) \) or \( \Omega(1/\log n) \). The aim is full scalability, that is \( O(1) \) and \( \Omega(1) \).

If the amount of employed server resources has to be independent of the problem size, several servers together have to provide the service needed. These servers are called dEntities, and the logical server which they form, is called dServer. The number of dEntities per dServer will be refered to by its number of dEntities \( e \). A dEntity may be an sServer or a native tuplespace server. The dServer has to preserve a dEntity’s attributes and functionality. In addition, it shall provide a scalable service. To do so, the dEntities have to be managed, and perhaps they have to be coordinated. A client application runs a client proxy to the dServer. It hides the dServer’s constitution of several dEntities and their space remoteness to the client application. The client proxies and the dEntities together form the D-Layer.

Tuples are distributed among the dEntities. In addition, it could be reasonable to replicate certain tuples on several dEntities. However, the tuples’ distribution and replication has to be kept transparent to the client application. It is pointed out, that this concept is similar to caching strategies on multiprocessor systems (Giloi 1993). However, storage and retrieval of tuples is fully associative. Aside, retrieval has to rely on templates, that only come along with a subset of information of the tuple to retrieve. On the other hand, more than one distinct tuple may fit to one retrieval request. Therefore, it might be fruitful to borrow from the existing caching strategies, but these concepts have to be adjusted to the specific problem domain of tuplespaces. In the following, several concepts are presented. It is examined whether they provide a means of resource and performance scalability.
Figure 6: Simple distribution and retrieval schemes

The simplest way of distributing tuples among dEntities is shown in upper figure. MU stands for a dEntity. A client proxy performing reading and writing primitives is denoted by R and W respectively. Figure 6’s upper scheme illustrates a write-all-read-once strategy (WARO), the lower one stands for a write-once-read-all strategy (WORA). WARO’s reading primitives and WORA’s writing primitives scale, WARO’s writing primitives and WORA’s reading primitives however do not. As far as the primitive performance is concerned, WARO is more adequate than WORA, because client applications are expected to use reading primitives more frequently than writing primitives. However, WARO leads to no scalability of both kinds of server resources, computing and memory load. Yet with WORA, the server’s memory usage scales. Still, server computing load does not scale with WORA due to a not scaling amount of reader’s queries. In conclusion, WARO is not viable, but WORA is usable as a foundation for a more sophisticated concept. Such a concept has to address the non scaling server computing load and in consequence the non scaling reading primitives.

In WARO and WORA, there is always one kind of primitive that is at $\Omega(1/n)$. Therefore, (Corradi et al. 1995) suggest that the dEntities are structured like a tree. By doing so, both reading and writing primitives are at $\Omega(1/\log n)$. Lower figure illustrates such a scheme.

Figure 7: A scheme of hierarchical storing and retrieval
MU still refers to dEntities, R1/R2 and W1/W2 still refers to proxies that perform reading and writing primitives. In the figure, each non leaf dEntity has two children, but this concept works with whatever number of children. Every proxy is associated with one dEntity. If the proxy has to execute a primitive, it is to be performed on the associated dEntity and on all of its ancestors. If the primitive is successful before attaining the root dEntity, it finishes prematurely, like W1 and R1 in the figure. Otherwise, the primitive will go up to the root, like W2 and R2 in the figure. Therefore, reading and writing primitives scale with $\Omega(1/\log e)$. If the number of the dServer’s dEntities is chosen according to the number of tuples, the primitives scale with $\Omega(1/\log n)$. The primitives’ performance is near $\Omega(1)$, if a topology of dEntities and client proxies is found which fits to the usage profile of the dServer. However, such a topology cannot be found statically with an ad-hoc set of participants. Dynamical structuring as discussed in (Rowstron 1998) is possible, but proves to be cumbersome to implement and it comes along with considerable overhead. In addition, server resources only scale in lower layers of the tree. E.g. the computation and memory load of the root dEntity is at $O(n)$. In conclusion, this concept is only efficient if and only if an adequate topology can be found. The only way to do so without engaging into dynamical structuring of the topology, is to let the client applications decide which dEntity to associate with. However, this is contradicted by demand for transparent distribution.

For achieving scalability in the D-Layer, none of the mentioned concepts have been proven to be directly applicable without alteration. It is pointed out, that only the latter concept of a hierarchy of dEntities makes use of additional information. E.g. in case of dynamic structuring, such information is which client proxies cooperate on the same tuples. Hence, the client application’s profile is made use of, in order to provide a quasi scalable strategy. Nevertheless, none of the upper concepts exploits information about the used tuples and templates. The condition of exploiting such information is that tuples are no longer black boxes. This condition holds in all existing tuplespace implementations. Tuple objects are made up of fields, and those fields implement tests on equality and matching. The proxies and the dEntities can and should make use of this.

It seems to be a promising idea to build upon WORA while ameliorating its reading primitives’ performance to $\Omega(1)$. Then, the server’s computational load would fully scale, too. Hence, the D-Layer would be able to be successfully conceived. The reading primitives’ performance has to be ameliorated by a priori knowledge of which dEntity to address. What is needed, is a prediction service that returns a permutation of dEntities. Therefore, the proxy tries to retrieve the tuple in the order induced by the permutation. If such a retrieval is in $O(1)$, the D-Layer will be fully scalable. If it is in $O(\log n)$, the D-Layer will still be scalable. The idea is to build such a prediction service by making use of a white box view of tuples. In such a case, the prediction service gets a tuple and determines the permutation of dEntities. This approach induces granularity of tuple distribution on the tuple level. Therefore, the granularity is finer as the one on client proxy level, as supposed by the prior hierarchical scheme.
The earliest concept based upon this idea, is to build a prediction service that works with hashes. A hash function assigns to each tuple a hash value. The hash value is taken modulo to e. The dEntities are indexed, so that the tuple has to be stored and retrieved on the dEntity with the corresponding index. Thus, the prediction service only returns one dEntity and no permutation. If the tuple is not found, it is guaranteed that the other dEntities will not contain it, too. It is an interesting fact that the performance of reading primitives is $\Omega(1)$, if the hash function is good enough to equally distribute the tuples among the dEntities. However, this concept is only applicable with actual templates. The hash value assigned to a formal and semantic template, cannot be equal to the hash values of all tuples it matches. Otherwise, the hash values of every tuple would have to be equal, because they would have to equal the hash value of the semantic template that matches every tuple. However, in case of templates always being actual, this concept is applicable. A further problem is, that the private attributes of a tuple’ s fields are not disclosable to the hash function. Therefore, a good hash function has to be generic, that is every field has to come along with its own hash function. In conclusion, WORA with a hashing prediction service provides full scalability, if retrieval is only done with actual templates. For retrieval with formal or semantic templates, the concept is just as bad as WORA on its own.

The hashing concept is promising, but it fails to deal adequately with formal and semantic templates. What is needed, is a concept which is applicable to actual, formal and semantic templates. Therefore, the theory of signatures should be applied. It naturally extends the hashing semantics to formal and semantic templates. In addition, tuples’ fields do not necessitate their own hash functions. The function $\varphi^-$ assigns a permutation of dEntity indices to a tuple’ s signature. The function is extended for tuples by $\varphi^-(t):=\varphi^-(t^-)$. When a tuple $t$ is to be stored, it is written to the dEntity implied by first index of the permutation $\varphi(t)$. Therefore, such an extended hash function perfectly fits into the prediction service concept, that is identical to $\varphi^-$ then. This concept still lacks an appropriate mechanism to compute $\varphi^-$. The idea is, that $\varphi^-(\sigma)$ and $\varphi^-(\rho)$ are very similar, if signature $\sigma$ matches signature $\rho$. The closer they are in the signature tree, the more similar $\varphi^-(\sigma)$ and $\varphi^-(\rho)$ should be. It is a highly non trivial task to design a $\varphi^-$ that complies to these demands. Since the concept of using signatures only recently evolved, more research work has to be done in this field. One section in chapter V is dedicated to this topic. Nevertheless, it remains the most promising approach for implementing scalable tuplespaces.

Because the upper concept is not mature enough yet, the implementation of xTSpaces is built upon WORA with a prediction service based on hashing. Therefore, the D-Layer does not scale on reading primitives with formal and semantic tuples.

The protocol on the D-Layer is fully on the client side. The dEntities do not know about each other, so that the clients’ proxies have to provide the full DLayer functionality on their own. A field has to provide its own hashing function, so that a tuple’ s hash value can be evaluated by taking them into account. If a proxy has to store a tuple, the tuple’ s hash value determines, which dEntity it is written on. If a tuple is to be retrieved by an actual template, the same procedure determines, on which dEntity to look for it. If a tuple is to be retrieved by a formal or semantic template, it is looked for on every dEntity.
Still, an additional mechanism has to be set in place, so that the client proxies know about the number of dEntities and about their network addresses. Therefore, an additional management service is run. It introduces a logical address scheme for dEntities. Primitives for associating logical addresses to network addresses are provided, too. When a dEntity is booted, a server is started and its network address is registered to the management service through these primitives. In addition, the service supports resolution of logical addresses to network addresses (Logical Address Resolution) and vice versa (reverse LAR). Therefore, the management service provides all the functionality that is needed by the proxies. Since the D-Layer should not alter attributes of the underlying tuplespace implementation, the management service is built upon a dedicated server, the so called mServer. Such an mServer is booted like dEntities, so that it has to register on itself. The primitives for LAR, rLAR and address binding are implemented in client proxies to the mServer. Therefore, the client application’s proxy to the DLayer has to use a mServer proxy, in order to retrieve the needed information about the dEntities.

ARCHITECTURE

xTSpaces builds upon a native tuplespace implementation (native tuplespace). The application programming interface (API) of such a native tuplespace implementations (native API) may differ. However, xTSpaces is conceived to provide a native API independent xTSpaces API to its users. This can be done by placing an abstraction layer above the native tuplespace implementation. In such a case, the program logic for xTSpaces’ failover safety and scalability would use the abstraction layer’s API. Hence it would be interchangeable regardless of which native API is used. But the drawback of such an abstraction layer is, that it is likely to provide a meagre API, the smallest common denominator of all native APIs. The other approach is to use characteristics of the native API in all parts of the program logic. But then again xTSpaces has to be implemented for each native API.

Upper sections’ conception of xTSpaces concludes in using two layers, the SLayer and the D-Layer. Even though their layering can be freely chosen, the D-Layer shall build upon the S-Layer in order to achieve higher availability. This design decision is induced by the following example: If the D-Layer works with four dEntities (e=4) and the S-Layer with a primary and secondary sEntity (s=2). A dServer fails if one of its dEntities fails, a sServer fails if all of its sEntities fail. If the S-Layer is built upon the D-Layer, an sEntity is a dServer and a dEntity is a native server. Then, if two native servers fail, the probability that both dServers and in consequence the sServer fail, is more than 50%. On the other hand, if the D-Layer builds upon the S-Layer, an sEntity is a native server and a dEntity is an sServer. Then, if two native servers fail, the probability that one sServer and in consequence the dServer fail, is about 15%. From the reliability point of view, a dServer is a serial system and a sServer is a parallel system (Lee et al. 1990). It can be proven that a serial system should build upon a parallel system, so that the D-Layer has to build upon the S-Layer.

Yet another design decision is to unite the S-Layer with the abstraction layer. This decision is triggered by the S-Layer’s need of every kind of functionality it can get from the native API, e.g. the eventing. The D-Layer is not so challenging towards the S-Layer’s API(S-API). Its API (D-API) is identical to the S-API, because dServers are designed to preserve the attributes and functionality of dEntities. Therefore, it seems adequate to divide xTSpaces in a part that depends on the native API and a part that is independent. Since the S-API abstracts from the native API, the D-Layer has only to be implemented once. However, the S-Layer has to be implemented for each native tuplespace implementation used.
The managing service in the D-Layer is taken out of it and builds the \textit{M-Tier}. The API of the M-Tier (\textit{M-API}) consists of primitives for accessing its services. An overview of the resulting architecture is given in the figure below. IBM’s TSpaces is taken as an example of a native tuplespace implementation.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{xtspaces_architectural_overview.png}
\caption{xtSpaces architectural overview}
\end{figure}

The xtSpaces API consists of the M-API and the D-API. Common xtSpaces client applications are expected to exclusively use the D-API. However, the M-API can be used to keep track of the dEntities’ registration. Since the mServer is built upon a sServer, the M-Tier uses the S-API. It is important to point out, that the D-API’s primitives provide a scalable and failover safe service. This is due to the conception of dServers to be failover safe if its dEntities are, too.

Upper figure illustrates the layering of xtSpaces’ client side. The layers’ functionality is furnished by proxies to sServers, the mServer and dServer. The server side protocol of the D-Layer and M-Tier is void. Therefore, the server side architecture only consists of a S-Layer link between two sEntities. The following figure illustrates an example of how a xtSpaces deployment may look like. In contrast to upper figure, distribution is made explicit. Here again, the underlying native tuplespace is IBM’s TSpaces.
Six TSpaces servers and three TSpaces proxies are connected via TCP/IP. In the upper part of the figure, it is shown how the six native servers form a dServer with two dEntities and an mServer. Each sEntity builds upon a native server. Every pair of entities is linked together to an sServer by the S-Layer server side protocol. Apart of the booting procedure, the components of an sServer are identical to those of a mServer and dEntities. The dServer is logically formed by a pair of dEntities, even though the dEntities do not know about each other. The lower part of the figure illustrates client applications of xTSpaces. The xTSpaces Monitor is an administration tool for monitoring and using the managing services of the M-Tier. Therefore, only the M-API is used. No proxy to the dServer runs on the monitor. In addition, a proxy to the mServer’s sServer is run, because the proxy to the mServer uses the S API. Finally, two clients are shown that use the D-API, in order to use xTSpaces’ services. Each of them has to run a proxy to the mServer, because the proxy to the dServer uses the M-API. Hence, the clients run all three types of proxies. From the clients’ viewpoint, xTSpaces is a middleware that enables coordination and cooperation. The clients are linked together on a layer above xTSpaces.

Upper figure abstracts from the fact that sEntities use native proxies to native servers, mServers use S-Layer proxies and dEntities use M-Tier proxies. It appears that use relations go through system borders. Therefore, lower scheme of use relations in xTSpaces is by far more complicated than the architectural overview figure. Like the two prior figures, it assumes that IBM’s TSpaces is the naïve tuplespace.
Component A uses component B if and only if A lies on top of B. The client side proxies are written in bold. The three use relations that go through system borders, are made explicit by arrows. The transitivity of xTSpaces’ use relations is made obvious in the figure. The numbering of components induces an ordering. xTSpaces is implemented step by step according to the ordering.

IMPLEMENTATION

A reference implementation of xTSpaces has been written in Java. It builds upon IBM’s TSpaces as native tuplespace. TSpaces was chosen for its compactness. The S-API and the D-API are identical. They contain the primitives inp (take), rdp (read), out (write) and a bulk version of rdp (scan). In addition, there are two primitives for creating (addTupleSpace) and deleting (removeTupleSpace) spaces. The S-Layer and D-Layer do not guarantee a certain order of retrieval, as e.g. FIFO. A consistent view of the tuplespace’s content is not guaranteed, either.

The M-API is made up of primitives for address binding (bind and unbind), logical address resolution (LAR and rLAR) and dServer management (getNumberOfDEntities). In addition, a management facility for external services is included (registerService, unregisterService and getServiceAddress). The facility does not know about the purposes of the services it manages, so that any service can be registered. xTSpaces’ monitor is included as an add-in. It offers remote shell-like command line access to the M-API.

The concept of logical and network addresses is encapsulated in classes (xLogicalAddress and sServer). Latter class is part of the S-Layer and hides the sEntities’ details from the M-Tier and D-Layer. The S-Layer abstracts from tuple (xTuple) and fields (xField) of the native tuplespace. The S-Layer’s fields implement a hash function which is based upon the corresponding polymorph Java function (hashCode).

No prediction service is implemented for reading primitives. Therefore, a random permutation of dEntities is computed, if tuples are retrieved by formal or semantic templates.
PERFORMANCE

xTSpaces introduces protocols and layers, in order to fulfill the requirements of being failover safe and scalable. Yet, their efficiency has to be measured, together with the overhead of the S-Layer and D-Layer protocol. Therefore, performance testing is carried out by the means of benchmarks.

A precise overview of the testing scheme and the benchmark is found in the appendix B. The test is expected to shed light upon the following two questions:

1. Up to which degree does the registration of a secondary or primary failure affect the performance of xTSpaces?
2. If an adequate number of dEntities is used, does xTSpaces fully scale with the problem size?

Figure 11 illustrates empirical data gathered to answer the first question. It is shown that the primary failure briefly leads to an inferior performance, just like the registration of a secondary does. The scanning intervals are too coarse to visualize that xTSpaces’ service is down for a certain amount of time. The conclusion from this performance test is, that primary failure and secondary registration affects xTSpaces’ performance only to an acceptable degree.

![Figure 11: Testing xTSpaces' failover safety](image)

Figure 12 shows the test data collected on dServers with one, two and three dEntities. According to the testing scheme, the dServer is increasingly put under stress. Therefore, a scalable dServer would have a proportional behaviour. However, the figure illustrates that this is true only for a very short range. The more dEntities, the longer the range. If the dServer is put under stress further on, the performance falls considerably.

The key for understanding latter fact is, that only one benchmark instance has been used for the test. Thus, the more intense the stress test becomes, the more the client side becomes the system’s bottleneck. Nevertheless, the test indicates that the use of a dServer provides a scaling service, if the dServer is adequately scaled to the problem size.
Figure 12: Scalability of xTSpaces
V. Future Work for xTSpaces

EXTENSIONS

In xTSpaces, some concepts are not yet included, even though they are well understood. Most of these concepts deal with optional requirements of a tuplespace middleware in an eCommerce environment. The concepts were neglected at first, because xTSpaces is focused on the core requirements of failover safety and scalability.

Even though it is possible to work with xTSpaces for XML tuple exchange, xTSpaces lacks built-in XML fields. Such a field should be initialized by a DOM document node. For including XML tuples, most of the work to be done is in implementing their matching. On the other hand, the only semantic tuple that is supported by xTSpaces yet, is the one with the signature $\perp$. XML tuples are able to naturally represent semantic tuples. Therefore, a built-in XML capability is very desirable.

The S-API and D-API should be enhanced by adding the blocking primitives $\texttt{in}$ and $\texttt{rd}$.

xTSpaces still lacks of a security concept. It is very difficult to use native security concepts because they are varying in their conception. xTSpaces has to be redesigned according to its own dedicated security concept.

xTSpaces is implemented in Java. Therefore, it does not comply to the optional requirement of language interoperability. What is needed, is e.g. a proxy component to xTSpaces’ API, with the component technology being language independent. It has to be paid attention to the component technology being portable, too.

Client applications are able to directly work on the S-API, so that they do not have to use the D-Layer. However, it is not possible to use the D-Layer without the S-Layer. The reason is that the D-Layer uses the S-API. Therefore, a dummy abstraction layer should be built upon several native APIs. The abstraction layer’s API should be identical to the S-API, so that the D-Layer can be used without the S-Layer.

Yet another idea is to implement the S-Layer on other native tuplespaces than TSpaces, e.g. JavaSpaces. Then it becomes possible to use xTSpaces as gateway between different tuplespace implementations. Therefore, xTSpaces unifies heterogenous subnets of tuplespaces. In such a case, a dServer can even be constituted of several dEntities with heterogenous sServers.

PREDICTION SERVICES BASED ON SIGNATURES

The key to full scalability is to find an appropriate prediction service $\varphi$, based on the theory of signatures. However, the problem is not well understood yet, and mechanisms computing $\varphi$ seem to be complex. But even though being complex, it is favourable because of its fine granularity model. E.g. the scheme of hierarchical structuring of dEntities comes with a similar complexity, but its coarse granularity model will not fit most eCommerce environments.
In general, there are two kinds of mechanisms for \( \varphi \), that are transparent and not fully transparent ones. Latter appears to contradict the demand of transparency of distribution and replication on the D-Layer. But not fully transparent is meant to be a concept, that enables the client application to provide additional information about tuples of which signatures it will use. Hence, distribution and replication is hidden from the client application. Even more, the client does not know whether the additional information that it provided, is taken into account or not.

One not fully transparent mechanism is introduced in the following. The client application is able to notify the D-Layer, that it will mainly use a subset of signatures \( \xi \subseteq \mathcal{X} \). The coarse representation of the subset is given by \( \bot \xi \). Therefore, two primitives are added to the D-API, one for registering and one for unregistering to \( \bot \xi \). Afterwards, this is taken into consideration by primitives working on tuples or templates, the signature of which being matched by \( \bot \xi \). It is expected that the D-Layer should be able to optimize \( \varphi \) by using this additional information. One way for doing so, is to directly associate a dEntity to an allocated signature \( \bot \xi \). Then, the performance of reading primitives is \( \Omega(1) \), if the templates’ signatures are matched by an allocated \( \bot \xi \). The allocated signatures have to be tracked by an additional service. The allocation primitive has to query the service, whether \( \bot \xi \) already has been allocated or not. It shall also be able to check, whether a signature has been allocated that matches \( \bot \xi \) or that is matched by \( \bot \xi \). However, if two client applications cooperate through the tuplespace, they both have to allocate an adequate signature or no signature at all. If one client allocates and the other does not, the mechanism as it is will not work. Yet another problem is allocating a signature \( \bot \xi \) that matches or is matched by an already registered signature \( \rho \). For consistency considerations, the concept demands for the same dEntity associated to \( \bot \xi \) as for \( \rho \). But in this case, scalability of the server resources is not guaranteed any more.

It appears that not fully transparent mechanisms are more efficient than transparent mechanisms to evaluate \( \varphi \). But it has still to be resolved in the future, how much the client applications are able to care about the signatures they use. Therefore, several fully transparent mechanisms are presented in the following. It is pointed out, that they can be combined for achieving better results.

The easiest mechanism is to rely on hashing, yet hashing on signatures. But it was shown in prior chapter, that hash values of signatures cannot express their ordering. The idea is to define a hash function on formal signatures. Therefore, an actual signature gets exactly the same hash value as its matching formal signatures. However, semantic signatures are not included in this concept. Even worse, most client applications are expected to use similar signatures, so that the hashing’s granularity proves to be too coarse. Hence, hashing on formal signature level is not as promising as hashing on tuple level. It is still to be researched, whether both concepts can be merged.
Yet, there is another approach based on taking a formal signature as a representative for the actual signatures it matches. The idea is to conceive a formal signature associative service. It is called signature lookup service and returns a permutation of dEntities. This is because tuples with the same associated formal signature have not to be stored on the same dEntity. It even becomes possible to use the tuple hashing method for storage and retrieval of actual tuples. Retrieval by formal templates would use the signature lookup service, in order to know where the tuple is likely to be found. To provide such a service, the signature lookup keeps track which tuple is written to and deleted from which dEntity. Based upon this observation, the service is able to compute the probability of the tuple to be found on a certain dEntity. This is reflected in the ordering of dEntities in the permutation. Still, this approach comes along with unsolved problems. It is unclear how $\Omega(1/\log n)$ can be achieved with such a probability profile. In addition, semantic templates are difficult to include into this scheme. The calculation of a permutation is complex, if retrieval is done by a semantic template. Lastly, the signature lookup service has to be scalable, too. This problem can be circumvented by assuming that the number of formal signatures used is at $O(\log n)$.

All of upper mechanisms have one point in common: If tuples are stored on a dEntity, they remain there until their deletion. Therefore, prior mechanisms are called static. On the other hand, dynamic mechanisms are able to move a tuple from one dEntity to the other. Nevertheless, such an action is hidden to the client application. It is still to be researched, how dynamic mechanisms may optimally adapt to the usage profile. This includes the problem, under which circumstances a tuple should be moved or replicated. In addition, such mechanisms necessitate a closer coordination of D-Layer proxies and dEntities. It has to be examined, whether the mechanisms’ protocol has to be extended to the server side or whether it requires a centralized administration entity. In the second part of this section, one concept for a dynamic mechanism is presented.

Let $\xi$ be the set of signatures of the tuples on a dEntity. Then every dEntity is represented by a signature $\bot_{\xi}$. If a new tuple with signature $\sigma$ has to be stored, it is stored on one or more dEntity with $\bot_{\xi} \leq \sigma$. The same principle is applied for deletion. If there is no dEntity with $\bot_{\xi} \leq \sigma$, the tuple shall be stored on the dEntity with $\bot_{\xi}$ being as "similar" as possible to $\sigma$. A clear definition of similarity of signatures has still to be worked out. Intuitively, it is clear, that e.g. $\bot_{\xi}$ is similar to $\sigma$, if $\sigma$ is parent of $\bot_{\xi}$. If the number of tuples on one dEntity grows to large, $\xi = \xi_1 \cup \xi_2$ is split into two disjunct subsets $\xi_1$ and $\xi_2$, with the signatures of $\xi_1$ and $\xi_2$ being as dissimilar as possible. If the number of tuples on two dEntities grows to small, the tuples shall be unified on one dEntity. It is suitable, that the $\bot_{\xi_1}$ and $\bot_{\xi_2}$ are very similar. The prediction service returns its permutation of dEntities according to their $\bot_{\xi}$. However, it is still unclear whether this concept leads to a prediction service that complies to the scalability demand. The overhead of a D-Layer implementation upon this dynamic scheme is still to be examined.

It is concluded that more research work has to be done in this field. It is aimed at solving the problem of finding an adequate prediction service based on the theory of signatures.
UNRESOLVED PROBLEMS

There are still some points which are part of xTSpaces’ scope but have not been realized yet. Scalability is a real concern. The scalability of the number of tuples has not been fully addressed. The most promising approach of using signatures is not mature for its application, yet. The concern of the scalability of a tuple’s size is not included at all. It can be argued that the client application should split oversized tuples itself, but this lacks transparency. Therefore, the splitting of oversized tuples should be performed by the client’s proxy.

Yet another problem is that the number of dEntities cannot be altered after having started the tuplespace service. All of the proposed scalability mechanisms fail to provide a means of dynamically adding or removing dEntities. Only prior section’s dynamic mechanism based on signatures, complies to this demand.

An extension of the S-Layer protocol for supporting more than two sEntities per sServer, is still to be worked out. It is unclear, whether the extension S-Layer protocol will get much more complex in this asymmetric case. In addition, the S-Layer uses the eventing scheme of the native API. But it still has to be worked out, how the S-API can include an eventing scheme.

Finally, transactions have been left out of consideration. It is questionable how native transaction mechanisms have to be used, in order to compile transactional mechanisms on the S-Layer and D-Layer.
VI. Conclusion

Recently, eCommerce applications have emerged as a new class of Web applications. Their elaboration is highly non trivial in itself. Therefore, eCommerce applications should be supported by a middleware that provides mechanisms for coordination among them.

In this contribution, it has been examined whether the tuplespace concept is suited to build the foundation for such a coordination middleware. It has been shown, that existing tuplespace implementations as they are, are not well suited. Especially the demands for failover safety and scalability are not met. Concepts for compliance to these two demands have been elaborated. It appears, that the demand for failover safety can be met, but the problem of scalability still remains not fully solved. As far as the elaborated concepts are applicable, they have been taken into consideration while developing the reference implementation xTSpaces. Furthermore, the concept complements existing tuplespace implementations.

Nevertheless, xTSpaces on itself is not yet applicable as a coordination middleware in an eCommerce environment. It is still to be worked out, how scalability can be achieved. In addition, other demands like transactional and security concepts, have to be integrated within. More research work shall give an answer to the question, whether tuplespaces can build a middleware for coordination of eCommerce applications.
VII. Literature


A. Appendix: Usage of xTSpaces

INSTALLATION

xTSpaces runs on a Java Virtual Machine with a version number of at least 1.1. xTSpaces consists of two packages. The domain.xTSpaces package contains the implementation of the basic xTSpaces functionality. In addition, the domain.xTSpaces.addIns is made up of supplementary services and utilities that shall ease the usage of xTSpaces. The two packages are included in the xTSpaces.jar. Apart of this, the TSpaces tuplespace has to be installed, too. All necessary TSpaces files can be found at http://www.alphaworks.ibm.com/tech/tspaces. TSpaces is the underlying technology chosen due to its most convenient API and attributes. xTSpaces uses it as the native tuplespaces API. That is why a correct installation of TSpaces is necessairy, even though its usage is absolutely transparent for the clients of xTSpaces.

In the following, it is exemplary shown how to set up and run xTSpaces services. The program examples are completely in Java. For a detailed documentation of the interfaces and command lines, the interested reader has to refer to the API documentation.

RUNNING THE SYSTEM

• sServer:

Sometimes there is no need for xTSpaces’ supplementary services and for compliance to the scalability demand. In such a case, a single sServer can be set up and the tuplespace clients work through the S-API. In order to set up a sServer, a pair of sEntities is needed. sEntities come along with an explicit HTTP service for observing the entities’ tuple content. F.e. these command lines could be used for starting a pair of sEntities on one server machine. They use the ports 2000 and 3000 for the tuplespace service and the ports 2001 and 3001 for the HTTP service:

java domain.xTSpaces.sEntityBoot -tp 2000 -hp 2001 -S 1 -S2 127.0.0.1:3000
java domain.xTSpaces.sEntityBoot -tp 3000 -hp 3001 -S 2 -S1 127.0.0.1:2000

If the primary server on port 2000 crashes, it can be restarted with the command line:

java domain.xTSpaces.sEntityBoot -tp 2000 -hp 2001 -S 1 -S2 127.0.0.1:3000

As a consequence, the next call of a S-API primitive lets the secondary server on port 3000 become the new primary server.

• Clients of the S-API:

Clients rely on the S-API in order to interact with a sServer. At the beginning, a proxy to the sServer has to be instantiated. F.e. a client program should instantiate a proxy to upper server by the following program line:

sServer mySServer=new sServer("ts=(127.0.0.1:2000,127.0.0.1:3000)");
sAPI mySProxy=new sTSpaces(mySServer);

Afterwards, the client can interact with the tuplespace through the S-API primitives implemented in the proxy. The example of a function implements a rendez-vous. Therefore, a proxy to a sServer has to be given along with the caller’s name and the name of the person with which to synchronize:
```
function rendezVous (sAPI mySProxy, String myName, String yourName) {
    String ourSpace="Rhonda";
    xTuple myToken=new xTuple(new xField(myName),new xField(yourName));
    xTuple yourToken=new xTuple(new xField(yourName),new xField(myName));
    mySProxy.write(ourSpace,myToken);
    while (mySProxy.take(ourSpace,yourToken)==null) Thread.yield();
}
```

**mServer:**

Scalability is a major concern for an eCommerce coordination middleware. Upper proceeding of setting up a sServer can’t comply with this demand. Therefore, a dServer composed of several dEntities has to be set up. For coordinating a dServer, a mServer has to be run, too. It is booted at the very beginning, since dEntities use its M-API. Setting up a mServer is just a sServer that runs a specific service, so that it is set up like a sServer. It takes a pair of sEntities to form a sServer. F.e. these command lines could be used for starting a pair of entities on one server machine. They use the ports 2000 and 3000 for the tuplespace service and the ports 2001 and 3001 for the HTTP service:

```
java domain.xTSpaces.mEntityBoot -tp 2000 -hp 2001 -S 1 -S2 127.0.0.1:3000
java domain.xTSpaces.mEntityBoot -tp 3000 -hp 3001 -S 2 -S1 127.0.0.1:2000
```

An mServer is able to keep track of itself. Therefore, if the primary server crashes, it can be restarted by exactly the same command line. In this case, the secondary becomes the primary.

**Clients of the M-API:**

The usage of xTSpaces normally doesn’t necessitate to care about the M-API. It is only used by dEntities and by the monitoring program that is included in the add-ins. The M-API works with logical addresses, which are used to express services provided on entities. M-API’s primitives resolve these addresses to the network address of the corresponding service. A reverse resolution is possible, too. The following shows how a certain just started dEntity uses the M-API, in order to get known whether it is a primary server:

```
sServer mySServer=new sServer("ts=(127.0.0.1:2000,127.0.0.1:3000)");
mAPI myMProxy=new mTSpaces(new sServer(mySServer);
xLogicalAddress myTupleSpaceLA=new xLogicalAddress("D5");
try {
sServer myPartner=myMProxy.LAR(myTupleSpaceLA);
}
catch (LARException e) {System.out.println("I'm alone");}
```

Note, that the network address of the tuplespace service has to be explicitly given, for both mEntities.

**dServer:**

Setting up parts of dEntities is just like setting up a mServer. However, the addresses of the mServer’s tuplespace services has to be explicitly given. In addition, the one-bound index of the dEntity has to be given, too. The number of dEntities cannot be altered dynamically. Therefore, a dServer is only set up if for each valid index the dEntity is running. Here is the command line for starting the dServer of upper example:

```
java domain.xTSpaces.dEntityBoot -i 5 -tp 4000 -hp 4001 -M ts=(127.0.0.1:2000,127.0.0.1:3000)
```
• Clients of the D-API:

The D-API is by far the most important API of xTSpaces, because its usage is sufficient for
the majority of clients of xTSpaces. The D-API is identical with the S-API, its semantics,
too. The distribution of tuples among dEntities is only measurable by the means of the
performance. The matching of formal templates against tuples can be very expensive, while
matching actual templates against tuples is just as efficient as through the S-API.

Take for example the following program line that instantiates a dServer proxy on upper
mServer:

```java
sServer mySServer=new sServer("ts=(127.0.0.1:2000,127.0.0.1:3000)");
dAPI myDProxy=new dTSpaces(mySServer);
```

The following class implements a distributed stack and uses an acces token as a semaphore.
By using this class, several clients can work on one stack. A possible application of this
example is to manage tasks within clients which are based on the master-worker design
pattern. The implementation isn’t efficient and a distributed stack is generally better
conceived as a RMI service provided by one object. However, the purpose of the example
is to demonstrate the usage of tuplespaces in general, and of xTSpaces in particular:

```java
import java.io.*;
import domain.xTSpaces.*;

interface DistributedStack {
    public void createStack () throws xException;
    public void destroyStack () throws xException;
    public void push (Serializable myObject) throws xException;
    public Serializable pop () throws xException;
}

class DistributedStackImpl implements DistributedStack {
    public static final String STACK_SPACE="MyDistributedStackSpace";
    public static final xTuple ACCESSTOKEN = new xTuple(new xField("DS_Access_Token"));
    private dAPI myDProxy;
    
    private void acquireAccessToken () throws xException {
        while (myDProxy.take(STACK_SPACE,ACCESSTOKEN)==null) Thread.yield();
    }
    private void releaseAccessToken () throws xException {
        myDProxy.write(STACK_SPACE,ACCESSTOKEN);
    }
    private int getSize () throws xException {
        xTuple sizeTuple = myDProxy.take(STACK_SPACE,new xTuple(new xField(Integer.class)));
        return ((Integer)sizeTuple.getField(0).getValue()).intValue();
    }
    private void setSize (int newSize) throws xException {
        xTuple sizeTuple = new xTuple(new xField(new Integer(newSize)));
        myDProxy.write(STACK_SPACE,sizeTuple);
    }

    public DistributedStackImpl (dAPI myDProxy) {
        this.myDProxy=myDProxy;
    }
    public void createStack () throws xException {
        myDProxy.addTupleSpace(STACK_SPACE);
        setSize(0);
        releaseAccessToken();
    }
    public void destroyStack () throws xException {
```
In upper example, it is possible that an instance of the distributed stack crashes after acquisition of the access token. Although the data of the stack would not get lost, other instances would be blocked. Therefore, this example clearly demonstrates that even though xTSpaces provides failover safe services, client services built upon it do not inherit this attribute. Client services have to provide their own mechanisms to comply with such a demand.

- The Monitor:

The monitor is able to track the state of the mEntities and dEntities. A state overview will be periodically written into a HTML file. The following command line would set up a monitor that tracks the state of the server entities from prior examples. The state overview would be updated in c:\xTSpaces_state.html every 10 seconds:

```
java domain.xTSpaces.addIns.Monitor -i 10 -o c:\xTSpaces_state.html -M ts=(127.0.0.1:2000,127.0.0.1:3001)
```

The monitor offers a command line service which allows interaction just as through the M-API. Additionally, it is possible to "ping" services registered and to find out which services are registered, but down. In such a case, they would be unregistered. The following figure shows a state overview created by the monitor.
Figure 13: A state overview generated by the monitor add-in

The red entities are not registered, the green ones are. If a pair of entities is not registered, the server field turns red, too. This is critical, because in such a case xTSpaces is inoperable.
B. Appendix: Performance testing

BENCHMARK

In the following, a benchmark is presented that is used for evaluating the performance of xTSpaces. It is based upon the idea of synchronizing as shown in the usage example of the S-API. Nevertheless, the benchmark is more complex, because several degrees of freedom are needed. They are:

- The number of partners that synchronize (sp)
- The number of tuples exchanged between partners per synchronization (st)
- The synchronizations per initializations ratio (si)

Apart of that, the number of running benchmark programs can be altered, too. Finally, the type of tuplespace implementation is chosen among IBM’s TSpace, xTSpaces’ S-Layer and xTSpaces’ D-Layer.

The benchmark source is as follows:

```java
import net.obreiter.xTSpaces.*;
import com.ibm.tspaces.*;
import java.io.*;

class xTSBenchmark {
    private static final int DISPLAYTIME=2000;
    private static final int AVERAGE=10;
    public static void main (String[] args) throws Exception {
        ParameterManager pm=new ParameterManager(args);
        int st,si,sp;
        tsAbstraction ts=null;
        try {
            st=new Integer(pm.getParameter("st")).intValue();
            si=new Integer(pm.getParameter("si")).intValue();
            sp=new Integer(pm.getParameter("sp")).intValue();
            String space="sync_"+pm.getParameter("space");
            String tsOption=pm.getParameter("ts");
            String tsServer=pm.getParameter("tp");
            if (tsOption.equals("t")) ts=new TSpacesAbstraction(space,tsServer);
            if (tsOption.equals("s")) ts=new sTSpacesAbstraction(space,new sServer(tsServer));
            if (tsOption.equals("d")) ts=new dTSpacesAbstraction(space,new sServer(tsServer));
            if (ts==null) throw new Exception();
        }
        catch (Exception e) {
            System.out.println("Options:
- ts: (t|s|d) for test on TSpaces, xTSpaces and dTSpaces"
- st: (#synchronization tuples)
- si: (#synchronizations per initialization)
- sp: (#synchronization pairs)"
- tsServer (tuplespace server)"
- space (space name)");
            return;
        }
        SyncThread[] sy=new SyncThread[sp*2];
        for (int i=0;i<sp;i++) {
            sy[2*i]=new SyncThread(ts,"A"+i,"B"+i,si,st);
            sy[2*i+1]=new SyncThread(ts,"B"+i,"A"+i,si,st);
```
sy[2*i].start();
sy[2*i+1].start();
}
int[] averageVals=new int[AVERAGE];
int average=0;
int averageValPointer=0;
for (int i=0;i<AVERAGE;i++) averageVals[i]=0;
while (true) {
  Thread.sleep(DISPLAYTIME);
  int perf=0;
  int blocked=0;
  int errs=0;
  for (int i=0;i<sp*2;i++) {
    if (sy[i].syncs==0) blocked++;
    perf+=sy[i].syncs;
    sy[i].syncs=0;
    errs+=sy[i].errs;
    sy[i].errs=0;
  }
  average=average-averageVals[averageValPointer]+perf;
  averageVals[averageValPointer]=perf;
  averageValPointer=(averageValPointer+1)%AVERAGE;
  System.out.println(""+perf+" - average: "+(average/AVERAGE)+" -
                             errors: "+errs+" - blocked: "+blocked);
}
}

class SyncThread extends Thread {
  public int syncs,errs;
  tsAbstraction ts;
  String myName,hisName;
  int si,st;
  SyncThread (tsAbstraction ts, String myName, String hisName, int si, int st) {
    this.ts=(tsAbstraction)ts.clone();
    this.myName=myName;
    this.hisName=hisName;
    this.si=si;
    this.st=st;
    syncs=0;
    errs=0;
  }
  public void run () {
    while (true) {
      ts.reInit();
      for (int i=0;i<si;i++) {
        for (int j=0;j<st;j++) {
          try {ts.write(myName+_+j);} 
          catch (Exception e) {errs++;} 
        }
        for (int j=0;j<st;j++) {
          try {ts.take(hisName+_+j);} 
          catch (Exception e) {errs++;} 
        }
        syncs++;
      }
    }
  }
}

interface tsAbstraction {
  Object clone ();
  void reInit ();
  void write (String tuple) throws Exception;
}
void take (String template) throws Exception;
}

class TSpacesAbstraction implements tsAbstraction {
    TupleSpace ts;
    String space, serverAddress;
    TSpacesAbstraction (String space, String serverAddress) {
        this.space=space;
        this.serverAddress=serverAddress;
        ts=null;
    }
    public Object clone () {
        return new TSpacesAbstraction (space,serverAddress);
    }
    public void reInit () {
        try {
            ts=new TupleSpace(space,serverAddress);
        } catch (Exception e) {System.out.println("reInit error");}
    }
    public void write (String tuple) throws Exception {
        ts.write(new Tuple(tuple));
    }
    public void take (String template) throws Exception {
        Tuple a=ts.take(new Tuple(template));
        while (a==null) {
            a=ts.take(new Tuple(template));
            Thread.yield();
        }
    }
}

class sTSpacesAbstraction implements tsAbstraction {
    String space;
    sServer serverAddress;
    sAPI ts;
    sTSpacesAbstraction (String space, sServer serverAddress) {
        this.space=space;
        this.serverAddress=serverAddress;
        ts=null;
    }
    public Object clone () {
        return new sTSpacesAbstraction (space,serverAddress);
    }
    public void reInit () {
        try {
            ts=new sTSpaces(serverAddress);
        } catch (Exception e) {System.out.println("reInit error");}
    }
    public void write (String tuple) throws Exception {
        ts.write(space,new xTuple(new xField(tuple)));
    }
    public void take (String template) throws Exception {
        xTuple a=ts.take(space,new xTuple(new xField(template)));
        while (a==null) {
            a=ts.take(space,new xTuple(new xField(template)));
            Thread.yield();
        }
    }
}

class dTSpacesAbstraction implements tsAbstraction {
    String space;
    sServer serverAddress;
}
dAPI ts;

dTSpacesAbstraction (String space, sServer serverAddress) {
    this.space=space;
    this.serverAddress=serverAddress;
    ts=null;
}

public Object clone () {
    return new dTSpacesAbstraction (space,serverAddress);
}

public void reInit () {
    try {
        ts=new dTSpaces(serverAddress);
    } catch (Exception e) {System.out.println("reInit error");}
}

public void write (String tuple) throws Exception {
    ts.write(space,new xTuple(new xField(tuple)));
}

public void take (String template) throws Exception {
    xTuple a=ts.take(space,new xTuple(new xField(template)));
    while (a==null) {
        a=ts.take(space,new xTuple(new xField(template)));
        Thread.yield();
    }
}

TESTING STRATEGY

Since all the performance test have to be representative, they must rely on similar infrastructures. Therefore, tuplespace servers and the running benchmark programs are placed on workstations with comparable attributes. The limitation of workstations’ computing power shall not influence the test result. Hence, servers and benchmark programs are placed on a separate workstation each. The testing environment is an Ethernet LAN with 10MBit/sec. Our projection concludes that the LAN is not the system’s bottleneck. The following describes the testing system’s configuration that leads to the data obtained in the two figures:

1. Two tests on a sServer with st=15, sp=2, bp=2, si=1000. At the first test, the primary fails at t=0, and at the second test, the secondary registers to the primary at t=0.

2. Seven tests on a dServer (e={1,2,3}) with t={1,...,7}. st=3*t, sp=t, si=1000, bp:=#(Benchmark_programs)=1. The number of synchronizations is multiplied by st, in order to obtain the number of tuples written to the tuplespace per second.

Before each test, all tuples are deleted from the servers.