XMLSpaces.NET: An Extensible Tuplespace as XML Middleware

Robert Tolksdorf, Franziska Liebsch,Duc Minh Nguyen
Freie Universität Berlin, Inst. für Informatik, AG Netzbasierte Informationssysteme
Takustr. 9, D-14195 Berlin, Germany
research@robert-tolksdorf.de,franziska@adestiny.de,nguyen@inf.fu-berlin.de

ABSTRACT

XMLSpaces.NET implements the Linda concept as a middleware for XML documents on the .NET platform. It introduces an extended matching flexibility on nested tuples and richer data types for fields, including objects and XML documents. It is completely XML-based since data, tuples and tuplespaces are seen as trees represented as XML documents. XMLSpaces.NET is extensible in that it supports a hierarchy of matching relations on tuples and an open set of matching amongst data, documents and objects.

1 INTRODUCTION

According to [3], middleware for XML-centric applications can be classified as middleware that supports XML-based applications – for example, a class library providing an XML-parser –, as XML-based middleware for applications – for example, a protocol suite that uses XML-representation for messages –, or as completely XML-based middleware – an example is the XML-based XSL language which transforms XML documents.

XMLSpaces ([10, 11]) extends the Linda coordination language by establishing a distributed shared space in which XML documents are stored. A process, object, component or agent contributing a result to the overall system will emit it as an XML document to the XMLSpace. Here, it is stored until some other active entity retrieves it. For retrieval, a template of a matching XML document is given. The matching relations possible are manifold, currently, XMLQueries, textual similarity of XML documents and structural similarity wrt. a DTD are supported.

XMLSpaces follows the Linda concept of uncoupled coordination. Producers and consumers of information do not have to reside at the same location. Also, they do not need to have overlapping lifetimes in order to communicate and to synchronize. The producer can well terminate after putting a document into the space while the consumer does not even exist. The consumer can try to retrieve a matching document while the producer has not started to exist. This uncoupledness in space and time makes the Linda concept attractive for open distributed systems.

XMLSpaces adds to Linda expressibility by providing a richer type of exchanged information. While Linda deals only with tuples composed of a set of primitive data types, XMLSpaces allows any well-formed XML document in tuple fields. The set of matching relations is not fixed but can be extended. The distribution and replication schema implemented in XMLSpaces is well-encapsulated and extensible.

XMLSpaces was implemented at TU Berlin on top of Java using RMI. For the basic tuplespace functionality, it relied on TSpaces, an IBM implementation of Linda with small extensions. In addition, it implemented a set of matching relations and a set of distribution strategies.

Following the above classification, XMLSpaces is middleware that supports XML-applications. In this paper, we describe an evolution of XMLSpaces, called XMLSpaces.NET which goes even further and tries to be a self contained XML-middleware. The XMLSpaces.NET project implements the XML-paces concept with high quality on the .NET platform. It consists of two parts. First, the implementation of an XMLSpaces kernel in C# that includes the basic coordination mechanisms and the specific XML support. Second, the implementation of a distributed XMLSpaces on top of the .NET framework. In this paper we describe the ideas for a complete XML-representation for both tuples, subtuples and tuplespaces in XMLSpaces.NET, its architecture and current implementation on the .NET platform.

2 TUPLESPACES IN XML

A generic middleware has to offer means to exchange data, documents and objects among distributed applications. See [3] for a review of the historic distinction between object- and document-oriented middleware. XMLSpaces.NET provides an integrated representation of data in standard Linda-tuples, objects from common programming platforms and documents in XML representation. The operations – following the Linda coordination language – implemented in XMLSpaces.NET become more powerful since they can be applied to all three mentioned kinds of data of interest in a uniform manner.

2.1 XML-based Tuplespaces

A standard Linda-tuple is a list of fields. Those fields carry values from or denote some primitive types, usually from that of a host language. For richer structuring of tuples, XMLSpaces.NET extends that basic notion by allowing nested tuples. An XMLSpaces.NET-tuple thus contains a sequence of fields or XMLSpaces.NET-tuples and is actually a tree of a certain "depth" with primitive data or objects as leaves. Such a *tupletree* is sufficient to represent all our tuples, since fields cannot contain references. The common Linda operations supported by XMLSpaces.NET always manipulate a complete tuple at a time, so the structure of an existing tupletree is never changed or manipulated.

As mentioned above, we strive for a middleware that supports data, documents and objects. A standard Linda-tuple can be considered as data with fields being primitives from some simple type-system. Lindas standard matching scheme can be applied for such tuples. For now, we leave the aspect of matching nested tuples open.

To support documents, we allow well-formed XML documents as tuple fields. The aforementioned XMLSpaces already allowed for tuples that contained XML documents and offered a set of matching relations to select tuples containing XML documents as fields, for example by referencing a DTD to which a document in a field had to comply. Furthermore, a tuple can contain an object from some programming language – Java objects or .NET objects are examples. Matching on them is object- resp. class-specific.

Our aim is to design an integrated and self contained XML-middleware. So far, we have talked about tuples, primitive data, XML documents and objects. For XMLSpaces.NET we have to find a uniform notion that integrates these. The natural choice is, of course, to use an XML representation for the tuples. A tuple (and a nested tuple, too) is a tree with fields as leaves or nested tuples as subtrees. It is obvious, that there can be an XML representation for such tuples. XML documents in fields are trees, since they are wellformed. Finally, the objects that we want to support can also be considered as trees, at least there can be some tree - based serialization of them. It is a reasonable assumption that in a modern object system, one can generate an XML-based serial representation which maps an object into an XML-document.

With that XMLSpaces.NET takes the idea of an XML based coordination medium a step further, since any tuple in XMLSpaces.NET is an XML document. We can go on to apply that principle to tuplespaces.

A tuplespace is a collection of tuples. In the case of multiple or nested tuplespaces, it is a collection of tuples and spaces. The tuplespaces are in any case also trees.

For XMLSpaces.NET, we consider a tuplespace as a collection of XML documents as described. This collection can be represented, in turn, as another tree similar to the tupletree described. The tuplespace differs from tuples in that it cannot contain fields as direct descendants of the root node.

So – at least conceptually – XMLSpaces.NET considers the complete coordination medium as a single XML document with the first level being the tuplespace (or one or several levels in the case of multiple or nested spaces) and the further levels being tuples and nested tuples. The leaves of this one XML document are the fields which are primitives, XML documents or XML serializations of objects. This view is one contribution of XMLSpaces.NET

2.2 Matching in XMLSpaces

Fields in Linda tuples are either *formals* – containing only a type as in $\langle$?int$\rangle$ – or *actuals* containing a typed value as in $\langle 2 \rangle$. Tuples that contain formals are considered templates in Linda.

In XMLSpaces.NET an item used with tuplespace operations can be classified as a tuple or a template. A tuple contains only actual fields or tuples as fields, like $\langle 1,2 \rangle$ or $\langle 1,\langle 2,3 \rangle\rangle$. A template can also contain formal fields or templates like $\langle$1,?int$\rangle$ or $\langle$1,$\langle$?int$\rangle\rangle$. The set of tuples is a subset of templates.

We do not introduce the classification as typing in XMLSpaces.NET, since this would require us to consider either tuples as subtypes of templates (they are more special in that they cannot contain formals), or vice versa (templates are more special in that they can contain formals). The *in* and *read* operations expect something that is classified as a template, an out something classified as a tuple. So the item $\langle 1,2 \rangle$ is classified by its *use* in an operation as a tuple or a template.

Matching in XMLSpaces.NET distinguishes actuals and formals as in Linda. Any matching tuple and templates must have the same length, that is the same number of fields and subtuples or subtemplates.

We now distinguish two extreme kinds of matching when considering subtuples. *FlatTemplate*-matching performs matching only on the fields of the first level of the tupletree. The content of fields containing primitive data, XML documents or objects is not even tested for equality or type-equivalence but only considered as being of the metatype "tuplefield". Similar, nested tuples and templates are only considered as being of the metatype "subtuple/subtemplate". It suffices that *some* (sub-)subtuple is present in a field, its structure and content is not considered further. In contrast to that, *DeepTemplate*-matching performs a complete recursive matching of the content of contained subtuples and templates considering type- and value-equivalence.

We write $\langle$1,2$\rangle_D$ for a template that requires deep matching and $\langle$1,2$\rangle_F$ for one with flat matching. A tuple $\langle$1,$\langle$2$\rangle$,3$\rangle$ will be matched by a template $\langle$1,$\langle$2$\rangle_D$,3$\rangle_D$, but not by $\langle$1,$\langle$0.0$\rangle_D$,3$\rangle_D$. Deep matching is intuitively the standard Linda matching recursively applied to nested tuples. Flat matching transforms the typing to a metalevel. A flat template $\langle$1,$\langle$2$\rangle_F$,3$\rangle_F$ matches both $\langle$1,$\langle$2$\rangle$,3$\rangle$ and $\langle$1,$\langle$0.0$\rangle$,4$\rangle$. The template is transformed into $\langle$F,T,F$\rangle$, where F means field and T means tuple. Flat and deep matching can be combined. $\langle$1,$\langle$2$\rangle_F$,3$\rangle_D$ matches $\langle$1,$\langle$2$\rangle$,3$\rangle$ and $\langle$1,$\langle$0.0$\rangle$,3$\rangle$ but not $\langle$1,$\langle$0.0$\rangle$,4$\rangle$.

Finally, flat matching takes precedence over deep matching. In a template $\langle$1,$\langle$2$\rangle_D$,3$\rangle_F$, the second field will be transformed to the metatype T, overriding the deep matching. This means that $\langle$1,$\langle$2$\rangle_F$,$\langle$3$\rangle_D\rangle_F$ is equal to $\langle$1,$\langle$2$\rangle_F$,$\langle$3$\rangle_F\rangle_F$. We therefore make deepmatching the default and require only the notation for flat matching if necessary. So we write $\langle$1,$\langle$2$\rangle_F$,3$\rangle_D$ as $\langle$1,$\langle$2$\rangle_F$,3$\rangle$ and $\langle$1,$\langle$2$\rangle_F$,$\langle$3$\rangle_F\rangle_F$ as $\langle$1,$\langle$2$\rangle$,$\langle$3$\rangle\rangle_F$.

It turns out that there are further interesting relations between flat and deep matching. While flat matching ignores all further characteristics of fields and subtuples, *flat/size* matching requires that subtuples must be of the same size as the one given as template. Size is defined as the sum of the number of fields and subtuples. We write $\langle\ldots\rangle_{FS}$ for a template that requires this matching. The template $\langle$1,$\langle$2$\rangle_{FS}$,3$\rangle_D$ matches $\langle$1,$\langle$0.0$\rangle$,3$\rangle$ but neither $\langle$1,$\langle$2,3$\rangle$,3$\rangle$ nor $\langle$1,$\langle$2,$\langle$3$\rangle\rangle$,3$\rangle$.

The “metatyping” of fields can also be of interest. We introduce *flat/type* matching for that case. Here, subtuples must contain the same number of fields and subtuples. We write $\langle\ldots\rangle_{FT}$ for that kind of matching. The template $\langle$1,$\langle$2$\rangle_F$,3$\rangle_{FT}$ matches $\langle$1,$\langle$2$\rangle$,3$\rangle$ and $\langle\langle$1$\rangle$,2,3$\rangle$ but not $\langle\langle$1$\rangle$,$\langle$2$\rangle$,3$\rangle$. As a further relation of interest, we introduce *flat/value* matching. Here, subtuples are not considered further while fields have to have equal value. We write $\langle\ldots\rangle_{FV}$. The template $\langle$1,$\langle$2$\rangle_F$,3$\rangle_{FV}$ matches $\langle$1,$\langle$0.0$\rangle$,3$\rangle$ but neither $\langle$1,2,3$\rangle$ nor $\langle$0.0,$\langle$2$\rangle$,3$\rangle$.

The relations mentioned are ordered, since $D \Rightarrow FV \Rightarrow FT \Rightarrow FS \Rightarrow F$. Further possible relations are currently under study. The differentiated and extensible view on structural matching of nested tuples is one of the contributions of XMLSpaces.NET.

Further matching is possible which combines the relations above. In the current implementation XMLSpaces.NET also supports a matching based on the FV and FT relations. It checks for value- and type-equivalence for fields on the first level of the tupletree, but only for equal numbers of fields and subtuples in any subtuples.

Three cases of field matching have to be distinguished for which different matching relations are defined:

Primitive data can be matched on type- and value equivalence as in Linda. In addition, we foresee matching relations like comparisons ($\langle\geq 5$,$\leq 3\rangle$).

Objects are matched on type and object equivalence. Object equivalence is defined here by equal representation of a normalized serialization. It is implemented by comparing the respective SOAP serializations of objects.

Type equivalence of objects and its use in matching is an interesting topic and has led to several proposals in tuplespace research ([2, 8, 9] and others). Objects usually are typed and classified. In most object oriented systems, there is a type- and class-hierarchy. With that, two objects can be in several relations – they can be type compatible if their interfaces are in a subtype relation or can be specializations/generalizations if their classes are in a sub-/superclass relation. The hierarchies mentioned form trees. Again, we have a deep and a flat matching. A template can reference a class or a type like $\langle$?AClass$\rangle_F$. For flat matching, an object matching such a field has to be an instance of that class or type like $\langle$aObject$\rangle$. Deep matching here means that matching objects are instances of direct or indirect subclasses or subtypes like $\langle$bObject$\rangle$ if BClass is a subclass of AClass or the interfaces of the objects are in a subtype relation.

XML documents are matched according to some further matching relation since we lack a definition of normalized equivalence of XML documents.

The flexible and extensible matching of values is another contribution of XML-Spaces.NET.

3 ENGINEERING XMLSPACES

In this section we give an overview of the internal structure and architecture of XML-Spaces.NET.

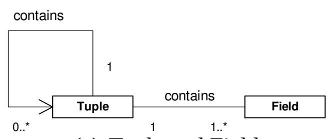
3.1 Local operations

Constructing tuples, nested or not, should be as easy as possible. As aforementioned, nested tuples have a tree-structure, therefore it is easy to build a complex nested tuple from the subtuples (subtrees). As Fig. 1(a) shows, two classes with appropriate methods and constructors are sufficient to describe nested tuples.

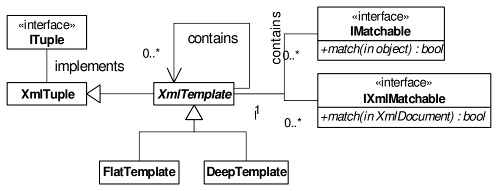
While nested tuples provide structure to what is put into a tuplespace, fields contain the specific data. A field should be capable of storing any type that is valid in a host programming language that uses XMLSpaces.NET. In addition XMLSpaces.NET adds XML-documents as a valid type.

After creating tuples and writing them to a tuplespace with an out, it is necessary to retrieve them. Linda specifies two retrieval operations, a consuming (*in*) and a non-consuming (*read*). To retrieve a tuple from a tuplespace, a template is defined against which a tuple has to match. If the template contains only values it acutally is a tuple. As we have stated in Section 2, one can see Template as a subclass of Tuple and vice versa. For an implementation, however, it is necessary to decide which approach to take. We therefore define Template as a subclass of Tuple, because apart from (actual) fields and tuples, a template can contain templates and formal fields.

At least three groups of types can be stored in a field: primitive types, objects and XML-documents (see Sec-



(a) Tuple and Field



(b) Template

Figure 1: Tuples and Templates

tion 2.1). In our implementation we can join two groups, primitive types and objects, since they are part of the host programming language C#.

The definined matching-relations on the two remaining groups (types of the host language and XML Documents) are totally different. Types of the host programming language can be checked for their specific type and value, using the programming language operations. The document type of a wellformed XML-document is determined by its structure and its value by the values of the tags, attributes and contained text. An XML-document itself could have a structure and contents that is itself as complex as a complete tuplespace. Matching relations can be defined on different levels of granulation, i.e. an XML-document's structure or even values of a single element or attribute. The most obvious way to define matching relations is by using XPath-expressions. Although XPath already offers a wide variety of matching-relations, many more matching-relations can be imagined, e.g. validation against XML-schema or XQuery.

To keep the creation and maintenance of matching-relations flexible, we have define two interfaces, which stand for one type of matching-relation each. This approach allows the collection of matching-relations, which is released, to be easily extended with user defined ones.

With nested tuples, there are at least two different ways of matching (see Section 2.2). XMLTemplate is defined as an abstract class, that contains rules for combination of Templates, Tuples, Fields and matching-relations. Any subclass of XMLTemplate can be used interchangeably. By defining a class that extends XMLTemplate it is possible to extend the set of templates. As we have observed in Sec. 2.2, there are many interesting templates for nested tuples that should be realizable via an easy extension-mechanism. The matching-algorithm should be able to decide which template to use at runtime, so new templates are just defined and used in matching without having to change existing code.

3.2 Remote Operation

Any active entity that emits tuples to or retrieves tuples from a TupleSpace is considered to be a client. In order to create and work on a tuplespace, a client needs a TupleSpace object. TupleSpace objects serve as references to tuplespaces on a server. Clients may have many TupleSpace objects, of course. Apart from the traditional Linda-operations (*in*, *out*, *read*, *eval* a TupleSpace object contains methods to log on or create tuplespaces and manipulate attributes that affect its behavior. Examples of such planned attributes are timeouts, lease-time of objects etc.

The server manages the tuplespaces and the distribution strategies. It has a collection of TupleBuckets, which represent tuplespaces. Any TupleSpace object that a client uses is associated exactly to one TupleBucket. However, many TupleSpace objects may be associated to the same bucket and thereby share the same tuplespace.

We plan to support three types of replication (none, full and partial replication) as described in [10, 11].

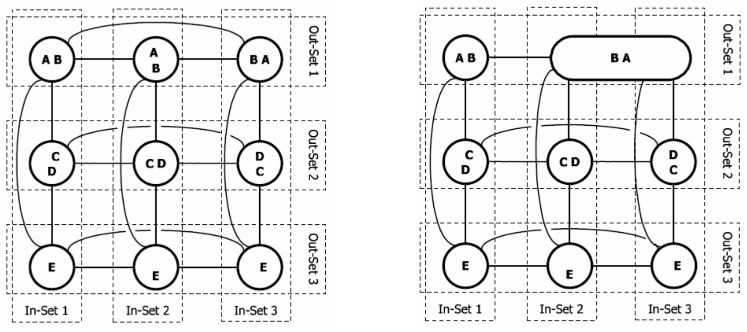
In a system where the tuples are not replicated, all servers manage their own tuplespaces only. If a client writes a tuple to a tuplespace that is on the local server, we have the non-distributed case and simply write (*out*) the tuple to the tuplespace. If the target tuplespace is on a remote server, a *Distributor* object forwards the tuple to the server, which manages that tuplespace. An *in* or *read* is executed on the local server first and then performed on remote servers, if the tuple or tuplespace can not be found. This strategy is easy to implement and consumes little resources compared to strategies with replication.

The counterpart to that strategy is the full replication strategy. Every tuple is stored locally and on every remote server. This brings about a lot of communication and organization overhead among the servers, as with every operation all servers have to be notified and their tuplespaces must be changed according to the source server. This strategy offers a high failsafety. The disadvantages, however, are potentially heavy network traffic and a high consumption of resources.

Between these two extremes is the partial replication strategy in order to gain the advantages of both. In a system performing partial replication all servers are regarded as nodes in a rectangular grid. The grid is partitioned into horizontal and vertical stripes, assigning each node to exactly one intersection of stripes. Each horizontal stripe is defined as an *in-set* and each vertical stripe is defined as an *out-set*. Tuplespaces of nodes in *in-sets* must be disjunct, whereas tuplespaces of nodes in *out-sets* exact copies.

This limits all operations to only a subset of servers. All *in* operations are performed on one *in subset* of servers. The advantage for *out* operations is that they are performed on one out-set only. If a tuple is consumed or added, only the nodes in that *out-set* need to be updated.

However, the number of participating servers should be dynamic. This does not affect the non-replication and the full replication strategy, but for the partial replication strategy it is impossible to guarantee a rectangular grid of nodes. To solve this problem simulated nodes were



(a) The Grid of Nodes (b) Example of a simulated node

Figure 2: Intermediate Replication of Tuplespaces

introduced. Whenever the number of nodes is not sufficient to form a rectangular grid, i.e. when new servers want to participate in or leave the distributed tuplespace, the neighbour in the *in-set* of such a “whole” in the grid simulates its presence. As they are members of the same *in-set* they have exactly the same contents.

In addition to these issues, a distributed system performing any kind of replication must guarantee the integrity of its data. Therefore all distributed operations must follow a communication and operation protocol, which allows locking and releasing of tuples and thereby guarantee data integrity.

4 IMPLEMENTATION

We use Microsoft’s .NET Framework to implement XMLSpaces.NET. It already features functionality we need to implement the Linda-System and the extensions. Languages like VB.NET, C++.NET, Python.NET were extended to work with the .NET Framework. We choose C# as the host language, as it is specially developed for the .NET Framework. All languages, however, compile to the Microsoft Intermediate Language (MSIL) and there should be no significant difference in terms of performance.

After XMLSpaces.NET is released, clients can be written in any host language of the .NET Framework, as they are capable of accessing the same assemblies.

4.1 Tuples

Tuples use the built-in .NET type *System.Xml.XmlDocument* to represent their contents. *System.Xml.XmlDocument* is an implementation of the W3C’s DOM and DOM2. It is therefore an in-memory representation of an XML-Document with methods for manipulation. In order to store data into XML, we need a serialization pattern. Pattern in this context means the XML-structure that represents the types. The .NET Framework has a uniform type-system for all host languages, called Common Type System (CTS). Types are named *System.**, where * is any of the types defined in .NET. Depending on the host language, the available types may vary. For example, C# does not support pointers so the Pointer- Types are not available in C# but they exist in C++.NET. XMLSpaces.NET is capable of handling all possible types, as the type-information is extracted during runtime and stored in the XML-document. On the other hand only clients that know of those specific types (written in a host language in which those types are available) will need to retrieve tuples with such fields.

For these primitive types a serialization is found easily, as we only need a string that represents the value. However, a string representing the value is ambiguous, since ”1” might be *System.Int16*, *System.Int32*, *System.Int64*, *System.Char* or a *System.String*. We therefore need to store the value’s type in order to deserialize it correctly. The serialization for primitive datatypes is therefore: *<Field type=”System.*”>VALUESTRING</Field>*.

Objects, in this context are instances of classes, arrays or structs (container for structured data in C#). They are serialized differently, of course. We could use *Reflection* to do the serialization to XML manually, but the .NET Framework already features functionality that serializes an object into a SOAP-document ([12]). Any other XML serialization of objects can be used instead, of course. The serialization for primitive datatypes is therefore: *<Field type=”Soap”>SOAPDOCUMENT</Field>*. It is possible to serialize primitive datatypes into SOAP-documents as well, but we have chosen to serialize into the indtroduced form, because the resulting SOAP-document would be much larger and thus takes more time for matching operations and occupies more memory.

XML-Documents do not need to be serialized, as they can already be represented as strings. The third serialization pattern is *<Field type=”XmlDocument”>XMLDOCUMENT</Field>*.

The following is a simple example of a tuple containing all three types:

```
<Tuple tuplecount="0" fieldcount="3">
 <!-- primitive datatype -->
 <Field type="System.String">Hello</Field>
 <!-- serialized dateTime - object -->
 <Field type="Soap">
  <SOAP-ENV:Envelope  ... omitted ...
   <SOAP-ENV:Body>
    <xsd:dateTime id="ref-1">
      <ticks>630720000000000</ticks>
    </xsd:dateTime>
   </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
 </Field>
 <!-- XML-document -->
 <Field type="XmlDocument">
  <Hello>World!</Hello>
 </Field>
</Tuple>
```

The serialized data is stored in Fields, which represent single units of data inside a tuple.

4.2 Templates

As we have stated in Section 3.1, we choose Template to extend Tuple with functionality for matching. It is obvious that we only need to make small modifications. Apart from Tuples a Template may contain other Templates and a field can be substituted by a matching-relation. We implement two interfaces, which form the basis for the extensibility of XMLSpaces.NET. Their serialization is as follows: *<Field type="IMatchable">SOAPDOCUMENT</Field>* and *<Field type="IXMLMatchable">SOAPDOCUMENT </Field>*.

We can determine if an object is an instance of a class that implements one of those interfaces. Thereby we differentiate two more types that are serialized in Templates, *IMatchable* and *IXMLMatchable*. The SOAP-Formatter of the .NET Framework serializes those objects, which produces well-formed XML-documents.

Only in a template, instances of classes with these interfaces have to be handled separately, as they are needed to perform the matching. In a Tuple templates and those objects would be treated like any other object, allowing even instances of matching-relations and templates to be stored in the tuplespace and be exchanged among clients.

So far only matching-relations where investigated. However, we need an extensibility-mechanism for templates, too. It is necessary to store the type of the template in the XML-representation. Any object in C# has a fully qualified name as its type description, e.g. *XMLSpaces.Templates.DeepTemplate*. We extend the XML-representation of a tuple to contain XML-elements, where the type attribute stores the fully qualified name of the template. On one hand the resulting XML-document contains all information that is needed for matching and keeps the core implementation independent from any extensions. On the other hand, there is no limitation to the number of templates. In the current implementation only matches on the XML-structure are allowed. A later implementation might provide adequate iterators on the XML-structure, allowing implementation of templates, that use the iterators instead of the XML-structure to match tuples.

4.3 Extending Matching Relations

In C# any class or primitive data type is a sub-class of *object*. We therefore define an interface *IMatchable* with a single method *bool matches(object o)*. Any matching operation on objects, i.e. primitive data types and instances of objects, can be defined using this interface. This concept is much more powerful than the Linda matching, which is either a type-match, or an exact match of value. Our approach allows the definition of finer relations. A string for example, can be matched in many different ways. A few examples are to match the string exactly, by ignoring the case of the letters, by matching on a substring or its conformity to a regular expression. Depending on the use of XMLSpaces.NET, different matching-relations may be preferred.

XML-documents can be matched in a wide variety of ways. There are existing standards such as XPath, XPointer, XSLT and drafts for future standards like XPath2 and XQuery. It is essential that the set of matching relations for XML-documents is at least as extensible as the set for objects and primitive types. We define the interface *IXMLMatchable* for that purpose. It contains a single method *bool matches(XmlDocument doc)*. Any matching-relation for XML Documents that is not part of the basic set released with XMLSpaces.NET can be defined by implementing this interface. If future development of the .NET Framework integrates, for example, XQuery (which it currently does not), or an API to an existing XQuery system is available, it will be easy to extend the matching-relations of the basic system with that matching relation.

4.4 Matching

A tuplespace consists of a collection of tuples. Following our concept, a tuplespace is a special form of a nested tuple. It contains only tuples and no fields on the first level. Again, we can represent the whole tuplespace as an XML-document. From a higher level a tuplespace can be considered as a tuple of an other tuplespace. This makes it possible to store whole tuplespaces in another and retrieve it at a later time as if it were a tuple.

Matching in XMLSpaces.NET (as in Linda) occurs only on *in* and *read* operations. All arguments passed to them are regarded as templates. Even if a tuple is passed to these methods, a DeepTemplate is wrapped around it to perform an *actual* match. As a tuplespace is an XML-document, we can use XPath, which is implemented in the .NET Framework, to perform a preselection (number of fields and subtuples) of potentially matching tuples. The server then checks if a tuple of that preselected set matches on a given template. The sequence of actions is shown in Fig. 3.

A client requests a tuple by calling *in* or *read* on the TupleSpace object. The call is delegated to the server, which does the preselection on the TupleBucket and performs the match on the collection of potential matches. The first matching tuple is returned to the TupleSpace object and deleted from the TupleBucket. The other tu-

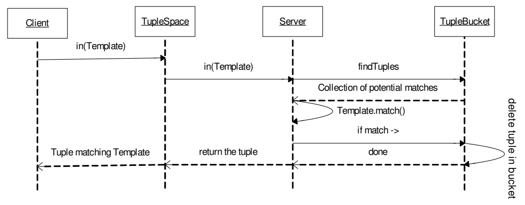


Figure 3: Matching

ples are left untouched. The TupleSpace returns either the retrieved tuple to the client, or a null-reference.

The template determines to which depth (DeepTemplate, FlatTemplate, etc.) a tuple is checked and how exact the Fields of the tuple are examined. As stated in Sec. 2.2 there are many interesting types of templates that match a tuple on a very high level (FlatTemplate), where only the metatypes of fields and subtuples are checked, or on a very low level (DeepTemplate), where a template has to match exactly on the tuple. The matching-algorithm traverses the DOM-tree of the XML-document and compares the nodes. Depending on the template the fields and depth are checked differently, so the algorithm has to determine whether there are any nested templates and switch to the algorithm of the nested template.

Whenever an IMatchable or IXMLMatchable object is found in a template, it is deserialized and the *matches()* method is called with the required parameter, i.e. *System.object* for *IMatchable* and *System.Xml.XmlDocument* for *IXMLMatchable*. If any field does not match or any IMatchable or IXMLMatchable object returns false, the algorithm terminates.

Every match operation performs following actions: a) preselect a set of matching tuples on the bucket based on their number of fields and subtuples, b) perform the match method of the template on each tuple in the set of potential matches.

Using the number of fields and tuples we can also decide early whether to continue matching on deeper levels of an XML-document or not. This information limits the matching times on nested tuples as the number of fields and tuples can be checked on any subtupletree.

4.5 Distribution

The .NET's *Remoting Framework* is used to implement the client-server architecture. Using a directory service, such as Microsoft's *Active Directory* [5] or *OpenLDAP* [7], allows a dynamic configuration of the participating servers and the replication mode, i.e. switching the replication mode of all servers during runtime.

However, on a campus network *Active Directory* is not always flexible enough, as the *schema* of the directory has to be modified to meet the needs of XMLSpaces.NET. The schema change might require administrative rights not available to an end-user. OpenLDAP is an alternative in this case. We decided to stay as independent as possible of those technical problems and have implemented an extra class to maintain the server list.

The distributed system differs from the non-distributed one in the use of the buckets. While in a non-distributed system the server directly calls methods on its local buckets, a distributor object manages the calls to the local buckets and the remote buckets. The implementation of the distributed system profits from the XML structure of the TupleBucket. If each tuple is assigned a unique identifier pointing to its source location, a TupleBucket is able to group those tuples in an XML subtree associated to that remote source. This is beneficial for the implementation of the replication as it is easy to sort out tuples of different servers, since all tuples with the same source server are under the same subtree. For an *out* operation the Distributor inspects all servers in the server list for their replication mode, adds the identifier to the tuple and sends it to appropiate target servers, depending on the replication strategy, where they are stored to a TupleBucket's subtree according to its identifier.

For an *in* operation the identifier is ignored and the search includes all tuples in the tuplespace. The removal of a match is easy, as the tuple's identifier points to the correct subtree in each TupleBucket, in which the tuple can be found and therefore speeds up the operation. The XPath API allows a fast search on the XML structure of the TupleBucket using the tuple's identifier.

In case the replication mode changes, or a server deregisters from the server list, the whole contents of the server's tuplespace can be easily removed by deleting the subtree representing that server's replicated tuplespace. If the replication starts up, the contents of a TupleBucket can be added as a subtree to a remote server's TupleBucket.

In order to lock a tuple we simply add a boolean attribute "locked" to the tuple's XML root element. If an operation is being performed on the tuple the attribute has to be set to "true" and else "false". The .NET Framework's native support for XPath queries and the DOM2 make this approach easy.

5 PERFORMANCE

We ran several performance tests on our system, a 2.40 GHz Pentium 4 with 512MB RAM running Microsoft Windows XP Pro and the Microsoft .NET Framework 1.1. As there are many dependencies in the XMLSpaces.NET system, we decided to explore the performance along the following dimensions: 1) type of tuple, i.e. tuples containing primitive data types, objects, or XML documents 2) number of tuples in the tuplebucket 3) number of potentially matching tuples in the bucket, i.e. tuples that have equal tuplecount and fieldcount as the template or tuple we want to match against

For the implementation of the performance test we designed some reference tuples, which contained 5 fields with primitive data or 5 fields with an object each or 5 fields with an XML document each.

The tests were ran on buckets of size 500, 1000 and 2000. At the beginning of each test the corresponding number of tuples is randomly generated to fill the bucket. The randomly generated tuples built from template fields to make sure they have a determinable form. The tuples only vary in the number of fields and their depth. At this point we assumed two different probabilites on matching tuples: In one experiment, we assumed that 25% of the tuples in the bucket are potential matches, in the other, we assumed 50% of potential matches.

No templates were used to retrieve tuples, as we intended to measure the time taken for an exact match of tuples. Owing to the recursive matching algorithm any match against a template (using matching relations) is usually faster since a template match only compares a fragment of information an exact match does.

Using this testbed we had the system play "ping pong" for each of the above defined type of tuples and got the results shown in Figure 4. Two clients play "ping pong" when each has a tuple the other is waiting for, i.e. one client writes its tuple to the tuplespace and waits for the tuple of the other client. The other client starts by waiting for the tuple and writes its own tuple only after having received the other client's tuple etc.

The observations can be explained easily. A match took longer the more potential matches were in the bucket, as the algorithm tries to match against any of the potentially matching tuples. In the worst case it is the last tuple (or none) that matches the template-tuple.

Apart from the number of potential matches the time elapsed for a match depends upon its type. As explained earlier the serialization pattern for primitive types is relatively compact and, except the type "string", cannot be very long. It is therefore easy to see that this type of matching is the fastest. As objects are serialized to SOAP-format XML documents they should be matched in approximately the same time as equally large XML documents. However, all objects that are represented in the SOAP - format have a large root element in common, which identifies the SOAP version and Common Language Runtime (CLR) the system is running on. If many potentially matching tuples with objects are in the bucket the overhead for comparing that root element is relatively high. One possible optimization is to skip the header and compare only the body of the SOAP envelope. The consequenc is, however, that objects of systems running different CLR are identified as the same object, even though they represent different ones.

The XML representation gives us some advantages. Using the attributes *tuplecount* and *fieldcount* we can make a preselection with XPath. As in our two test scenarios there are 50% or 25% of potentially matching tuples in the tuple bucket, the preselection speeds the matching algorithm up by the maximum factor of two or four.

Currently the matching algorithm is very simple and compares each node in the XML representation of the tuple to the template or the template-tuple. Further performance improvement might be achieved if the matching algorithm was to apply the preselection to each subnode. Additionally one can think of an extended preselection that uses the value of the current node. The result could be an even smaller range of potentially matching tuples and a faster matching algorithm.

Of course the performance improvement that is possible depends heavily on the implementation of the XPath API. With an efficient implementation, though, one can still expect further improvements.

6 RELATED WORK

There are several projects documented on extending Linda-like systems with XML documents. However, XMLSpaces seems to be unique in its support for multiple matching relations and its extensibility.

MARS-X [1] is an implementation of an extended JavaSpaces [4] interface. Tuples are represented as Java-objects where instance variables correspond to tuple fields. Such an tuple-object can be externally represented as an element within an XML document. Its representation has to validate towards a tuple-specific DTD. MARS-X closely relates tuples and Java objects and does not look at arbitrary relations amongst XML documents.

XSet [14] is an XML database which also incorporates a special matching relation amongst XML documents. Here, queries are XML documents themselves and match any other XML document whose tag structure is a strict superset of that of the query. It should be simple to extend XMLSpaces with this engine.

[6] describes a preversion for an "XML-Spaces". However, it provides merely an XML based encoding of tuples and Linda-operations with no significant extension. Apparently, the proposed project was never finished.

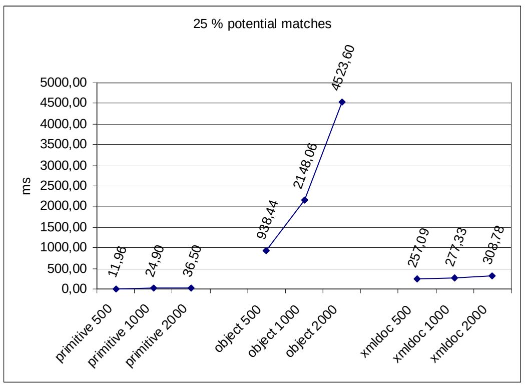
TSpaces has some XML support built in [13]. Here, tuple fields can contain XML documents which are DOM-objects generated from strings. The *scan*-operation provided by TSpaces can take an XQL query and returns all tuples that contain a field with an XML document in which one or more nodes match the XQL query. This ignores the field structure and does not follow the original Linda definition of the matching relation. Also, there is no flexibility for further relations on XML documents.

7 SUMMARY AND OUTLOOK

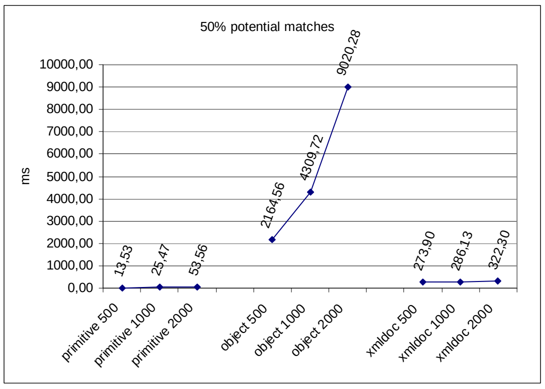
With the XMLSpaces.NET conception we have developed a very extensible XML-based middleware. The further work is on finalizing the set of supported matching relations. The challenge here is to find a set of practically useful relations amongst the wide variety of possible combinations. Also, comparisons like $\langle \geq 5, \leq 3\rangle$ have to be carefully limited not to deadlock the selection of matches.

As mentioned in the beginning, the XMLSpaces.NET project consists of two parts. The XMLSpaces.NET kernel in C# and the distribution of the kernel itself by applying mechanisms like replication etc. Part of the research on distribution will be to explore possibilities to support detachment of parts of a tuplespace for transportation and manipulation by mobile devices.

Furthermore, we will explore to what extend we can easily incorporated further functionalities like secure spaces



(a) 25% potential matches



(b) 50% potential matches

Figure 4: Performance

by the adoption of the respective XML technologies. We hope that such extensions are quite seamless.

In conclusion, XMLSpaces.NET is a flexible XML-based middleware founded on the tuplespace principles. The main contributions are the integrated view on data, documents and objects, the support for structural matching, the extensibility and flexibility of match mechanisms and consequent usage of XML technologies.

Acknowledgment XMLSpaces.NET is funded under contract 2003-144 by Microsoft Research Cambridge.

REFERENCES

[1] G. Cabri, L. Leonardi, and F. Zambonelli. XML Dataspaces for Mobile Agent Coordination. In *15th ACM Symposium on Applied Computing*, pages 181–188. ACM Press, 2000.

[2] C. J. Callsen, I. Cheng, and P. L. Hagen. The auc c++ linda system. In G. Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 39–73. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[3] P. Ciancarini, R. Tolksdorf, and F. Zambonelli. Coordination Middleware for XML-centric Applications. *Knowledge Engineering Review*, 2002. to appear.

[4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

[5] Mircosoft. Active Directory, 2004. *http://www.microsoft.com/windows2000/technologies/directory/ad/defau% lt.asp*.

[6] D. Moffat. XML-Tuples and XML-Spaces, V0.7. http://uncled.oit.unc.edu/XML/XMLSpaces.html, last seen May 6, 2002, Mar 1999.

[7] OpenLDAP Community. Openldap. Website, 2004. http://www.openldap.org.

[8] A. Polze. The object space approach: decoupled communication in c++. In *Proceedings of TOOLS USA'93*, pages 195–204, 1993.

[9] R. Tolksdorf. Laura: A coordination language for open distributed systems. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems ICDCS 93*, pages 39–46, 1993.

[10] R. Tolksdorf and D. Glaubitz. Coordinating Web-based Systems with Documents in XMLSpaces. In *Proceedings of the Sixth IFCIS International Conference on Cooperative Information Systems (CoopIS 2001)*, number LNCS 2172, pages 356–370. Springer Verlag, 2001.

[11] R. Tolksdorf and D. Glaubitz. XMLSpaces for Coordination in Web-based Systems. In *Proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises WET ICE 2001*. IEEE Computer Society, Press, 2001.

[12] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. W3C note for public discussion, 2000. http://www.w3.org/TR/SOAP/.

[13] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[14] B. Y. Zhao and A. Joseph. The XSet XML Search Engine and XBench XML Query Benchmark. Technical Report UCB/CSD-00-1112, Computer Science Division (EECS), University of California, Berkeley, 2000. September.