Fault-Tolerance in Distributed Tuplespaces

Alessandro Fabricio Garcia

Pontifical Catholic University of Rio de Janeiro - Computer Science Department

e-mail: {afgarcia}@inf.puc-rio.br

PUC-RioInf.MCC08/02 June, 2002

Abstract: The tuplespace data model is widely recognized for serving as a foundation for exchanging data and/or coordinating events in distributed systems. In fact, in the last of couple of years the tuplespace paradigm has experienced a renaissance because it is suitable for distributed Internet applications. However, this model is originally based on a centralized scheme, being exposed to classical failures of centralized systems such as single-point failures and bottleneck situations, all which conflicts some of the basic characteristics of distributed systems: fault-tolerance and scalability. So there is a need for providing the tuplespace data model with fault-tolerance and scalability using the replication technique, i.e. distribute replicas of tuplespaces over different tuplespace servers. In this context, the goals of this work are: (i) presenting the tuplespace data model, (ii) identifying the problems in replicating tuplespaces, (iii) surveying the techniques for dealing with such replication problems, and (iv) presenting a model for fault-tolerant tuplespaces called Enterprise TSpaces (ETS). ETS is a further development of the stand-alone version of TSpaces that provides TSpaces with fault-tolerance in loosely coupled systems. The level of fault-tolerance of a ETS tuplespace can dynamically and independently be adjusted by modifying the number of replicas.

Keywords: Tuplespaces, fault-tolerance, distributed systems, data replication, dynamic replication.

Resumo: O modelo de dados baseado em espaço de tuplas é amplamente reconhecido como um modelo uniforme para troca de dados e/ou coordenação de eventos em sistemas distribuídos. De fato, nos últimos anos o paradigma de espaços de tuplas tem renascido devido a sua adequabilidade para o desenvolvimento de aplicações distribuídas baseadas na Internet. Entretanto, tal modelo é originalmente baseado em um esquema centralizado, sendo exposto a defeitos clássicos de sistemas centralizados, o que conflita com características básicas de sistemas distribuídos, tais como tolerância a falhas e escalabilidade. Logo, evidencia-se a necessidade de enriquecer o modelo centralizado de espaços de tuplas com capacidades de tolerância a falhas e escalabilidade através do uso da técnica de replicação, ou seja, distribuir réplicas dos espaços de tuplas em diferentes servidores. Neste contexto, os objetivos deste trabalho são: (i) apresentar o modelo básico de espaços de tuplas, (ii) identificar os problemas relacionados a replicação destes espaços, (iii) apresentar as técnicas existentes para lidar com problemas de replicação de dados, e (iv) apresentar um modelo escalável e tolerante a falhas para espaços de tuplas, chamado Enterprise TSpaces (ETS). ETS é uma nova versão para TSpaces, uma arquitetura que implementa o modelo centralizado de espaços de tuplas. O nível de tolerância a falhas pode ser ajustado dinamicamente e independentemente através da modificação do número de réplicas.

Palavras-chave: Espaços de tuplas, replicação de dados , tolerância a falhas, sistemas distribuídos, replicação dinâmica.

1 Introduction

The tuplespace data model [6] is widely recognized for serving as a foundation providing a uniform model for exchanging data and/or coordinate events in distributed systems. This model is based on the tuple and tuplespace concepts. A tuplespace is a shared, associatively addressed memory space that is organized as a bag of tuples. The basic element of a tuplespace system is a tuple, which encapsulates pieces of data as a vector of fields. Applications access data, communicate and coordinate with each other, using a small set of simple operations to write and read tuples. Tuplespaces effectively decouple communication between processes, both with respect to time (communication is asynchronous) and location (communicating processes do not need to be aware of each other's identity or location in a distributed system). This inherent simplicity and the decoupling of processes offer a number of benefits over systems based on message-passing, remote method invocation, and so on [15]. In fact, in the last of couple of years the tuplespace paradigm has experienced a renaissance because it is suitable for distributed Internet applications.

The tuplespace paradigm is based on a centralized scheme, and as such several centralized tuplespace implementations exist, for instance the original Linda implementation [6], JavaSpaces [14] and TSpaces [15]. Typically, a tuplespace is located on a central computer in the network, meaning that it is an isolated entity. As a centralized system, a tuplespace is exposed to classical failures of centralized systems such as single-point failures and bottleneck situations, all which conflicts some of the basic characteristics of distributed systems: fault-tolerance and scalability. A consequence of having a centralized tuplespace is that tuples are only stored in one location. In case the computer on which the centralized tuplespace is located becomes unavailable due to some failure, all the tuples become unavailable to the processes using them. Therefore, the centralized scheme suffers from not being very fault-tolerant. So there is a need for providing fault-tolerant distributed tuplespaces using the replication technique. If the original tuplespace server crashes, it would be able to recover by communicating with additional servers that contain tuplespace replicas.

In addition, using centralized tuplespaces on a loosely coupled network, such as the Internet, imposes difficulties in terms of estimating potential workload. Therefore, it is difficult to estimate what level of fault-tolerance the distributed tuplespace should have. The problem is that there has been no way to dynamically adjust the level of fault-tolerance and scalability of the distributed tuplespace in previous systems. Once started the level of fault-tolerance and scalability have been fixed in terms of the number of tuplespace replicas. However, the introduction of dynamic replication to tuplespace architecture is not trivial, since is implies in: (i) defining a new semantics for distributed tuplespaces since the notion of concurrency and atomicity is different from centralized tuplespaces, (ii) introducing tuplespace replication without severely affecting performance, (iii) identifying suitable techniques and protocols for keeping the consistency of the different tuplespace replicas, and (iv) supporting nicely the dynamic configuration of the number of tuplespace replicas.

In this context, the goals of this work are: (i) presenting the tuplespace data model, (ii) identifying the problems in replicating tuplespaces, (iii) surveying the techniques for dealing with such replication problems, (iv) presenting a model for fault-tolerant tuplespaces called Enterprise TSpaces (ETS). ETS is a further development of the stand-alone version of TSpaces that provides TSpaces with fault-tolerance in loosely coupled systems. The level of fault-tolerance of a ETS tuplespace can dynamically and independently be adjusted by modifying the number of replicas. ETS defines a tuplespace semantics specifically tailored for large-scale, loosely coupled networks. The remaining of this text is organized as follows. Section 2 introduces the tuplespace data model and Section 3 presents the additional features of TSpaces. Section 4 discusses replication issues in the context of tuplespaces and Section 5 presents the ETS model. Finally, Section 6 presents related work and Section 7 shows the conclusions and ongoing work.

2 The Tuplespace Data Model

This section provides an overview of the tuplespace data model, discussing its basic concepts and its centralized semantics.

2.1 Basic Concepts

Tuplespaces and Tuples. The tuplespace data model origins from the Linda project at Yale University [6]. A tuplespace is a shared, associatively addressed memory space that a number of processes[1] are able to share. Being a global memory, the tuplespace paradigm is often characterized as a distributed shared memory (DSM) abstraction. A tuplespace is a organized as a bag of tuples. The basic element of a tuplespace system is a tuple, which is a vector consisting of one of more typed *fields* each containing some *value*. The basic operations defined on a tuplespace insert tuples into the tuplespace and withdraw tuples from the tuplespace. An example of a tuple is:

```
(253, "ALESSANDRO", true)
```

where this tuple has three fields: the integer number `253`, a string `ALESSANDRO` and a boolean with the value `true`.

Associative Matching and Templates. Somes operations uses a template to lookup tuples within the tuplespace. Templates are used to associatively address tuples via matching. A template is similar to a tuple, but some (zero or more) fields in the vector may be replaced by typed placeholders (with no value) called formal fields (or wildcards). A formal field in a template is said to match a tuple field if they have the same type. If the template field is not formal, both fields must also have the same value. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field. In this way, unlike traditional DSM abstractions, addressing in a tuplespace is associative meaning that it is *content-addressable.*

Write and Blocking Operations. The basic operations defined on a tuplespace insert tuples into the tuplespace using the `write` operation (*out* in Linda) and withdraw tuples from the tuplespace using the `wait-to-take` operation (*in* in Linda). Also, it is possible to inspect a tuple without having to withdraw it from the tuplespace using the `wait-to-read` operation (*rd* in Linda). Withdrawal of tuples is mutually exclusive, which is ensured by the atomicity of the tuplespace operations. The `wait-to-take` and `wait-to-read` operations use a *template* to lookup tuples within the tuplespace. When these operations are performed, the tuplespace is searched for tuples having the same type signature as the template, and having *matching* values. Wild cards match any value of the same type. When multiple tuples match a template, an arbitrary value is returned to the process. If the tuplespace does not contain any matching tuple then the operations block until a matching tuple becomes available in the tuplespace.

Non-Blocking Operations. Linda includes predicate variants of the two operations take and read, named `take` (*inp* in Linda) and `read` (*rdp* in Linda). These are non-blocking versions of the operations, meaning that if no tuple matches the template provided, these operations do not block. Instead, they return a value indicating that no matching tuple was found. In general, the non-blocking operations `read` and `take` cause a problem because they inspect the present state of a tuplespace, and one could argue that they are inappropriate in a distributed environment, where the most recent operation is not defined [9]. Since there is no mutual exclusion on tuplespaces (only on individual tuples residing within it) it is semantically unclear what it means when the operation return 0 (a value indicating that no matching tuple was found). The FT-Linda tuplespace implementation by Bakken and Schlichting in [1] addresses this issue.

[1]The words application and process are used interchangeably.

In their implementation `take` and `read` provide absolute guarantee as to whether there is a matching tuple or not. They refer to this property as *strong read/take semantics*. Some implementations do not guarantee this, which means that an operation returning a value indicating that no matching tuple was found does not guarantee that there was no matching tuple when the operation was invoked. This property is referred to as *loose read/take semantics*.

Ordering of Operations. The ordering of tuplespace operations is an important property of the tuplespace paradigm. The tuplespace data model distinguishes between *globally unordered* operations and *partially ordered* operations seen from a process' point of view. Because the tuplespace operations are issued by processes that are independent, meaning that they do not mutually coordinate the ordering of their operations, the tuplespace operations are globally unordered. However, the tuplespace operations are also partially ordered. By partial order is meant that the tuplespace operations are performed sequentially by every process in accordance with the process' execution order.

Multiple Matching. A problem related to *matching* is the so-called multiple matching problem – a semantic problem that has caused a lot of arguing [12]. The problem origins when multiple tuples matches a `read` or `wait-to-read` operation. The `read` operation just returns one arbitrary matching tuple, but suppose the process wants to list all tuples matching a given template. Another way to put this problem is: how can the process iterate over a set of matching tuples? This feature is not included in Linda. An equivalent problem exists for the `take` and `wait-to-take` operation. The reason iteration is not part of the tuplespace paradigm is semantic consistency [12]. New matching tuples could show up while iterating. If they also match, the iteration could run infinitely. If they do not, locking on tuplespace level would be necessary when starting the iteration. Then new matching tuples would be invisible. In any case the iteration operations provide a snapshot of the tuplespace and the state of the tuplespace will most likely change immediately. Thus, if an application insists that iteration is needed, then a tuplespace is probably not the right foundation for that application.

2.2 Centralized Tuplespace Semantics

Semantic Result. The semantics of the Linda tuplespace operations is vague since it was not formally defined in [6]. Several attempts have be made to define the Linda semantics, but as the survey article by Campbell, Osborne and Wood [2] describes, there has been no consensus in the literature. In [12] the authors thoroughly analyze and discuss the semantics of centralized tuplespaces. The following describes a synthesis based on this analysis. In the previous section, the semantics of each of the centralized tuplespace operations `write`, `read`, `wait-to-read`, `take` and `wait-to-take` were described. An important semantic property was left out; the semantic impact of concurrent operations. The following focuses on the semantics of concurrent tuplespace operations, specifically the semantic implications of ordering the concurrent tuplespace operations internally in the tuplespace. In other words, internally the tuplespace must perform one tuplespace operation at a time – even tuplespace operations received concurrently. The implications of choosing to perform one tuplespace operation before the other is the core issue, which can affect the *semantic result* from a process' point of view.

Interference. The operations must *interfere* in order for concurrent processes to be affected by the internally tuplespace operations ordering [12]. Operations that do not interfere are semantically equivalent to sequential operations. For two or more concurrent tuplespace operations to potentially interfere they must (1) operate on tuples having the same type signature, and (2) the result of one operation influences the result of other operations. For example, given a tuple `t` and a tuple template $t_{template}$ each having the same type signature, an operation `write(t)` inserting `t` potentially influences the result of a successive `wait-to-take(`$t_{template}$`)` operation withdrawing a tuple matching $t_{template}$. For simplicity, in the following sections it is assumed that unless otherwise stated, concurrent tuplespace operations always match the specific tuple `t` – they always interfere. In relation to `wait-to-read`, `wait-to-take` and their non-blocking variants, it means that the processes use the same tuple template. In relation to `write`, it means that only tuples matching the `wait-to-read`, `wait-to-take` operations and their non-blocking variants are inserted into the tuplespace. In addition, it is assumed that at any given time at most one tuple in the tuplespace matches the template used. For simplicity only combinations of two concurrent tuplespace operations are considered as this could be generalized to cover combinations of any finite number of concurrent operations.

Categories of Dependency. Table 1 [12] shows the combinations of all the tuplespace operations and shows the relation between semantic result and internal ordering of concurrent tuplespace operations. For simplicity, the table only shows the values in the table diagonal and up as the concurrency relation is cummutative, which means that the values in the table are symmetrical in the diagonal. The dependency between semantic result and internal ordering of concurrent tuplespace operations is categorized in three categories: (i) *Independent (I)*, (ii) *Dependent (D)*, (iii) *Strongly Dependent (SD).* In the first category, the semantic result is independent of internal ordering of concurrent tuplespace operations.

$Op1 \parallel Op2$	`write(t)`	`wait.read(t)`	`wait.take(t)`	`read(t)`	`take(t)`
`write(t)`	I	I	I	D	D
`wait.read(t)`		I	D	I	D
`wait.take(t)`			SD	D	SD
`read(t)`				I	D
`take(t)`					SD

Table 1: Dependency between semantic result and internal ordering of concurrent tuplespace operations. [12]

Weak Dependency. In the second category, there is a *relaxed* dependency between semantic result and internal ordering of concurrent tuplespace operations. Table 1 shows six combinations of concurrent tuplespace operations that are dependent. In these combinations, the semantic result is affected by the internal operation ordering. The combinations of concurrent tuplespace operations in this category $Op1 \rightarrow Op2$ is *not semantically equivalent* to $Op2 \rightarrow Op1$. Since tuplespace operations by definition are globally unordered and the two operations are *concurrent* both results are *semantically valid* from the processes' point of view. Internally, the tuplespace can choose to perform either operation first, and the semantic result will depend of this choice, but they are semantically valid from the processes' point of view. The processes have no way of knowing which operation will be performed first. In order to satisfy as many processes as possible, it could be argued that the sequence where both processes are allowed to continue, in the example `wait-to-read` $\rightarrow$ `wait-to-take`, should be chosen when possible. This semantic flexibility is interesting in relation to the semantics of a distributed tuplespace, as will be described later.

Strong Dependency. In the third category there is a *strong* dependency between semantic result and internal ordering of concurrent tuplespace operations. Table 1 shows that concurrent tuplespace operations included in this category includes the four permutations of concurrent `wait-to-take` and `take` operations. The atomicity of the tuplespace operations mentioned in Section 2.1 ensures that only one process can withdraw the tuple. If both processes were allowed to withdraw the tuple the semantics of the tuplespace would be incorrect, and as a consequence the coordination feature of the tuplespace would be lost. Since only one process is allowed to withdraw the tuple, this category is denoted *strong* dependency. The two concurrent operations can be internally ordered and performed as follows. First, a process `X` successfully withdraws the tuple by performing the `wait-to-take(t)` operation, and then a process `Y` tries to withdraw the tuple `t` by performing the `wait-to-take(t)` operation, but since there is no longer a matching tuple in the tuplespace process `Y` blocks. A alternative scenario is: first, `Y` successfully withdraws the tuple `t` by performing the `wait-to-take(t)` operation, and then `X` tries to withdraw the tuple by performing the `wait-to-take(t)` operation, but since there is no longer a matching tuple in the tuplespace process `X` blocks.

The same result would have been achieved using an example with a non-blocking operation in combination with another non-blocking operation or a blocking operation. The only difference occurs when the non-blocking operation is performed last. Instead of blocking as the blocking operation, the non-blocking operation continues and returns a value indicating that no matching tuple was found. In the example above, the semantic result is affected no matter what combination of internal tuplespace operation ordering is chosen, and as such is strongly dependent on the internal operation ordering. Therefore, for the combinations of concurrent tuplespace operations in this category `X` $\rightarrow$ `Y` is *not semantically equivalent* to `Y` $\rightarrow$ `X`. Tuplespace operations are by definition globally unordered as mentioned previously. Since the two operations are *concurrent* both results are *semantically valid* from the processes' point of view. Internally, the tuplespace can choose to perform either operation first, and the semantic result will depend of this choice, but they are semantically valid from the processes' point of view. The processes have no way of knowing in which order operations will be performed. The tuplespace paradigm semantics does not dictate an internal operation ordering. Often the internal operation ordering is arbitrary as this should seek to avoid starvation of one process [4]. However, it is left to the specific tuplespace implementation to handle this situation.

3 The TSpaces Model

TSpaces is network middleware for the new age of ubiquitous computing. It is implemented in the Java programming language, and thus it automatically possesses network ubiquity through platform independence, as well as a standard type representation for all datatypes. It extends the basic Linda tuplespace framework with real data management and the ability to download both new datatypes and new semantic functionality. The salient features of the TSpaces system are [15]:

Tuplespace Operator Superset. TSpaces implements the standard set of tuplespace operators. In addition, it includes both blocking and nonblocking versions of take and read, set-oriented operators such as scan and consumingscan, and a novel rendezvous operator, rhonda, explained later.

Dynamically Modifiable Behavior. In addition to the expanded set of built-in operators, TSpaces allows new operators to be defined dynamically. Applications can define new datatypes and new operators that are downloaded into the TSpaces server and used immediately. This is in contrast to relational database systems that have limited datatype support and limited dynamic function (usually in the form of triggers).

TSpaces Solutions for the Multiple Matching Problem. One solution presented in TSpaces is adding an operation `readall` to the set of basic operations. This operation reads all matching tuples in the tuplespace and returns a copy of them to the client. A similar operation `takeall` is added to solve the equivalent multiple `take` problem. This operation *removes* all matching tuples from one tuplespace. Another solution is to change the application using the tuplespace instead. One way is to include an index field in the tuples and iterate over that field when performing `read` and `take` operations.

Persistent Data Repository. TSpaces employs a real data management layer, with functions similar to heavyweight relational database systems, to manage its data. T Spaces operations are performed in a transactional context that ensures the integrity of the data.

Database Indexing and Query Capability. The TSpaces data manager indexes all tagged data for highly efficient retrieval. The expanded query capability provides applications with the tools to probe the data with detailed queries, while still maintaining a simple, easy-to-use interface.

Access Controls and Event Notification. Users can establish security policies by setting user and group permissions on a Tuplespace basis. Applications can register to be notified of events as they happen in the TSpaces server. TSpaces is appropriate for any application that has distribution or data storage requirements. It can perform many of the duties of a relational database system without imposing an overly restrictive (and primitive) type system, a rigid schema, a bulky user API (application programming interface), or a severe run-time memory requirement. In a sense, it is a database system for the common everyday Tier-0 computer–one that does not generate complex SQL (Structured Query Language) queries, but one that needs reliable storage that is network accessible.

4 Tuplespace Replication Issues

This session describes the theoretical issues involved in tuplespace distribution using replication. First, it gives an overview of the primary reasons for choosing a replication strategy in a distributed system, and the problems that follows. It then goes into details with specific replication issues where different approaches are presented and analyzed.

4.1 Data Replication Overview

Replication is the task of maintaining multiple copies of some data, called *replica*, while providing the illusion of a single piece of data. As such, replication builds on a strategy of redundancy of data. There are several reasons for using a replication strategy in a distributed system. The most common are [12]: (i) *fault-tolerance* - distributed systems are exposed to partial failures and are therefore ideal for replication - if one replica becomes unavailable, other replicas are likely to be available, and data availability is therefore increased; (ii) *scalability* - replicated systems can in theory grow indefinitely, as they can grow modularly by simply adding another replica to the replicated system - in contrast, a centralized system has physical limitations to its growth potential; (iii) *performance* - distributed systems are exposed to delays imposed by communication, which have an impact on performance - by replication it is possible to reduce the communication delays by bringing the applications closer to the replicas.

These advantages of replication comes at a cost in terms of practical as well as theoretical problems related to replication. The problems are [12]: (i) *inconsistency* - having replicas spread over multiple computers may eventually lead to inconsistency as one or more replicas are changed - therefore, a protocol to ensure consistency by managing all replicas must be implemented as part of the replication scheme; (ii) *scalability* - replication may enforce limitations to the number of replicas, thereby reducing the scalability potential - the limitations arise due to the amount and frequency of communication and co-operation needed to maintain consistency between replicas; *performance* - as mentioned, replication comes at a cost in terms of a replication overhead - maintaining multiple copies rather than a single copy of some data is naturally more costly, which may reduce performance to such an extent that the system is no longer useful.

So, there are competing goals for selecting a replication strategy. As mentioned, these goals are not independent, thus trade-offs are necessary and have to be mutually evaluated. Despite of these practical problems, replication is still an attractive strategy to achieve fault-tolerance, scalability and performance. However, these practical problems have to be taken into account to fully exploit the advantages of replication.

4.2 Replication Elements

There are two different ways in which a system can be replicated. Either a system can be replicated by replicating the state of its data structure or by replicating the operations changing the state of its data structure [12]. Translated to the tuplespace data model, a tuplespace can either be replicated by replicating the state of the data structure representing the tuplespace or by replicating the operations issued on the tuplespace.

State Replication. By replicating the state of the distributed tuplespace, the tuplespace is conceptually viewed as a data structure. As such, state replication is replication of that data structure. The state replication approach is used in FT-Linda [1]. There are two different approaches to state replication: (i) full state replication, and (ii) partial state replication. Full state replication replicates the entire tuplespace data structure each time some part of it changes. In object-oriented environments full state replication means that the data structure, represented by an object graph, must be serialized into a byte stream in order to be transmitted over the network. However, transmitting the entire serialized data structure is costly if the data structure is large and changes frequently. This may impose severe performance as well as scalability problems. A classical optimization is to extract and replicate only the part of the data structure, which have changed since last replication – partial state replication. By replicating only the extracted differences in the data structure, the byte stream transmitted over the network is reduced to enhance performance and scalability.

Operation Replication. By replicating the operations issued on the distributed tuplespace, the tuplespace is conceptually viewed as a result of a sequence of operations issued on the tuplespace. This means that all tuplespace operations changing the state of the distributed tuplespace need to be replicated to the other tuplespace replicas. The operation replication approach is used in [16]. The state of the tuplespace only changes if a tuple is inserted into or withdrawn from the tuplespace. This means that it is necessary to replicate `write`, `take` and `wait-to-take` operations, which all change the tuplespace state. The `read` and `wait-to-read` operations, which read tuples from the tuplespace without withdrawing them, do not change the state of the tuplespace, and do not necessarily have to be replicated.

4.3 Distributed Tuplespace Semantics

Transparency. Ideally, the semantics of distributed tuplespace operations should be the same as the centralized tuplespace operations, or at least it should be transparent to a process if it uses a centralized tuplespace or a distributed tuplespace. As such, the semantic result of the centralized tuplespace operations as shown in Table 1 should be the same as the distributed tuplespace operations. The issues of semantic validity, as mentioned in the previous sections, and internal operations ordering give some flexibility, which can be exploited in the semantics of a distributed tuplespace. In the following, the discussion of tuplespace semantics is exemplified using a distributed tuplespace abstraction containing two consistent tuplespace replicas. This example can be generalized to cover any finite number of tuplespace replicas. Using this example, it is considered how the semantics of the distributed tuplespace is affected by concurrent processes as shown in Table 1. In addition, the assumption from Section 2.2 that concurrent tuplespace operations always match a specific tuple `t` – they always interfere – is retained. In addition, it is assumed that at any given time at most one tuple in each of the tuplespace replicas in the distributed tuplespace matches the template used.

Global Atomicity. In relation to a distributed tuplespace, the notion of *sequential* and *concurrent* operations have a different meaning. When referring to sequential and concurrent tuplespace operations or their internal ordering in the following sections, this should be seen from the point of view of the entire distributed tuplespace abstraction. As such, two operations are concurrent in a distributed tuplespace if the result of one tuplespace operation is not visible in the entire distributed tuplespace, when another tuplespace operation is performed. In addition, in relation to a distributed tuplespace the notion of atomicity of tuplespace operations also have a different meaning. To differ between atomicity in a centralized tuplespace and a distributed tuplespace, the latter case is referred to as *global atomicity.* Global atomicity means that tuplespace operations are performed atomically on the distributed tuplespace, that is, on all the tuplespace replicas in the distributed tuplespace or none. In the following, we discuss the semantics of the three categories of concurrent operations presented in Section 2.2: (i) independent, (ii) dependent, and (iii) strongly independent. The semantics of combinations of concurrent tuplespace operations in the first category is trivial. As combinations of concurrent tuplespace operations in this category do not interfere when performed, they can be performed in any order regardless of a centralized tuplespace or a distributed tuplespace.

Weak Dependency. As discussed in Section 2.2, the dependent category has a more *relaxed* dependency between semantic result and internal ordering of concurrent tuplespace operations. Imagine that the processes `X` and `Y` concurrently performs a `wait-to-read(t)` and an `wait-to-take(t)` operation respectively, but each operation is performed on different tuplespace replicas T_1 and T_2 respectively. Imagine that `X` performs `wait-to-take(t)` on tuplespace replica T_1 and successfully withdraws the tuple `t` from T_1. T_1 sends a notification message to tuplespace replica T_2 requesting it to also withdraw tuple `t` in order to ensure consistency. Now, concurrently `Y` performs `wait-to-read(t)` on T_2 and successfully reads tuple `t` from T_2. Shortly after, T_2 receives a notification message from T_1. The lack of atomicity in the tuplespace operations on the distributed tuplespace enables one operation overtake the other due to delay in the propagation of tuplespace operations. However, this situation is *not semantically incorrect.* As noted in Section 2.2, the different outcomes are *not semantically equivalent*, but since there is no global physical time in the distributed system scenario they are *semantically valid* due to the semantic flexibility described in Section 2.2. Thus, to satisfy as many processes as possible, from a semantic result point of view this situation is acceptable.

The consequence is that for the operations `write`, `read` and `wait-to-read` it is not *necessary* to enforce global atomicity as the semantics is valid without this property. The `take` and `wait-to-take` operations must still have global atomicity. For the `write` operations this means that a tuple inserted does not need to become visible immediately at all tuplespace replicas in the distributed tuplespace. Another consequence of this is reported by Bakken and Schlichting in [1] and is known as *loose take/read semantics.* Loose take/read semantics implies that the non-blocking operations could return a value indicating that no matching tuple was found even though the distributed tuplespace actually contained a matching tuple that was not yet propagated to all tuplespace replicas.

Strong Dependency. As discussed in 2.2, the strongly-dependent category has a *strong* dependency between semantic result and internal ordering of concurrent tuplespace operations. Suppose that the processes `X` and `Y` concurrently perform `wait-to-take(t)` operations, but each `wait-to-take` operation is performed on different tuplespace replicas T_1 and T_2 respectively. The problem here is that unless prevented this situation leads to a state of conflict – something that is semantically incorrect. Imagine that `X` performs `wait-to-take(t)` on tuplespace replica T_1 and successfully withdraws the tuple `t` from tuplespace replica T_1. T_1 sends a message to tuplespace replica T_2 requesting it to also withdraw tuple `t` in order to be synchronous. Now, concurrently `Y` performs `wait-to-take(t)` on T_2 and successfully withdraws tuple `t` from T_2. T_2 sends a message to T_1 requesting it to also withdraw tuple `t`. Shortly after, both T_1 and T_2 receives a message from each other.

In this example, both processes, `X` and `Y`, would successfully withdraw the same tuple from the distributed tuplespace which is semantically incorrect by definition since withdrawal of tuples should be mutually exclusive. Another consequence is that the semantic result is no longer the same. Therefore, only one process should be allowed to withdraw the tuple and the other should block. The source of the problem is that the atomicity property of the centralized tuplespace is lost in this replication scheme. Therefore, to avoid this situation the atomicity property – *global atomicity* – should be applied to the distributed tuplespace abstraction to ensure that the scope of the tuplespace operations is *global* in the distributed tuplespace instead of only at a tuplespace replica. By enforcing global atomicity only one process would successfully withdraw the tuple, thus the semantic result would be consistent with that described in Section 2.2. This means that the results of the operation ordering are *semantically not equivalent*, but seen from the processes' point of view *semantically valid*.

4.4 Consistency Models

The semantics of a shared memory system is expressed through a consistency model. Consistency models are often categorized into respectively *strong* and *weak* consistency models [13]. Strong consistency models are characterized by providing the application using the replicated system with the illusion of non-replicated data, whereas weak consistency models do not. In general, to optimize the replicated system, that is, maximize performance and scalability and still ensure the semantics of the system, it is advantageous to choose the weakest consistency model possible as performance and scalability often increases when relaxing the consistency. However, the requirement of global atomicity is ensured by the consistency models providing *strong* consistency. The following two strong consistency models are considered: (i) strict consistency, and (ii) sequential consistency. An execution that satisfies strict consistency also satisfies sequential consistency [8]. In the following sections strict and sequential consistency are described.

Strict Consistency. Strict consistency, also known as linearizability, defines the most restrictive consistency model of the two, and is characterized by the condition that any read to a memory location x returns the value stored by the most recent write operation to x. As can be seen from this characterization, a system is strictly consistent if operations in a multiprocessor system are issued in an order as if it was a uniprocessor system. A distributed system is said to be strictly consistent if operations are executed in a real-time order according to the physical global time at which they are issued. To do so, the definition assumes the existence of a physical global time used to order the operations. However, enforcing an absolute global time in a distributed system is not trivial.

Sequential Consistency. Sequential consistency is a relaxation of strict consistency. Sequential consistency is described by Lamport in [11] as a multiprocessor system, where the result of any execution is the same as if (1) the operations of all processors were executed in some sequential order, and (2) the operations of each individual processor appear in this sequence in the order specified by its program [11]. As in a strictly consistent system, a system is sequentially consistent if operations in a multiprocessor system are issued in an order as if it was a uniprocessor system. Unlike strict consistency, no real-time global clock is needed for maintaining sequential consistency. As can be seen by comparing strict consistency with sequential consistency, the former uses a physical real-time clock whereas the latter says nothing about the time – it only says something about the logical ordering. As such the processes do not have to agree on the exact time, but have to agree on an exact operation order. In other words, sequential consistency can be viewed as strict consistency without the requirement of real-time ordering of non-concurrent operations.

4.5 Replica Update Protocols

Based on the consistency model as described in the past subsection, the next issue deals with the selection of an appropriate replica update protocol which ensures the sequential consistency necessary for the `take` and `wait-to-take` operations. In order to ensure the sequential consistency, the following conditions must be met by the replica update protocol: (i) *global atomicity* - the replica update protocol must guarantee that `take` and `wait-to-take` operations are atomic, that is, they are performed on all tuplespace replicas in the distributed tuplespace or none; and (ii) *order* - the replica update protocol must guarantee that tuplespace operations where required are performed in the correct order according to the sequential consistency model. No global ordering or atomicity is necessary for the `write`, `read` and `wait-to-read` operations. The operations are only partially ordered from the application point of view. Three different replica update protocols that ensure the consistency model have been considered: (i) *primary copy replication*, (ii) *active replication*, (iii) *weighted voting.*

Primary Copy Replication. In primary copy replication one of the tuplespace replicas plays a central role – it is the *primary* while the others are so-called *backups.* As such, the primary is responsible for receiving the operation invocations from all processes using the distributed tuplespace, propagate the operations to the other tuplespace replicas, and sending results back to the processes. An important issue in primary copy is the tightness of synchronization between the primary and the backup tuplespace replicas. The scenario above uses tight synchronization meaning that changes to the primary tuplespace replica are immediately propagated to the backups. An alternative is loose synchronization based on a batch-oriented approach where multiple changes are performed on the primary tuplespace before synchronization. Tight synchronization increases fault-tolerance at the cost of performance. Loose synchronization allows for independent updates to the tuplespace thereby increasing performance at the cost of fault-tolerance (and false confidence) as changes may be lost due to a crash of the primary. To use loose synchronization between the primary and backups, the primary copy replication must be non-blocking, meaning that the primary returns the result of the operation to the invocating process before receiving acknowledgements from the backups [8].

If the primary should fail at some point, a backup takes over and acts as new primary. To the processes a crash of a backup tuplespace replica is transparent, but a crash of the primary tuplespace is *not* transparent when using primary copy replication. The classical problem of the primary copy replica update protocol is handling a crash of the primary tuplespace. Another important issue is the case where the distributed tuplespace is subject to network partitioning. Here the problem is to avoid that each of the network partitions elect their own primary tuplespace replica, which may lead to incorrect semantics, for instance if the same tuple is withdrawn by different processes in each network partition. To avoid this it must be possible to distinguish if a failure originates from a tuplespace replica crash or network partitioning. In case the primary crashes, an election should be initiated and one of the backups should take over the role as primary as mentioned above. However, in case of network partitioning no election should be initiated and the partition containing the primary copy continues to function. In practice crashes and network partitioning are indistinguishable failures in loosely coupled network environments that Enterprise TSpaces 5 is targeted for. This problem is insoluble unless additional hardware facilities are added.

Active Replication. In active replication all tuplespace replicas have the same responsibility. This scheme is used to implement a distributed tuplespace in FT-Linda [1]. As such, all tuplespace replicas are responsible for receiving the operation invocations from the processes using the distributed tuplespace, propagate the operation invocations to the other tuplespace replicas, and return the results to the processes. Specifically, a scenario with active replication can be described as follows: (i) process `X` sends the operation `wait-to-take(t)` to all the tuplespace replicas, (ii) upon receiving the invocation, the `wait-to-take(t)` operation is performed on each tuplespace replica, (iii) having performed the operation, each tuplespace replica acknowledges to the process `X`, (iv) the process `X` receives acknowledgements from one tuplespace replica or the majority of tuplespace replicas depending on the level of fault-tolerance before continuing.

As can be seen in the example above, the active replication ensures all operations are performed atomically and in the same order which could ensure the sequential consistency needed in Enterprise TSpaces 5. The main problem of active replication is ensuring atomicity of the operations. This functionality is often provided by either using a transaction based mechanism, for instance two-phase commit or atomic multicast. While the former is relatively easy to implement the latter is non-trivial to implement. Common to both is that these mechanisms are costly in terms of performance and scalability. In relation to failures, a crash of a tuplespace replica is transparent to the processes when using active replication – the processes do not have to reissue the operations as the other tuplespace replicas have performed the operations. However, reintegration of a failed tuplespace replica into the distributed tuplespace upon recovery is non-trivial as the replicas might be inconsistent.

Weighted Voting. Another technique, based on distributed control, is that of weighted voting proposed by Gifford [7]. The weighted voting technique is used in a distributed tuplespace design in [10]. By voting is meant that an update on replicated data is decided collectively. In short, the weighted voting strategy is based on obtaining a majority quorum using votes among replicas when performing updates of a replica – in this case given the consistency model when performing `take` and `wait-to-take` operations. If a majority can not be established the update, `take` and `wait-to-take` operations, can not succeed.

Unlike the primary copy replica update protocol, all tuplespace replicas can initiate a withdrawal of a tuple. This approach potentially leads to race-conditions as multiple tuplespace replicas can initiate withdrawal of the same tuple. However, voting ensures that concurrent `take` and `wait-to-take` operations on the same tuple are performed sequentially by enforcing ordering of the operations, thus only one succeeds at a time. In addition, the approach potentially also leads to deadlock situations where no process is able to obtain majority for tuple withdrawal, for instance having four tuplespace replicas of which two votes for and two votes against. This problem also exists in the distributed tuplespace implementation by Xu and Liskov in [16], but its occurrence is solved by introducing a random delay to competing, deadlocked processes thereby trying to inflict an order of the majority requests. The problems of race-conditions and deadlocks are exactly what is avoided in primary copy replication by having a centralized point of control. The weighted voting strategy is advantageous in that it does not require any negotiation in case a tuplespace replica becomes unavailable since there is no central point of control – the voting continues, but with fewer votes. The voting technique also avoids the insoluble problem of distinguishing between tuplespace crashes and network partitioning failures.

5 The Enterprise TSpaces Model

Enterprise TSpaces (ETS) is a further development of the stand-alone version of TSpaces that provides TSpaces with industry-strength enterprise required facilities such as fault-tolerance. The previous session described the theoretical issues involved in tuplespace distribution using replication. Section 5.1 overviews the main features of ETS. In the following subsections, for each of these replication issues, the approach selected in ETS is presented.

5.1 The ETS Distributed Semantics

To ensure correct semantic behaviour by ensuring consitency and extending the scope of atomicity ETS distinguishes between two categories of tuplespace operations: (i) strong operations - operations requiring atomicity across all tuplespace replicas. This category includes the operations for tuple withdrawal - in and inp; (ii) weak operations - operations not requiring atomicity across all tuplespace replicas. This category includes the operations for tuple insertion and inspection - write, read and wait-to-read. From this categorization, ETS has a loose take/read semantics. The motivation for this is to improve performance in terms of operation execution time from the processes' point of view. In practice, this means that the `write` operation is categorized as a weak operation instead of a strong operation. The consequence of this placement is that a `read` or `take` operation returning a value indicating that no matching tuple was found does not actually guarantee that there was no matching tuple on another tuplespace replica in the distributed tuplespace when the operation was invoked.

To summarize the analysis of the complete set of operation combinations in the distributed tuplespace, the tuplespace operations are listed and the semantics of the distributed tuplespace in ETS are described: (i) `write` - is performed without global atomicity meaning that a tuple inserted is not necessarily visible immediately at all tuplespace replicas in the distributed tuplespace, (ii) `wait-to-read` - is performed without global atomicity, but the semantics of the operation is the same as in the centralized tuplespace, (iii) `wait-to-take` - is performed with global atomicity meaning that only one process can withdraw a specific tuple, (iv) `read` - is performed without global atomicity meaning that the `read` operation may in fact return a value indicating that no matching tuple was found even though a matching tuple is present in the distributed tuplespace (*loose take/read* semantics), (v) `take` - like the `wait-to-take` operation, the `take` operation is performed with global atomicity meaning that only one process can withdraw a specific tuple. The `take` operation also may in fact return a value indicating that no matching tuple was found even though a matching tuple is present in the distributed tuplespace.

5.2 Using the Sequential Consistency Model

Whereas the *strong* operations of ETS require a specific consistency model this is not necessary for the *weak* operations. The only requirement needed for *weak* operations is that of ensuring the partial ordering of the operations on all destination replicas. As can be seen from the description of strict consistency and sequential consistency both consistency models are suitable for ensuring the global atomicity requirement. ETS uses a sequential consistency model for strong operations. The reasons for this is that implementation of sequential consistency is simple using logical ordering of operations instead of ordering them according to a physical clock. Even though strict consistency is the ideal consistency model as operations are totally ordered in the real-time order in which they occur, the strict consistency model also has drawbacks as mentioned earlier. It relies on the existence of an physical global time used to order the operations. Implementing a physical global time is far from trivial, and implemented it relies heavily on coordination protocols imposing practical scalability problems when the number of replicas increases [13].

5.3 Operations as Replication Elements

As can be seen from the previous description of state replication and operation replication both are possible as replication methods, though full state replication is not practically usable because of the overhead involved. It is important to note that the choice of replication method has no consequence on the semantics of the operations. ETS uses a replication method based on replication of operations. The reason for this is that replicating operations is easier and provides a better foundation for scalability and performance. To replicate state, the state difference must first be extracted every time a state change has occurred and then replicated to the other tuplespace replicas. To replicate operations, the operations issued should simply be caught before they are performed and replicated to the other tuplespace replicas. Thus, by replicating operations instead of state, the cost of extracting the state difference is saved.

By replicating operations a tight integration with a specific tuplespace implementation is also avoided, giving an open solution which could easily be extended to include multiple tuplespace implementations without regard to the internal representation of the tuplespace. To traverse the internal data structure or to extract state difference, the replication mechanism must be tightly integrated with a specific tuplespace implementation in order to know its internal data structure. This tight integration results in a dependency on the specific tuplespace implementation used, which might not be preferable. Also, extracting the state difference itself is not trivial, and the techniques might not be reusable between multiple tuplespace implementations.

5.4 Replica Update Protocol based on Voting

ETS uses a replica update protocol based on voting. A majority voting is used and each tuplespace replica in the distributed tuplespace is assigned one vote. ETS uses the same approach as used by Xu and Liskov [16] to address deadlocks. Also, ETS exploits the random delay, which lies implicitly in the latency time experienced on loosely coupled networks. The reason for using voting instead of primary copy is that a primary copy potentially becomes a bottleneck as the primary must handle all operations. Also, the primary copy technique requires that it is possible to distinguish if a failure originates from a tuplespace replica crash or network partitioning. These are indistinguishable failures on the loosely coupled network that ETS operates in.

Active replication avoids the bottleneck situation of the primary copy but requires protocols like two-phase commit or atomic broadcast. These techniques are costly, specifically on a loosely coupled network, which is the primary reason for not choosing active replication. Also, reintegration of a failed tuplespace replica into the distributed tuplespace upon recovery is non-trivial in this approach. Recovery and reintegration of a failed tuplespace replica is an important property of ETS.

5.5 Dynamic Adjustability: Replication Groups

ETS provides fault-tolerance on different abstraction levels. By using operation replication between TSpaces servers in so-called replication groups the availability of tuples increases with the number of TSpaces servers. This increases the tolerance to failures of TSpaces servers and thereby eliminating the single point of failures that TSpaces currently suffers from. In addition to providing fault-tolerance ETS provides support for this property to be dynamically adjusted at runtime. It is archieved by dynamically increasing or decreasing the number of TSpaces servers in a replication group. Another aspect of adjustability is that it is possible to dynamically add or remove tuplespaces to be replicated between TSpaces servers. In the following, we introduce the terminology for using replication groups.

Terminology. The term *TSpaces server* denotes a tuplespace server holding any finite number of tuplespaces. The TSpaces server can either be used as a stand-alone tuplespace server with non-replicated tuplespaces, or it can participate in one or more *replication groups* with other TSpaces servers and replicate some or all of the tuplespaces it holds. The term *replicated tuplespace* denotes a tuplespace being replicated between a finite number of TSpaces servers in a *replication group*. The term *replication group* denotes a finite named collection of TSpaces servers that participate in replication of tuplespaces in the replication group. A TSpaces server can participate in multiple replication groups at the same time. However, a *tuplespace* can only be a member of one replication group at a time. The term *client* or *client application* denotes the application using Enterprise TSpaces, and not the TSpaces proxy code that resides on the client side.

Enterprise TSpaces replicates tuplespaces by using a *replication group* abstraction on top of the TSpaces servers. A TSpaces server can either be a stand-alone server or be a member of one or more replication groups. As member of a replication group a TSpaces server participates in the replication of the tuplespaces assigned to the replication group. A tuplespace on a TSpaces server can either be a member of zero or one replication group. Being member of a replication group, the tuplespace is replicated between the TSpaces servers that participate in the replication group. If a tuplespace is not a member of a replication group the tuplespace is non-replicated.

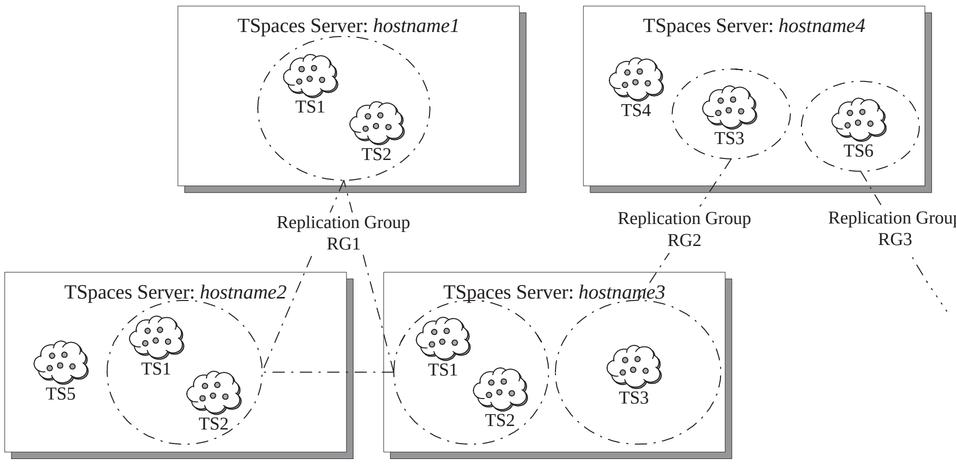
TSpaces Server: *hostname1*
TS1
TS2
TSpaces Server: *hostname4*
TS4
TS3
TS6
Replication Group RG1
Replication Group RG2
Replication Group RG3
TSpaces Server: *hostname2*
TS5
TS1
TS2
TSpaces Server: *hostname3*
TS1
TS2
TS3

Figure 1: TSpaces servers replicating tuplespaces in two replication groups

Figure 1 shows four TSpaces servers *hostname1*, *hostname2*, *hostname3* and *hostname4* replicating tuplespaces. The TSpaces servers *hostname1*, *hostname2* and *hostname3* all participate in a replication group called *RG1*. In addition, the TSpaces server *hostname3* participates in another replication group called *RG2* with TSpaces server *hostname4*. Replication group *RG1* contains two replicated tuplespaces *TS1* and *TS2* whereas replication group *RG2* contains one replicated tuplespace *TS3*. In addition, TSpaces server *hostname4* participates in a replication group *RG3* containing a tuplespace *TS6* with some other TSpaces servers, and TSpaces servers *hostname2* and *hostname4* contains non-replicated tuplespaces *TS5* and *TS4* respectively. The following subsections show how: (i) creating and destroying replication groups, (ii) inserting and removing servers from replication groups, and (iii) inserting and removing tuplespaces from replication groups.

5.6 Managing Replication Groups

Creating a Replication Group. Creating a new replication group involves two phases. The replication group is created having only one initial member – the TSpaces server on which the operation is performed on. Then, one by one TSpaces servers are added to the replication group. The code for creating a replication group looks like this: *ReplicationGroup rg = new ReplicationGroup(”RG1”, ”hostname1”, 8260, 10, ”adminid”, ”adminpw”).* Using this example the instantiation of the ReplicationGroup object creates a replication group called *RG1* on the TSpaces server *hostname1:8260.* If the hostname and port number are not set *localhost:8200* is used as default. The constructor of the ReplicationGroup object also takes the administrator userid and password as replication group creation is an administrative operation.

When creating the replication group and adding additional TSpaces servers it is ensured that the name of the new replication group is unique on all the TSpaces servers. In addition, it is checked that the name of the TSpaces servers are unique and that duplicate TSpaces servers are not added. If these criteria are not meet the TSpaces server cannot be added to the replication group, an exception is thrown and the operation is aborted. Having created a replication group, it is successively possible to add tuplespaces to the replication group as well as additional TSpaces servers.

Destroying a Replication Group. Like creating a replication group, a replication group can be destroyed. The code for removing a replication group looks like this: *rg.destroy()*. The instance method initiates destruction of the replication group referenced by *rg*. Destroying a replication group involves two phases. The members of the replication group are removed one by one. Each time a TSpaces server is removed from the replication group a majority voting between the remaining TSpaces servers in the replication group is initiated. Having removed all but one TSpaces server from the replication group the replication group is simply deleted. Destroying a replication group also means that all the replicated tuplespaces in the replication group will no longer be replicated. Instead, each tuplespace will go back being non-replicated on the TSpaces server on which it initially was created. On all the other TSpaces servers in the replication group, the tuplespace will be deleted.

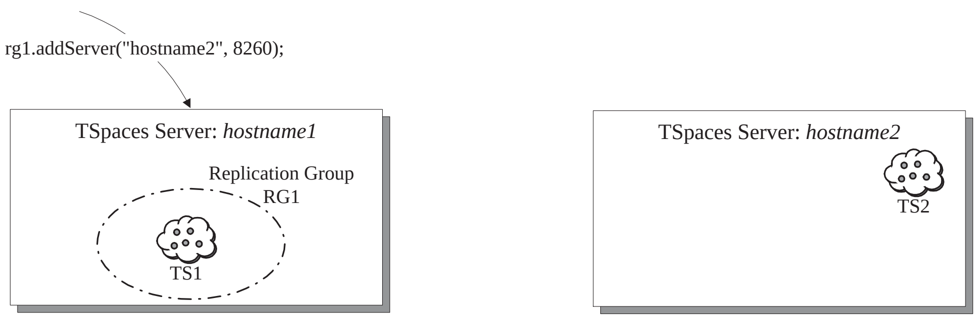
rg1.addServer("hostname2", 8260);
TSpaces Server: hostname1
Replication Group RG1
TS1
TSpaces Server: hostname2
TS2

Figure 2: Before adding TSpaces server to replication group

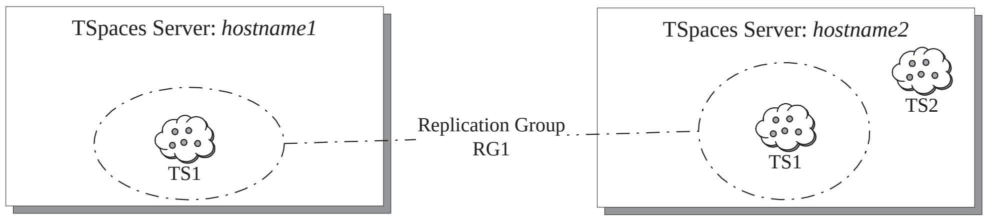

Figure 3: After adding TSpaces server to replication group

5.7 Adjusting Replication Groups

Adding a TSpaces Server to a Replication Group. As mentioned it is possible to adjust the number of TSpaces servers in a replication group. Conceptually, adding a TSpaces server to an existing replication group looks like in Figure 2 and Figure 3. The code to add a TSpaces server to a replication group looks like: *rg.addServer(”hostname3”, 8520, ”adminid”, ”adminpw”)*. The instance method adds the TSpaces server and *hostname3:8520* to the replication group referenced by *rg*. The TSpaces servers will mutually have to request administrative operations. Therefore it is necessary to have administrative access on each of the TSpaces servers. Some might share the same administrative userid and password, and others might not.

Each time a TSpaces server is to be added to an existing replication group a majority voting between the TSpaces servers currently in the replication group is initiated. Only when a majority of the TSpaces servers in the replication group vote for the adding of the new TSpaces server is it added. When adding a TSpaces server to a replication group it is checked that there are no duplicate tuplespace names in the replication group and the new TSpaces server. This is done by requesting the new TSpaces server for permission to lock the rights to use the names of the tuplespaces in the replication group, and the replication group itself. If this request fails, there is a naming conflict. If the new TSpaces server holds a tuplespace with a name that conflicts with a name of a tuplespace in the replication group, the TSpaces server cannot be added to the replication group, an exception is thrown and the operation is aborted. Having obtained a majority in the voting between the TSpaces servers already in the replication group and passed the naming check, the TSpaces server is added to the replication group, which implicitely releases the locks for the names on the new TSpaces server. Immediately thereafter the tuplespaces in the replication group are replicated to the new TSpaces server.

Removing a TSpaces Server from a Replication Group Conceptually, removing a TSpaces server from an existing replication group looks like in Figure 4 and Figure 5. The code to remove a TSpaces server from a replication group looks like: *rg.removeServer("hostname2", 8200).* The instance method removes the TSpaces server *hostname2:8200* from the replication group referenced by *rg.* To remove a TSpaces server from an existing replication group a majority of the TSpaces servers, excluding the TSpaces server to be removed, must agree on removing the TSpaces server. When removing a TSpaces server from a replication group it is ensured that the tuplespace in the replication group are consistent with the tuplespaces on at least a majority of the TSpaces servers in the replication group so that no tuples are lost. If any of the tuplespaces in the replication group were initially created on the TSpaces server that is removed, it is ensured that those tuplespaces are adopted by another TSpaces server in the replication group. Having obtained a majority in the voting between the TSpaces servers remaining in the replication group and passed the consistency check, the TSpaces server is removed from the replication group.

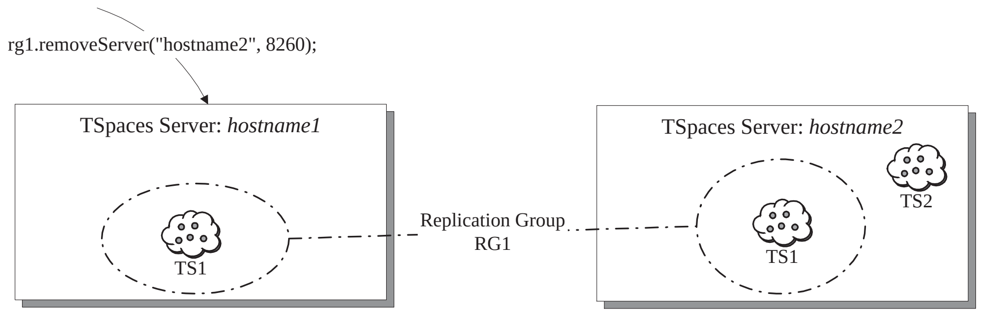
rg1.removeServer("hostname2", 8260);
TSpaces Server: *hostname1*
TS1
Replication Group
RG1
TSpaces Server: *hostname2*
TS1
TS2

Figure 4: Before removing TSpaces server from replication group

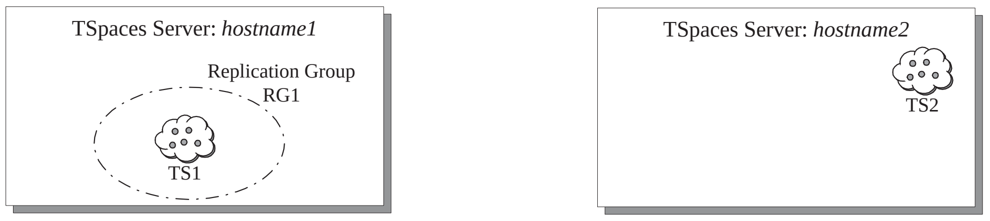

Figure 5: After removing TSpaces server from replication group

5.8 Tuplespaces and Replication Groups

Adding Tuplespace to a Replication Group. Figure 6 shows the conceptual effect on the TSpaces server of adding the tuplespace to the replication group. A tuplespace can successively be added to an existing replication group using this code: *rg.addTupleSpace(ts)*. The instance method adds the tuplespace *TS2* referenced by *ts* to the replication group referenced by *rg*. By adding a tuplespace to a replication group, the tuplespace is automatically replicated to all the TSpaces servers that are members of the given replication group. However, to do so a majority of the TSpaces servers must agree on the tuplespace replication.

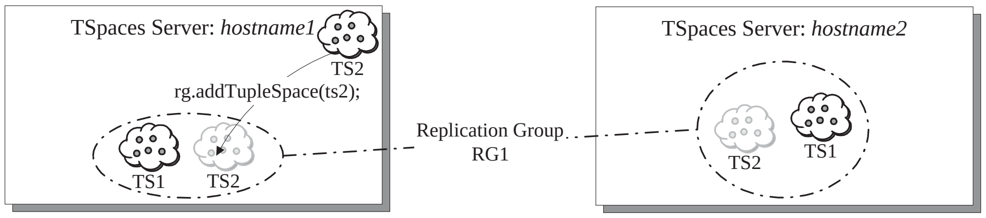

Figure 6: Tuplespace is added to replication group

Upon adding a tuplespace to a replication group it is checked that the name of the tuplespace is unique on all the TSpaces servers in the replication group. If any of the TSpaces servers in the replication group holds another tuplespace in another replication group or a non-replicated tuplespace having the same name the tuplespace should not be added to the replication group. However, if a majority of TSpaces servers vote for there is nothing to prevent it from occuring. On the TSpaces servers that vote against the adding of the tuplespace due to a name conflict the tuplespace can not be created before the other conflicting tuplespace is removed by the Administrator. Upon detecting a name conflict the Administrator is notified.

Removing Tuplespace from Replication Group. Likewise, a tuplespace can be removed from the replication group using this: *rg.removeTupleSpace(ts)*. The instance method removes the tuplespace reference by *ts* from the replication group reference by *rg*. Removing the tuplespace from the replication group means that the tuplespace will no longer be replicated. Instead, the tuplespace will go back being non-replicated on the TSpaces server on which is initially was created. On all the other TSpaces servers in the replication group, the tuplespace will be deleted. Again, to do this a majority of the TSpaces servers must agree on the removal of the tuplespace from the replication group. This is shown in Figure 7.

Tuplespace Adoption. If the TSpaces server on which a tuplespace was initially created is being removed the above mentioned approach will fail. To address this during the removal of the TSpaces server the tuplespace will be *adopted* by another TSpaces server in the replication group. As such, the TSpaces server adopting the tuplespace will be marked as being the TSpaces server on which the tuplespace initially was created. Before a tuplespace is removed from a replication group it is ensured that the tuplespace across *all* TSpaces servers in the replication group is consistent. It is not enough to ensure that only a majority have received all operations. Let's say a client inserts a tuple in a tuplespace on a TSpaces server in the replication group, and that the TSpaces server then crashes before the tuple insertion was sent to any other TSpaces servers. Now, using another TSpaces server the tuplespace is requested to be removed from the replication group. Since a majority of the TSpaces servers in the replication group are up and vote for, the tuplespace is actually removed. However, the now non-replicated tuplespace never got the tuple from the crashed TSpaces server. This makes the operation rather fragile to network failures.

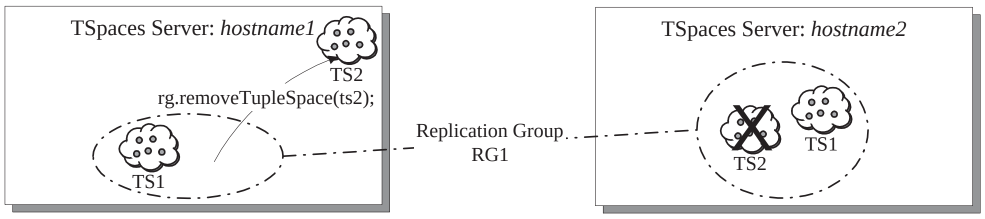
TSpaces Server: *hostname1*
TS2
rg.removeTupleSpace(ts2);
TS1
Replication Group
RG1
TSpaces Server: *hostname2*
TS2
TS1

Figure 7: Tuplespace gets removed from replication group

6 Related Work

Various replica update protocols have been applied in the projects in order to optimize the implementations in different contexts. The S/Net Linda Kernel [3] uses a fast reliable broadcast mechanism, thus ensuring consistency by updating all replicas using the broadcast mechanism. In FT-Linda [1] active replication and an atomic multicast mechanism is used to deliver messages to all replicas reliably and in the same order. The distributed tuplespace implementation by Xu and Liskov [16] uses locking mechanisms and a general commit protocol to perform updates. The replication mechanism used by Kambhatla [10] is based on a weighted voting technique to ensure consistency. MTS [5] uses different replica update protocols among these a two-phase commit protocol. The other protocols relaxes consistency in order to improve performance.

The related projects mentioned above base their replication mechanism on a full replication strategy in order to provide fault-tolerance in terms of availability. As pointed out in [1] and [16] full replication puts an upper limit to the scalability potential in terms of performance of the distributed tuplespace. A well-known technique for solving the scalability problem relating to replication is through partial replication. Both [1] and [16] recognizes the scalability problem of full replication and mention tuplespace partitioning as future work.

7 Conclusions and Ongoing Work

In this work, we presented the tuplespace data model, and identified the problems in replicating tuplespaces. We surveyed the techniques for dealing with such replication problems, and presented a model for fault-tolerant tuplespaces called Enterprise TSpaces (ETS). ETS has a dynamically adjustable level of fault-tolerance which is achieved by using dynamic replication. The level of fault-tolerance is adjustable by dynamically increasing or reducing the number of tuplespace replicas. In addition, ETS uses checkpointing of its internal state to be able to recover a crashed tuplespace replica and reintegrate it into the distributed tuplespace, thus adding another aspect of fault-tolerance.

The goal of ETS is to provide the TSpaces platform with support for fault-tolerance through replication and scalability through partitioning. The first release of Enterprise TSpaces only supports fault-tolerance using replication and not scalability using partitioning. However, the design of Enterprise TSpaces took into account that scalability through partitioning is a future goal.

References

[1] D. E. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3), 1995.

[2] D. K. G. Campbell, H. R. Osborne, and A. M. Wood. Characterising the design space for linda semantics. techreport YCS-97-277, University of York, February 1997. http://www.cs.york.ac.uk/ftpdir/reports/.

[3] N. Carriero and D. Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2), 1986.

[4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[5] S. Chiba, K. Kato, and T. Masuda. Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 416–423. IEEE Computer Society Press, June 1992.

[6] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[7] D. Gifford. Weighted voting for replicated data. In *Proceedings of Seventh Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[8] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *International Conference on Reliable Software Technologies.* Springer Verlag, 1996.

[9] K. K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, Institute for Electronic Systems, Department of Mathematics and Computer Science, Aalborg, Denmark, 1992. http://www.cs.auc.dk/research/DS/PhD/mts.abstract.html.

[10] S. Kambhatla. Replication issues for a distributed and highly available linda tuple space. Master's thesis, Oregon Graduate Institute of Science and Technology, February 1991. CS/E 91-TH-001.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 1979.

[12] J. E. Larsen and J. H. Spring. Globe - a dynamically fault-tolerant and dynamically scalable distributed tuplespace for heterogeneous, loosely coupled networks. Master's thesis, Department of Computer Science - University of Copenhagen, 1999.

[13] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from minos' labyrinth). *IEEE: International Conference on Future Trends of Distributed Computing Systems, Lisboa*, September 1993.

[14] Sun Microsystems. *JavaSpaces Specification Revision 1.0*, 1999. http://www.sun.com/jini/specs/js.ps.

[15] P. Wyckoff, T. Lehman, et al. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[16] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. In *Proceedings of the ninth International Symposium on Fault Tolerant Computing*, pages 199–206. IEEE, 1989.