Towards a Dependable Tuple Space

Alysson N. Bessani, Miguel Correia,
Joni da Silva Fraga, Lau C. Lung

DI–FCUL TR–06–04

March 2006

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Towards a Dependable Tuple Space

Alysson Neves Bessani[1], Miguel Correia[2],
Joni da Silva Fraga[1], and Lau Cheuk Lung[3]

[1] DAS, PGEEL, Universidade Federal de Santa Catarina – Brazil
[2] LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
[3] PPGIA, Pontifícia Universidade Católica do Paraná – Brazil

Abstract. The tuple space coordination model is one of the most interesting communication models for open distributed systems due to its space and time decoupling and its synchronization power. Several works have tried to improve the dependability of tuple spaces. Some have made tuple spaces fault-tolerant, using mechanisms like replication and transactions. Others have put the emphasis on security, with mechanisms like access control and cryptography. However, many practical applications in the Internet require both these dimensions. This paper describes how a set of commercial off-the-shelf (COTS) tuple spaces can be used to implement a dependable (distributed) tuple space. This tuple space is dependable in a strong sense of the word: it is secure, fault-tolerant and intrusion-tolerant, i.e. it behaves as expected even if some of the machines that implement it are successfully attacked.

1 Introduction

The **generative (aka tuple space) coordination** model, originally introduced in the LINDA programming language [12], uses a shared memory object called a **tuple space** to support coordination between distributed processes. Tuple spaces can support communication that is decoupled in time – processes do not have to be active at the same time – and space – processes do not need to know each others addresses [6]. The operations supported by a tuple space are essentially the insertion of tuples (finite sequences of values) in the space, the reading of tuples in the space and the removal of tuples from the space.

There has been some research about **fault tolerance** in the generative coordination model, both in the construction of fault-tolerant tuple spaces [27,2] and in application level fault tolerance mechanisms [15,18]. The objective of these works is essentially to guarantee that *(i.)* the service provided by the tuple space is available even if some of the servers that implement it crash and *(ii.)* the tuple space state is valid, according to the semantics of application, even when application processes crash. The main mechanism employed in providing *(i.)* is replication, while *(ii.)* is ensured usually with transactions. There has been also some research about **secure tuple spaces** (e.g. [17,5,9,26]). The goal of these works is basically to guarantee that a malicious process does not execute tuple space operations without permission, using access control mechanisms at space and tuple level.

Those works on fault tolerance and security for tuple spaces have a narrow focus in two senses: they consider only simple faults (crashes) or simple attacks (invalid accesses); and they are about either fault tolerance *or* security.

This paper goes one step further by presenting a solution for secure and fault-tolerant tuple spaces using commercial off-the-shelf (COTS) tuple spaces. The solution is based on a clear definition of what is a dependable tuple space: one that provides the dependability properties of reliability, availability, integrity and confidentiality [1], despite the occurrence of arbitrary faults, including so called Byzantine faults, like attacks and intrusions. Therefore, this work is part of an ongoing effort on designing systems that tolerate both accidental and malicious faults, an area dubbed **intrusion tolerance** [11,25].

The dependability properties are enforced using a layered architecture in which each layer is responsible for providing one or more of these properties. The resulting dependable tuple space is implemented in a set of distributed servers using the **state machine approach** [20]. This approach guarantees that the tuple space behaves according to its specification if up to a number of these servers fail, either accidentally (e.g. by crashing) or maliciously (e.g. by being attacked and starting to misbehave). Moreover, our construction also tolerates accidental and malicious faults in an unbounded number of clients that access the tuple space and provides security mechanisms for protecting itself from unauthorized clients access. The general design is presented in Figure 1.

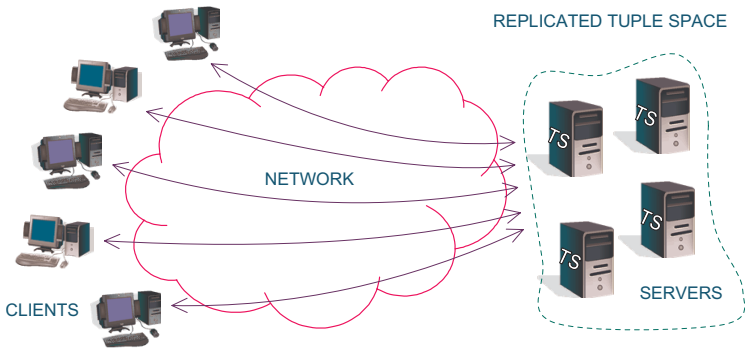

Fig. 1. Dependable tuple space based on a set of COTS tuple space servers.

The paper has two main contributions. Firstly, it presents a definition of what means for a tuple space to be dependable, in a strong sense of the word: secure, fault-tolerant and intrusion-tolerant. Secondly, it presents how a dependable tuple space can be implemented using a set of COTS tuple spaces through a non-trivial combination of security and fault tolerance mechanisms: state machine replication, space and tuple level access control, and cryptography to obtain confidentiality. To the best of our knowledge this is the first work to present a tuple space architecture that uses these mechanisms in a integrated way.

The paper is organized as follows. Section 2 presents some previous works about dependability in tuple spaces. Section 3 provides a definition of a dependable tuple space (in terms of dependability properties) and discusses the mechanisms required to fulfill this definition. The dependable tuple space architecture is presented in Section 4. Finally, Section 5 present some final remarks.

2 Related Work

There is a large collection of works aiming to improve the dependability of the tuple space coordination model through the extensive use of security or fault tolerance mechanisms. In this section we briefly review some of these efforts.

Recently, several papers have proposed the integration of security mechanisms in tuple spaces. These mechanisms are becoming more and more relevant since the generative model is shifting its use base from parallel applications for closed high-speed clusters to open heterogeneous distributed systems where security threats are common. Amongst the proposals already published, some try to enforce security policies that depend on the application [17], while others provide integrity and confidentiality through the implementation of access control at tuple space level [9], tuple level [26] or both [5]. However, none of these works consider the availability of the tuple space, which is the objective of using fault tolerance, neither propose a confidentiality scheme like the one used in this paper.

An area of work on fault tolerance in tuple spaces aims to increase the dependability of the coordination infrastructure – the tuple space – using replication. Some of these works are based on the state machine approach [20] (e.g. [2]) while others use quorum systems [16] (e.g.[27]). A more recent work proposes the use of autonomic computing techniques to select in run time the best replication scheme, aiming to improve the performance and availability [19]. However, all of these proposals are concerned with crash failures ([19] uses a crash-recover system model) and none of them regards the occurrence of malicious faults in the system, the main requirement of intrusion tolerance.

Some works focus on providing fault tolerance mechanisms at application level, providing features like transaction support [15]. These systems provide the ACID properties (Atomicity, Consistency, Isolation and Durability), i.e. they guarantee that when executing a sequence of tuple space operations, either all or none of these operations are executed. Another work proposes the mobile coordination approach [18], where an application process can send part of its code (called a coordination unit) to be executed in the tuple space server, ensuring atomicity and consistency even in case of process failure. We recognize the need for application level support for fault tolerance, however, in this paper we do not enforce any of these mechanisms since in Byzantine-prone settings they are problematic. For instance, if transactions are used, a malicious process can begin and abort transactions all the time affecting the liveness of the application. Our approach (for now) is to try to develop more elaborated algorithms that do not require this kind of application level support (like the ones presented in [3]).

Recently, we developed novel replication algorithms for the first Byzantine-resilient tuple space – BTS [4]. This work proposes an efficient Byzantine-resilient tuple space that requires Byzantine agreement only for the *inp*() operation and provides weak shared memory semantics (does not satisfy linearizability [14]). The work presented here differs from BTS in at least three important ways: it presents a clear definition of what is a dependable tuple space; it presents a complete solution for building a dependable tuple space (BTS does not include access control and does not guarantee confidentiality); and it is based on Byzantine state machine replication (BTS is based on Byzantine quorum systems [16]).

3 Defining a Dependable Tuple Space

A **tuple space** can be seen as a shared memory object that provides operations for storing and retrieving ordered data sets called **tuples**. A tuple t with all its fields defined is called an **entry**, and can be inserted in the tuple space using the $out(t)$ operation. A tuple in the space is read using the operation $rd(\bar{t})$, where $\bar{t}$ is a **template**, i.e. a special tuple in which some of the fields can be wild-cards or formal fields (e.g. $?v$, meaning the variable v will take the value read). The operation $rd(\bar{t})$ returns any tuple in the space that **matches** the template, i.e. any tuple with the same number of fields and with the field values equal to all corresponding defined values in $\bar{t}$. A tuple can be read *and* removed from the space using the $in(\bar{t})$ operation. The *in* and *rd* operations are blocking. Non-blocking versions, *inp* and *rdp*, are also usually provided.

In this paper, we provide another operation usually not considered by most tuple space works: $cas(\bar{t}, t)$ [2,22]. This operation works like an indivisible execution of the code: **if** $\neg rdp(\bar{t})$ **then** $out(t)$ ($\bar{t}$ is a template and t an entry). The operation inserts t in the space iff $rdp(\bar{t})$ does not return any tuple, i.e. iff there is no tuple that matches $\bar{t}$ currently in the space. The *cas* operation is important mainly because a tuple space that supports it is capable of solving the consensus problem [22], which is a building block for solving many important distributed synchronization problems like atomic commit, total order multicast, leader election and fault-tolerant mutual exclusion. This also means that such a tuple space is an universal shared memory object [13], i.e. it can implement any other shared memory object.

3.1 Dependability Attributes

A tuple space is dependable if it satisfies the **dependability attributes** [1]. Like in many other systems, some of these attributes do not apply or are orthogonal to the core of the design (e.g. safety and maintainability). The relevant attributes in this case are:

- **Reliability:** the operations on the tuple space have to behave according to their specification.
- **Availability:** the tuple space has to be ready to execute requested operations.

– **Integrity:** no improper alteration of the tuple space can occur.
– **Confidentiality:** the content of (some) tuple fields cannot be disclosed to unauthorized parties.

The difficulty of guaranteeing these attributes comes from the occurrence of **faults**, either due to accidental causes (e.g. a software bug that crashes a server) or malicious causes (e.g. an attacker that modifies some tuples in a server). The objective is to avoid that these faults cause the **failure** of the tuple space, i.e. that one or more of the four attributes above are violated. Malicious faults are particularly naughty since it is hard to make assumptions about what the attacker can and cannot do [25]. These faults are usually modeled in terms of the most generic class of faults – arbitrary or Byzantine faults – so the solution we propose here is quite generic in terms of the faults it handles.

The meaning of reliability and availability should be clear, but integrity and confidentiality need some discussion. An alteration of the tuple space is said to be proper (vs. improper) if and only if *(i.)* it is the outcome of one of the operations *out*, *in* or *inp*; and *(ii.)* the operation satisfies the **access policy** of the tuple space. A basic access policy must define which processes are allowed to execute which operations in a tuple space. The most simple and effective way to provide this is defining who can insert tuples in a tuple space and, for each tuple inserted, who can read and who can remove it. A more elaborated access policy is used to allow or disallow the execution of operations in the tuple space depending on three types of parameters: the identifier of the invoker; the operation and its arguments; the tuples currently in the space. An example access policy stated informally might be: *Allow only (i.) any rd/rdp operation and (ii.) any out operation invoked by clients p_1 or p_2 with the first field with a positive integer.* We call these policies **fine-grained** because they allow the use of a considerable variety of parameters [17,3]. Access policies are important to prevent unauthorized clients from removing or reading tuples from the space, and to prevent malicious clients from flooding the tuple space with large quantities of tuples with the objective of causing a denial of service [5].

This discussion also helps clarifying the meaning of the confidentiality attribute. The general idea is that the content of a tuple cannot be disclosed except for an operation that satisfies the tuple space's access policy. This policy defines who are the authorized (vs. unauthorized) parties for each operation. The idea is slightly more complicated though: confidentiality may be requested for some fields but not for others (details below).

3.2 Dependability Mechanisms

The dependability of the tuple space is enforced using a combination of several mechanisms. The most basic mechanism used to implement a dependable tuple space is **replication**: the tuple space is maintained copied in a set of n servers in such a way that the failure of some of them (that we will call **faulty**) does not impair the reliability, availability and integrity of the system. The idea is that if some of the servers fail (e.g. by crashing of by being controlled by an attacker

and behaving maliciously), the tuple space is still ready (availability) and the operations work correctly (reliability and integrity) because the correct (i.e. not faulty) replicas manage to overcome the misbehavior of the faulty replicas. There is obviously a bound on the number of replicas f that can fail, which depends on the replication solution used.

Another fundamental dependability mechanism, specially for maintaining integrity and confidentiality, is **access control**. This kind of mechanism is needed to prevent unauthorized clients from inserting (*out*) removing (*in*, *inp*) or reading tuples (*rd*, *rdp*) from the tuple space. Access control is usually implemented in replicated servers using a reference monitor implementing the same access policy in each server.

A third type of mechanism is **cryptography**, which is used to ensure the reliability of the communication and the confidentiality of the tuples. Enforcing confidentiality in a replicated tuple space is not simple for several reasons. The first is that we cannot trust the servers to guarantee the confidentiality of the tuples because f servers can fail, possibly disclosing tuples stored there. The second is the need for matching tuples. Encrypted fields in general cannot be compared with fields in a template. Some access policies can also impose limits on the fields that are encrypted. Therefore we define three kinds of fields: *public fields* whose content can be disclosed and compared; *private fields* whose content cannot be disclosed neither compared; *comparable fields* whose content cannot be disclosed but that allow simple equality comparisons.

4 Building a Dependable Tuple Space

This section presents the design of a dependable tuple space that satisfies the definition in the previous section. We begin with a model of the underlying system, then delve into the details of the implementation.

4.1 System Model

The system is composed by an infinite set of **clients** $\Pi = \{p_1, p_2, p_3, ...\}$ which interact with a set of n **servers** $U = \{s_1, s_2, ..., s_n\}$ that implement a dependable tuple space with the properties introduced in the previous section. We consider that each client and each server has an unique id.

All communication between clients and servers is made over **reliable authenticated point-to-point channels**. These channels can be implemented on common networks, including the Internet, using for instance the SSL protocol. A side effect of having this channel is that each client p_i has a shared key k_{ij} with each server s_j. This key is used in the confidentiality mechanism.

We assume that an arbitrary number of clients and a bound of up to f servers can be subject to **Byzantine failures**, i.e. they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Our architecture requires $n \geq 3f + 1$ servers to tolerate the aforementioned f faulty servers. This ratio between n and f is the best that

can be attained in practical systems (like Internet) for implementing Byzantine-resilient active replication [7]. Clients that do not follow their algorithm in some way are said to be **faulty**. A client that is not faulty is said to be **correct**. We assume **fault independence** for servers, i.e. the probability of each server failing is independent of another server being faulty. This assumption can be substantiated in practice using hardware and software diversity [10,8], therefore each server should have a different COTS tuple space.

4.2 Architecture Overview

Our dependable tuple space architecture consists in a series of integrated layers for satisfying each one of the dependability properties stated in Section 3. Figure 2 presents the dependable tuple space architecture with all its layers.

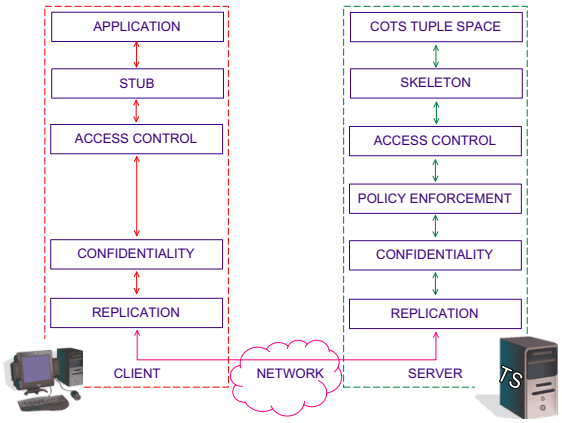

Fig. 2. Dependable tuple space architecture

On the top of the client-side stack is the application that uses the space, while on the top of the server-side stack is the COTS tuple space. The communication follows a scheme similar to remote procedure calls. The application interacts with the system by calling functions with the usual signatures of tuple space's operations: $out(t)$, $rd(\bar{t})$, ... These functions are implemented by a stub. The layer below handles tuple level access control (Section 4.5). After, there is a layer that takes care of confidentiality (Section 4.4) and then one that handles replication (Section 4.3). The server-side is similar, except that there is a new layer to check the access policy for each operation requested (policy-enforcement, Section 4.6). The skeleton (or server-side stub) calls the operations on the COTS tuple space. Since no known COTS tuple space available today supports the *cas* operation, this operation has to be implemented by the skeleton. This layer translates each call to $cas(\bar{t}, t)$ in two operations: when a request is received, first $rdp(\bar{t})$ is executed and if this operation returns $false$, $out(t)$ is executed.

4.3 Replicated Tuple Space

In Section 3 we argued that the most basic mechanism for making the tuple space dependable is replication. The idea is to replicate the tuple space in n servers, i.e. to put a COTS tuple space in each of the n servers and to guarantee some kind of consistency of these n replicas. A simple solution is to use **active replication** based on the **state machine approach** [20]. This approach guarantees linearizability [14], which is a strong form of consistency in which all replicas take the same sequence of states.

The state machine approach requires that all replicas *(i.)* start in the same state and *(ii.)* execute all requests (i.e. tuple space operations) in the same order [20]. The first point is easy to ensure, e.g. by starting the tuple space with no tuples. The second requires a fault-tolerant **total order reliable multicast** protocol, which is the crux of the problem. Some protocols of this kind that tolerate Byzantine faults are available in the literature (e.g. [7]). The state machine approach also requires that the replicas are deterministic, i.e. that the same operation executed in the same initial state generates the same final state in every replica. This implies that a read (or removal) in different servers in the same state (i.e. with the same set of tuples) must return the same response. This constrains the use of COTS tuple spaces since the original definition does not require these operations to be deterministic [12]. However, if the COTS tuple spaces used are not deterministic, determinism can be emulated using abstract specifications and abstraction functions [8].

State machine replication guarantee some of the dependability attributes we desire. The system is available if up to f servers fail; the integrity and reliability are kept if up to f servers fail because a vote is made with the results returned by the servers [20].

The protocol executed by the replication layer in the client and server sides is in Algorithm 1. The algorithm assumes the existence of a total order multicast protocol. To multicast a message m using this protocol, the algorithm calls function *TO-multicast*(U, m). On the other hand, the total order multicast protocol calls the function *TO-receive*(p, m) to receive a message m delivered by the communication network.

The client part of the replication protocol is very simple. All operations are executed by the client through a call to the *execute_op* procedure passing the operation name (OUT, RDP, INP, CAS, RD or IN) and the arguments of the operation. In this procedure, the client issues the request operation to all n servers using the total order multicast communication primitive (line 7). Then it waits for replies until some response can be extracted from the reply vector R (lines 12-13). In standard (Byzantine-tolerant) active replication protocols, a response will be accepted if it is replied by at least $f + 1$ distinct servers, thus ensuring that some correct server replied this response. However, in our system, the confidentiality layer (next section) will impose different constraints on a request result extraction.

The server side of the replication protocol is activated when the total order multicast protocol delivers a message. In the first part of the algorithm (lines

Algorithm 1 Replicated Tuple Space protocols (client p_i and server s_j).

{CLIENT}

procedure $out(t)$
1: $execute\_op(\text{OUT}, t)$

procedure $rdp(\bar{t})$
2: **return** $execute\_op(\text{RDP}, \bar{t})$

procedure $inp(\bar{t})$
3: **return** $execute\_op(\text{INP}, \bar{t})$

procedure $cas(\bar{t}, t)$
4: **return** $execute\_op(\text{CAS}, \langle \bar{t}, t \rangle)$

procedure $rd(\bar{t})$
5: **return** $execute\_op(\text{RD}, \bar{t})$

procedure $in(\bar{t})$
6: **return** $execute\_op(\text{IN}, \bar{t})$

procedure $execute\_op(op, arg)$
7: $TO\text{-}multicast(U, \langle op, \bar{t} \rangle)$
8: $R[1...n] \leftarrow \perp$
9: **repeat**
10: $receive(s_j, \langle \text{RESP}, r_s \rangle)$
11: $R[j] \leftarrow r_s$
12: $r \leftarrow extract\_response(R)$
13: **until** $r \neq \perp$
14: **return** r

{SERVER}

ts: server side upper layer

upon $TO\text{-}receive(p_i, \langle op, arg \rangle)$
15: **if** $op = \text{OUT}$ **then**
16: $t \leftarrow arg$
17: $ts.out(t)$
18: $t_r \leftarrow t$
19: **else if** $op = \text{CAS}$ **then**
20: $\bar{t} \leftarrow arg[0]$
21: $t \leftarrow arg[1]$
22: $t_r \leftarrow ts.cas(\bar{t}, t)$
23: **else if** $op = \text{IN} \vee op = \text{INP}$ **then**
24: $t_r \leftarrow ts.inp(arg)$
25: **else if** $op = \text{RD} \vee op = \text{RDP}$ **then**
26: $t_r \leftarrow ts.rdp(arg)$
27: **end if**
28: **if** $t_r = \text{NM} \wedge (op = \text{IN} \vee op = \text{RD})$ **then**
29: $add\_blocked(op, p_i, \bar{t})$
30: **else**
31: $send(p_i, \langle \text{RESP}, t_r \rangle)$
32: **end if**
33: **if** $op = \text{OUT} \vee (op = \text{CAS} \wedge t_r = t)$ **then**
34: $notify\_blocked(t)$
35: **end if**

15-27) the required operation is executed by the server in its upper layer ts. This call will eventually reach the COTS tuple space skeleton of this replica (see Figure 2). Notice that the blocking operations of the COTS tuple space are never called, to avoid blocking the server. When a blocking operation is requested, its non-blocking version is executed (lines 23-27) and if it is not successful because there is no matching tuple ($t_r = \text{NM}$) the operation is stored in a set of pending operations for future execution (lines 28-29). If the requested operation is executed, a response is sent back to the client (line 31). Finally, if some tuple is inserted in ts, the procedure *notify_blocked* checks if some pending operation can be executed (lines 33-35).

4.4 Adding Confidentiality

Replication is often seen not as a helper but as an impediment for confidentiality. The reason is easy to understand: if secret information is stored not in one but in several servers, it probably becomes easier for an attacker to get it, not harder. Therefore, the enforcement of confidentiality in a replicated tuple space is not trivial. Several solutions that come to mind simply do not work or are

unacceptable for the generative coordination model. One solution would be for each client to encrypt the tuples it inserts, but it would require the distribution of a decryption key to any client that might read or remove the tuple. Another solution would be for the servers to encrypt the tuples, but a malicious server would be capable of disclosing any tuple. In fact, all previously proposed tuple space confidentiality mechanisms [26,5] are useless in our system model since they assume a trusted tuple space.

The solution we propose follows in some way the idea of letting the servers handle the confidentiality. However, instead of trusting each server to keep the confidentiality of the tuple fields, we trust a *set* of servers. The solution is based on a (n, k)–**publicly verifiable secret sharing** scheme (PVSS) [21], with $k = f + 1$. Each server s_i has a private key x_i and a public key y_i. The clients know the public keys of all servers. Clients play the role of the dealer of the scheme, encrypting tuple fields with the public keys of each server and obtaining a set of field **shares** (function share). Any tuple field can be decrypted with $k = f + 1$ shares (function combine), therefore a collusion of malicious servers cannot disclose the contents of confidential tuple fields (we assume at most f servers can be faulty).

The confidentiality scheme has also to handle the problem of matching (possibly encrypted) tuples with templates. When a client calls $out(t)$ it chooses one of three types of protection for each tuple field:

- **public**: the field is not encrypted so it can be compared arbitrarily (e.g. if it belongs to a range of values) but its content may be disclosed[4];
- **comparable**: the field is encrypted but a cryptographic *hash* of the field is also stored so equality comparisons are possible (details below);
- **private**: the field is encrypted and no hash is stored so its content cannot be disclosed and no comparisons are possible.

A **collision-resistant hash function** $\mathsf{H}(v)$ (e.g. SHA-256) maps an arbitrarily length input to a fixed length output, satisfying two properties: it is computationally infeasible to find two values $v \neq v'$ such that $\mathsf{H}(v) = \mathsf{H}(v')$ (collision resistance); given an output it is computationally infeasible to find an input that produces that output (unidirectionality). We informally call the output of such function *a hash*.

The idea behind *comparable fields* is the following. Suppose client p_1 wants to insert in the space a tuple t with a single comparable field f_1. p_1 sends t encrypted and $\mathsf{H}(f_1)$ to the servers. Suppose later a client p_2 calls, without loss of generality, $rd(\bar{t})$ and the tuple space needs to check if t and $\bar{t}$ match. p_2 calculates $\mathsf{H}(\overline{f}_1)$ and sends it to tuple space that verifies if this hash match $\mathsf{H}(f_1)$. This scheme works for equalities but clearly does not work with more complex comparisons like inequalities. The scheme has another limitation. Although hash functions are unidirectional, if the range of values that a field can take is known and limited, then a brute-force attack can disclose its content. Suppose a field takes 32 bit values. An attacker can simply calculate the hashes of all 2^{32} possible values to

[4] It may be needed, for example, in fine-grained policy enforcement (Section 4.6).

discover the hashed value. This limitation is a motivation for not using typed fields in a dependable tuple spaces. Also, the limitation of comparable fields is the reason why we also define *private fields*: no hash is sent to the servers so comparisons are impossible but their content cannot be disclosed.

Algorithm 2 Confidentiality Layers (client p_i and server s_j).

{CLIENT}

ts: client side replication layer

procedure $out(t)$
1: $ts.out(mask(t))$

procedure $rdp(\bar{t})$
2: **return** $ts.rdp(fingerprint(\bar{t}))$

procedure $inp(\bar{t})$
3: **return** $ts.inp(fingerprint(\bar{t}))$

procedure $cas(\bar{t}, t)$
4: $\bar{t}' \leftarrow fingerprint(\bar{t})$
5: $t' \leftarrow mask(t)$
6: **return** $ts.cas(\bar{t}', t')$

procedure $rd(\bar{t})$
7: **return** $ts.rd(fingerprint(\bar{t}))$

procedure $in(\bar{t})$
8: **return** $ts.in(fingerprint(\bar{t}))$

procedure $mask(t)$
9: $t_h \leftarrow fingerprint(t)$
10: $\langle s_1, ..., s_n, PROOF_t \rangle \leftarrow \mathsf{share}(t)$
11: $s \leftarrow \langle \mathsf{E}(s_1, k_{i1}), ..., \mathsf{E}(s_n, k_{in}) \rangle$
12: **return** $t_h.s.PROOF_t$

procedure $fingerprint(t = \langle f_1, ..., f_m \rangle)$
13: **for all** $1 \leq i \leq m$ **do**
14: $f'_i \leftarrow \begin{cases} * & \text{if } f_i = * \\ f_i & \text{if } f_i \text{ is public or formal} \\ \mathsf{H}(f_i) & \text{if } f_i \text{ is comparable} \\ \text{PR} & \text{if } f_i \text{ is private} \end{cases}$
15: **end for**
16: **return** $\langle f'_1, ..., f'_m \rangle$

{SERVER}

ts: server side upper layer

procedure $out(t)$
17: $ts.out(unmask(t))$

procedure $rdp(\bar{t})$
18: **return** $ts.rdp(\bar{t}.*.*)$

procedure $inp(\bar{t})$
19: **return** $ts.inp(\bar{t}.*.*))$

procedure $cas(\bar{t}, t)$
20: **return** $ts.cas(\bar{t}.*.*, unmask(t))$

procedure $unmask(t_h.s.PROOF_t)$
21: $s_i \leftarrow \mathsf{D}(s[j], k_{ij})$
22: $PROOF_i \leftarrow \mathsf{prove}(s, PROOF_t)$
23: **return** $t_h.s_i.PROOF_i$

This confidentiality scheme is implemented by the confidentiality layers in client and server sides. The algorithms for these layers are presented in Algorithm 2. Both the client-side and the server-side layers have one procedure to process each tuple space operation. In the client (viz. server), these procedures are called by the layer above (viz. below) the confidentiality layer. In the client (viz. server), the confidentiality layer calls the layer below (viz. above) using a pseudo-object ts (see the algorithm, top/left).

The confidentiality layers simply make transformations to the tuples and templates that are sent and received. In the client side, templates used to read or remove a tuple are transformed in a **fingerprint** (lines 2-8). The idea is to transform each tuple field according to its kind (see lines 13-15). Private fields are simply substituted by a constant PR since no match is possible. Also in the client side, tuples to be inserted using *out* or *cas* are **masked**, i.e. the content of

each of its fields is protected from disclosure depending on its kind (lines 9-12). This procedure starts by obtaining a fingerprint of the tuple (line 9), then uses the PVSS scheme to obtain shares of the tuple (line 10) and encrypts each share with the secret key shared by the client and each of the servers (line 11). The PVSS scheme also generates a proof $PROOF_t$ that the shares are really shares of the tuple (line 10). The encrypted shares and $PROOF_t$ are concatenated ('.' operator) with the fingerprint of the tuple and passed to replication layer.

In the server side, tuples to be inserted are unmasked (lines 17, 20-23), i.e. the tuple share corresponding to the server is decrypted with the key shared with the client (line 21), a proof that the share is correct is generated (line 22), and these two informations are concatenated with the tuple fingerprint (line 23). Templates are simply passed to the upper layer concatenated with two wild-card fields ('*') with the objective of reading some tuple that matches the template fingerprint whatever its share and proof (lines 18-20). Blocking operations do not appear because the server-side replication layer never calls them.

The replication algorithm in the previous section, used an *extract_response* procedure to obtain the result of an operation (line 12 of Algorithm 1). We did not explain the functioning of the procedure in that section since it was related to the confidentiality scheme. Now, the functioning of the procedure should be clear. The procedure verifies if the shares already received are correct; if $f+1$ are, then it combines the shares and obtains the original tuple. There is a possibility that the tuple obtained does not satisfy the template used to read/remove it: the client that introduced it might be malicious and have introduced a tuple with hashes of the comparable fields that were not really hashes of those fields. When that happens, if the client only read the tuple, it has to send the tuple and the proofs to the servers (using total order multicast) so that they discard the tuple. Then, the operation has to be repeated.

4.5 Access Control

Access control has been shown to be a fundamental security mechanism for tuple spaces [5]. For controlling tuple and space access we do not choose any specific access control model to allow the use of different implementations in different applications. For instance, access control lists (ACLs) might be used for closed systems, but some type of role-based access control (RBAC) might be more suited for open collaborative systems [24]. To accommodate these different mechanisms in our dependability architecture, access control mechanisms are defined in terms of credentials: each tuple space TS has a set of required credentials C^{TS} and each tuple t has two sets of required credentials C^t_{rd} and C^t_{in}. To insert a tuple in a tuple space TS, a client must provide credentials that match C^{TS}. Analogously, to read (resp. remove) a tuple t from a space, a client must provide credentials that match C^t_{rd} (resp. C^t_{in}).

The credentials needed for inserting a tuple in TS are defined by the administrator that configures the tuple space. If ACLs are used, we can have, e.g. $C^{TS} = \{p_1, p_2\}$, i.e., C^{TS} is the set of processes allowed to insert tuples in

TS. Analogously, with ACLs the credentials C^t_{rd} and C^t_{in} associated with tuple t would be the sets of processes allowed to read and remove the tuple, respectively.

The implementation of access control in our architecture is done in the access control layers in the clients and servers, as show in Figure 2. In the client side, when an *out* or a *cas* operation are invoked, the associated credentials C^t_{rd} and C^t_{in} are appended to the tuple as two public fields. Additionally, the client credentials are appended as another public field of the tuple, either if the tuple is a entry (*out* or *cas*) or a template (*cas* and read/remove operations).

In the server side layer, each (correct) server tests if the operation can be executed. If the operation is a tuple insertion, say t, client credentials (appended to t) must be sufficient for inserting t in the space. If this condition holds, t is inserted in the space (with its associated required credentials C^t_{rd} and C^t_{in} as public fields). If the requested operation is a read/remove, the credentials associated with the template $\bar{t}$ passed, must be sufficient for executing the operation. In practice the server side layer may have to read several tuples from the COTS tuple space to find one that can be read/removed by the client. This reading of several tuples may be supported by the COTS tuple space (e.g. the *scan* operation in TSpaces [23]) or may have to be implemented in the skeleton. If several tuples are read, they have to be checked in the same deterministic order in all servers, in order to guarantee that all correct servers choose the same.

Notice that a faulty server can return a tuple even if a client has no permission to read it. However, the confidentiality scheme does not allow a tuple returned by less than $f + 1$ distinct servers to be read, so malicious servers cannot reveal a tuple to unauthorized clients. A faulty server can also store a tuple inserted by a client that has no permission to do it. This tuple will not be read by correct clients for the same reason.

4.6 Policy Enforcement

In a recent paper we have shown that tuple spaces that implement a more generic form of access control – policy-enforced tuple spaces – have interesting properties for the coordination of clients that can fail in a Byzantine (or arbitrary) way [3]. For instance, we have provided two universal constructions, which allow the implementation of *any* shared-memory object using a tuple space. Here we describe how policy enforcement is implemented in our tuple space in the policy enforcement layer (see Figure 2).

The idea behind **policy enforcement** is that the tuple space is governed by a **fine-grained access policy**. This kind of policies take into account the already mentioned three types of parameters: identifier of the invoker; operation and arguments; tuples currently in the space. The application of these ideas for securing tuple spaces was first proposed in [17].

An example policy is in Figure 3. Operations without rules are prohibited so this policy allows only *rd* and *out* operations. The rule for *rd* states that any *rd* invoked is executed. The rule for *out* states that only tuples with two values can be inserted and only if there is no tuple in the space with the same second value as the one that the invoker is trying to insert.

Tuple Space state TS
R_{rd}: $execute(rd(t))$:- $invoke(p, rd(t))$
R_{out}: $execute(out(\langle a, b\rangle))$:- $invoke(p, out(\langle a, b\rangle)) \wedge \forall c : \langle a, c\rangle \notin TS$

Fig. 3. Example access policy.

A tuple space has a single access policy. It can be defined in several ways, e.g. during the system setup by the system administrator. Whenever an operation invocation is received in a server, there is a verification if the operation satisfies the access policy of the space in the policy enforcement layer. The verification itself is a simple local computation of a condition expressed in the rule of the operation invoked. When an operation is rejected the server returns an error code to the invoker. The client only accepts the rejection if it receives $f + 1$ copies of the same error code.

5 Final Remarks

In this paper, we presented an integrated architecture for an intrusion-tolerant tuple space. This construction based on COTS tuple spaces satisfies a set of important dependability attributes: reliability, integrity, availability and confidentiality.

As future work, we plan to investigate the support for transactions in our dependable tuple space. This might be an interesting mechanism for supporting some applications, although the problem is far from trivial when malicious faults are possible. We also plan to implement the dependable tuple space and assess its performance, possibly making a comparison with quorum-based solutions.

References

1. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Mar. 2004.
2. D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programing in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, Mar. 1995.
3. A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. Submitted for publication, 2006.
4. A. N. Bessani, J. da Silva Fraga, and L. C. Lung. BTS: A Byzantine fault-tolerant tuple space. In *Proc. of the 21st ACM Symposium on Applied Computing*, Apr. 2006. to appear.
5. N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. SecSpaces: a data-driven coordination model for environments open to untrusted agents. In *Electronic Notes in Theoretical Computer Science*, volume 68, 2003.
6. G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, Feb. 2000.
7. M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

8. M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, 2003.
9. R. De Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.
10. Y. Deswarte, K. Kanoun, and J. C. Laprie. Diversity against accidental and deliberate faults. In *Computer Security, Dependability, & Assurance: From Needs to Solutions*. IEEE Press, 1998.
11. J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd Int. Conference on Computer Security*, pages 203–218, 1985.
12. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.
13. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, Jan. 1991.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.
15. K. Jeong and D. Shasha. PLinda 2.0: A transactional checkpointing approach to fault tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems.*, pages 96–105, Dana Point, CA, USA, Oct. 1994.
16. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
17. N. H. Minsky, Y. M. Minsky, and V. Ungureanu. Making tuple-spaces safe for heterogeneous distributed systems. In *Proc. of the 15th ACM Symposium on Applied Computing*, pages 218–226, Mar. 2000.
18. A. Rowstron. Using mobile code to provide fault tolerance in tuple space based coordination languages. *Science of Computer Programming*, 46(1–2):137–162, Jan. 2003.
19. G. Russelo, M. Chaudron, and M. van Steen. Dynamically adapting tuple replication for managing availability in a shared data space. In *Coordination Model and Languages - COORDINATION'05)*, volume 3454 of *LNCS*, Apr. 2005.
20. F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
21. B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Advances in Cryptology - CRYPTO'99*, volume 1666 of *LNCS*, pages 148–164, Aug. 2004.
22. E. J. Segall. Resilient distributed objects: Basic results and applications to shared spaces. In *Proc. of the 7th Symposium on Parallel and Distributed Processing, SPDP'95*, pages 320–327, Oct. 1995.
23. T. J. Lehman et al. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, Mar. 2001.
24. W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Computing Surveys*, 37(1):29–41, Mar. 2005.
25. P. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
26. J. Vitek, C. Bryce, and M. Oriol. Coordination processes with Secure Spaces. *Science of Computer Programming*, 46(1-2):163–193, Jan. 2003.
27. A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proc. of the 19th Symposium on Fault-Tolerant Computing - FTCS'89*, pages 199–206, June 1989.