Design and Implementation of an Intrusion-Tolerant Tuple Space

Alysson Neves Bessani*† Eduardo Pelison Alchieri† Joni da Silva Fraga† Lau Cheuk Lung§

† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil

§ Prog. de Pós-Grad. em Informática Aplicada, 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 while others have focused on security. However, many practical applications in the Internet require both these dimensions. This paper describes the design and implementation of DEPSPACE, a dependable communication infrastructure based on the tuple space coordination model. DEPSPACE 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. Moreover, it is a policy-enforced augmented tuple space, a shared memory object that we have recently proven to be universal, i.e., capable of implementing any other shared memory object.

1 Introduction

The *generative* (or *tuple space*) *coordination* model, originally introduced in the LINDA programming language [8], relies on 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 locations or addresses [5], providing some level of synchronization at the same time. The operations supported by a tuple space are essentially the insertion, reading and removal of tuples, i.e., of finite sequences of values.

Previous works on fault-tolerant (e.g. [17, 2]) and secure tuple spaces (e.g. [10, 4]) have a narrow focus in two senses: they consider only simple faults (crashes) or simple attacks (invalid access); and they are about *either* fault tolerance *or* security. The present paper goes one step further by investigating the implementation of *secure and fault-tolerant tuple spaces*. The solution is inspired on a current trend in dependability that applies fault tolerance concepts and mechanisms in the domain of security, *intrusion tolerance* [7, 16]. The proposed tuple space is not centralized but implemented by a set of tuple space servers. This set of tuple spaces forms a tuple space that is *dependable*, meaning that it enforces the attributes of reliability, availability, integrity and confidentiality [1], despite the occurrence of arbitrary faults, like attacks and intrusions in some servers.

*Corresponding author. Email: *neves@das.ufsc.br*

The implementation of a dependable tuple space with the above-mentioned attributes presents some interesting challenges. Our design is based on the classical *state machine replication* approach [13, 6]. However, this approach does not guarantee the confidentiality of the data stored in the servers; quite on the contrary, replicating data in several servers is usually considered to reduce the confidentiality since the potential attacker has more servers where to attempt to read the data, instead of just one. Therefore, combining the state machine approach with confidentiality is a non-trivial challenge that has to be addressed. A second challenge is intrinsically related to the tuple space model. Tuple spaces resemble associative memories: when a process wants to read a tuple, it provides a template and the tuple space returns a tuple that "matches" the template. This match operation involves comparing data in the tuple with data in the template, but how can this comparison be possible if tuples are encrypted to guarantee confidentiality? In this paper we present DEPSPACE, a system that addresses these challenges using a particular kind of secret sharing scheme together with cryptographic hash functions in such a way that it guarantees that a tuple stored in the system will have its content revealed only to authorized parties.

The design of a dependable tuple space is not a merely academic exercise. The tuple space system presented in this paper might be useful in several practical application domains, like the following. *(i.) Ad hoc networks* are an important current trend in computer science. It has been shown that tuple spaces can be a powerful solution to coordinate activities in those environments, and systems like LIME [12] already explore this paradigm. *(ii.) Mobile agents* are programs that migrate from node to node in the network, usually to gather data or to perform computations close to the data source. The interaction of these agents is usually complex due to the lack of fixed location. Tuple spaces are an obvious solution to support this communication since they provide time and space decoupling [5]. *(iii.) Grid computing* involves using resources in large numbers of computers to perform complex computations. These computations are decoupled both in space and time so a tuple space would be a good solution to coordinate the tasks performed.

Algorithms based on a tuple space with the properties of DEPSPACE are well suited for coordination of non-trusted processes in practical dynamic systems. Instead of trying to compute some distributed coordination task using a complete dynamic model (like, for instance, the one proposed in [11]), we pursue a more pragmatic approach where a tuple space is deployed on a fixed and small set of servers and is used by a unknown, dynamic and unreliable set of processes that need

to coordinate themselves. An example of scenario were this kind of system can be deployed are peer-to-peer systems and infrastructured wireless networks.

The paper has two main contributions. The first is the presentation of the dependable and intrusion-tolerant tuple space. This design involves a non-trivial combination of security and fault tolerance mechanisms: state machine replication, space and tuple level access control, and cryptography. To the best of our knowledge this is the first work to implement Byzantine state machine replication for tuple spaces and to integrate this technique with a confidentiality scheme. The second contribution is the first practical assessment of the performance of an intrusion-tolerant scheme that provides data confidentiality even when there are intrusions in some of the servers. We are not aware of any other practical assessment of such a scheme in the literature.

2 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. 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 [8].

The tuple space implemented in this paper provide another operation usually not considered by most tuple space works: $cas(\bar{t}, t)$ (conditional atomic swap) [2, 15, 3]. This operation works like an indivisible execution of the code: **if** $\neg rdp(\bar{t})$ **then** $out(t)$. The operation inserts t in the space iff $rdp(\bar{t})$ does not return any tuple, i.e. if there is no tuple in the space that matches $\bar{t}$. The cas operation is important mainly because a tuple space that supports it is capable of solving the consensus problem [15], 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.

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 the operations requested), *integrity* (no improper alteration of the tuple space can occur.), and *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). Since it is difficult to model the behavior of a malicious adversary, intrusion tolerant systems mostly assumes the most generic class of faults – arbitrary or Byzantine faults – so the solution we propose and describe in the next section is quite generic in terms of the faults it handles.

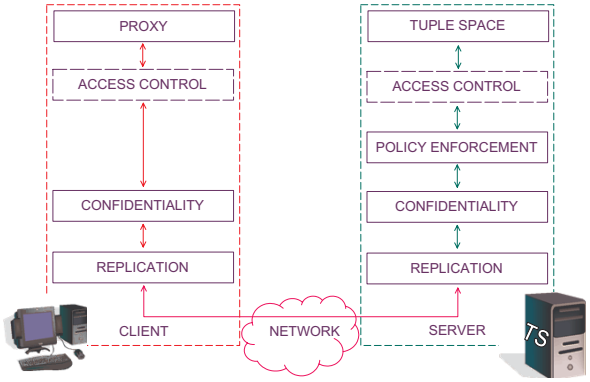

Figure 1: DEPSPACE architecture

3 Building a Dependable Tuple Space

This section presents the design of DEPSPACE. We begin with a model of the underlying system and the basic assumptions of our design, then delve into the general layered architecture and finally into each layer.

3.1 Underlying Assumptions

The system is composed by an infinite set of *clients* which interact with a set of n *servers* that implement the dependable tuple space with the properties introduced in the previous section. At most f servers and an unbounded number of clients can suffer Byzantine failures, i.e. they can deviate arbitrarily from their specification. We assume *fault independence* for servers, i.e. the failures of the servers are uncorrelated. This assumption can be substantiated in practice using diversity.

All communication between clients and servers is made over *reliable authenticated point-to-point channels*. These channels can be implemented using TCP and some cryptographic mechanism such as MACs (Message Authentication Codes) with session keys.

The dependable tuple space does not require any explicit time assumption, however, since it is based on the *state machine replication* model [13], it requires a total order multicast primitive. We implement this primitive using the BYZANTINE PAXOS protocol [6, 18], which only ensures liveness if the system eventually becomes synchronous.

3.2 DEPSPACE Architecture Overview

The architecture of the dependable tuple space consists in a series of integrated layers that enforce each one of the dependability attributes stated in Section 2. Figure 1 presents the DEPSPACE architecture with all its layers.

On the top of the client-side stack is the proxy layer, which provides access to the replicated tuple space, while on the top of the server-side stack is the tuple space implementation (a local 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 spaces' operations: $out(t)$, $rd(\bar{t})$, ... These functions are called on the proxy. The layer below handles tuple level

access control. After, there is a layer that takes care of confidentiality (Section 3.4) and then one that handles replication (Section 3.3). The server-side is similar, except that there is a new layer to check the access policy for each operation requested. Access control and policy-enforcement are not described in this paper due to space constraints. Some aspects of these mechanisms are described in [3].

3.3 Replication

The most basic mechanism used in DEPSPACE is *replication*: the tuple space is maintained in a set of n servers in such a way that the failure of up to f of them does not impair the reliability, availability and integrity of the system. The idea is that if some servers fail, the tuple space is still ready (availability) and the operations work correctly (reliability and integrity) because the correct replicas manage to overcome the misbehavior of the faulty replicas. A simple approach for replication is *state machine replication* [13]. This approach guarantees *linearizability* [9], which is a strong form of consistency in which all replicas appear to 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 [13]. 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 multicast* protocol, which is the crux of the problem. 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.

The protocol for replication is very simple: the client send an operation request using total order multicast and wait for $f + 1$ replies with the same response from different servers. Since each server receives the same set of messages in the same order (due to the total order multicast), and the tuple space is deterministic, there will be always at least $n - f \geq 2f + 1$ correct servers that execute the operation and return the same reply.

3.4 Confidentiality

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 of those solutions would be to encrypt the client-server communication and let the tuple space encrypt the tuple fields with its own key(s). This is unacceptable because we assume f servers can fail maliciously, so they might decrypt the tuple fields and disclose their contents. A second solution would be to let the client that inserts a tuple to encrypt the tuple fields either with a secret key (with a symmetric cryptography algorithm like AES) or with its private key (with a public-key algorithm like RSA). The problem of this solution is that it requires all clients that might read and/or remove this tuple to know the decryption key. This contradicts the anonymity property of the generative coordination model [8], which states that clients should not need to know information about each other.

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 $(n, f+1)$*–publicly verifiable secret sharing* scheme (PVSS) [14]. Clients play the role of the dealer of the scheme, encrypting the tuple with the public keys of each server and obtaining a set of tuple *shares*. Any tuple can be decrypted with $f + 1$ shares, therefore a collusion of malicious servers cannot disclose the contents of confidential tuple fields. A server can build a proof that the share that it is giving to the client is correct. The PVSS scheme also provides two verification functions, one for each server to verify the share it received from the dealer and other for the client to verify if the shares collected from servers are not corrupted.

The confidentiality scheme has also to handle the problem of matching (possibly encrypted) tuples with templates. To solve this problem we use a *collision-resistant hash function* $\mathsf{H}(v)$ (e.g. SHA-1) that maps an arbitrarily length input to a fixed length output (called a *a hash*).

The idea is to use the hashes of the fields of a tuple as a fingerprint of the tuple, and execute the matching of tuples using the hashes of the fields of the tuple, instead of the their values. The fingerprint of a tuple is stored in each server together with its tuple share. One limitation of this scheme is that although hash functions are unidirectional, if the range of values hat a field can take is known and limited, then a brute-force attack can disclose its content. This limitation is a motivation for not using typed fields in a dependable tuple spaces. Using fingerprints and the PVSS scheme, the procedures for providing confidentiality for tuple spaces on top of state machine replication is the following:

Tuple insertion. All shares are sent encrypted together with the fingerprint of the tuple and its validity proof by the client using total order multicast. The encryption of each share s_i addressed to server p_i is made through symmetric cryptography, using the session key shared between the client and the server p_i. Notice that all servers will receive all encrypted shares, however, each server will have access only to its corresponding share, the fingerprint of the tuple and the proof generated by the PVSS algorithm. These three pieces of data are stored in the tuple space.

Tuple access. To access a tuple, the client sends the fingerprint of the template and then waits for the replies from the servers containing the same tuple fingerprint that matches the template fingerprint sent, the encrypted share of the server for this tuple and its corresponding proof of validity (produced by the server). Each share is encrypted by the servers with the session key shared between the client and the server to avoid eavesdropping on the replies. Additionally, the replies from the servers can be signed to make the client capable of cleaning invalid tuples from the space (see bellow). The client decrypts the received shares, verifies their validity, and combines $f + 1$ of them to obtain the stored tuple.

Recovery procedure. Notice that nothing prevents a malicious client to insert a tuple with a fingerprint that does not correspond to it. Consequently, after obtained a stored tuple,

the client has to verify if the tuple corresponds to the fingerprint. If such correspondence does not exist, the client must clear the tuple from the space (if it is not removed yet) and reissue its operation to the space. The "tuple cleaning" is made in two steps: *(1.)* the client sends all replies received to the servers to prove that the stored tuple is invalid; and *(2.)* the servers verify if the replies are produced by the servers and, if the tuple returned does not correspond to the fingerprint, this tuple is removed from the local tuple space. Moreover, the client that inserted the invalid tuple can be put on a black list (and its further requests ignored). This ensures that a malicious client cannot insert tuples after some of its invalid insertions have been cleaned.

A key advantage of the confidentiality scheme of DEPSPACE is that most of the cryptographic processing is done at client side. This improves the scalability of the system, as will be show in Section 5.

An interesting point of our scheme is that the confidentiality layer weakens our tuple space semantics since it no longer satisfies linearizability in all situations: a malicious client can insert invalid shares in some servers and valid shares in others, so it is not possible to ensure that the same read/remove operation executed in the same state of the tuple space will have the same result: the result depends of the $n-f$ responses collected. However, DEPSPACE satisfies linearizability for all tuples that have been inserted by correct processes.

4 DEPSPACE Implementation

The DEPSPACE was implemented using the Java programming language, and at present it is a simple but fully functional dependable tuple space. The Byzantine-resilient state machine replication algorithm implemented is the PAXOS AT WAR described in [18], combined with a total ordering scheme inspired by the one defined by [6][1]. Authentication was implemented using the SHA-1 algorithm for producing HMACs (providing an approximation for authenticated channels on top of Java TCP Sockets). SHA-1 was also used for computing hashes. For symmetric cryptography we employed the Triple DES algorithm while RSA with exponents of 1024 bits was used for digital signatures. All the cryptographic primitives used in the prototype were provided by the default provider of version 1.5 of JCE (Java Cryptography Extensions). The only exception was the PVSS scheme, which we implemented following the specification in [14], using algebraic groups of 192 bits.

Two main implementation optimizations are specially relevant for the system performance. The first is to try to execute rdp and rd first without total order multicast and wait for $n - f$ responses. If all of them are equal, the returned value is the result of the operation, otherwise the normal protocol operation must be executed. The second optimization is for servers to send read replies without signing them. The clients must explicitly request signed responses for an operation if they find that the read tuple is invalid. This improves the latency of the read operations since the processing cost for asymetric cryptography is still very high. Since it is expected that invalid tuples will be very rare, in most cases the read operations will not require digital signatures.

5 Experimental Evaluation

This section presents an experimental evaluation of DEPSPACE. The execution environment was composed by a set of five Athlon 2.4GHz PCs with 512 Mb of memory and running Linux (kernel 2.6.12). They were connected by a 100Mbps switched Ethernet network. The Java runtime environment used was Sun's JDK 1.5.0_06.

We considered tuples with 4 fields and sizes equal to 64, 256, and 1024 bytes, running on a system with 4 servers[2]. Two cases were considered in all experiments: the complete system (with confidentiality) and the system with the confidentiality scheme deactivated. All our experiments considered fault-free executions. Figure 2 present the results.

The first experiments (Figures 2(a) to 2(d)) measured the delay perceived by the client for each one of the tuple space non-blocking operations. The client was in one of the machines and the servers in the other four. We executed each operation 1000 times and obtained the mean time discarding the 5% values with greater variance.

The results presented in the figure show that out, inp and cas have almost the same latency when the confidentiality layer is not used – the solid lines in Figures 2(a) to 2(d) This is the latency imposed by the total order multicast protocol (about 6 ms). rdp, on the other hand, is much more efficient (about 2 ms) due to the optimization presented in Section 4, which avoids running the total order multicast protocol.

The dotted lines in the graphs show the latency of the protocols when the confidentiality layer is used. In these experiments all tuples inserted and read have all their fields comparable. In fact, the number of comparable fields is not relevant since the overhead for producing a hash is negligible when compared to the overhead of the PVSS scheme. The cas operation (Figure 2(d)) has two dotted lines, one measuring the cases where a tuple is inserted and other for the cases when some tuple is read. The additional latency cost caused by the confidentiality scheme is mostly due to the client-side processing of the operations. The global cost of the confidentiality scheme is also higher for out since this is the only operation in which the shares and their proofs have to be generated. Notice that the processing cost of the cas operation when a tuple is read is approximately the cost of out plus the cost of rdp. This reflects the fact that this operation executes both tuple insertion and access procedures.

From the Figure 2, it is clear that the size of the tuple has almost no effect on the latency experienced by the protocols. This happens due to two implementation features: *(i.)* our BYZANTINE PAXOS implementation makes agreement over message hashes; and *(ii.)* the secret shared in the PVSS scheme is not the tuple, but a symmetric key used to encrypt the tuple. *(i.)* implies that it is not the entire message that it ordered by the PAXOS protocol, but only its hash (MD5

[1] Our algorithm is an extension to PAXOS AT WAR to provide total order multicast. It differs from BFT [6] since we assume reliable channels instead of using checkpoints.

[2] We do not present experiments with more servers due to space constraints. However we could say that our system suffers the same scalability problems of other protocols with message complexity $O(n^2)$.

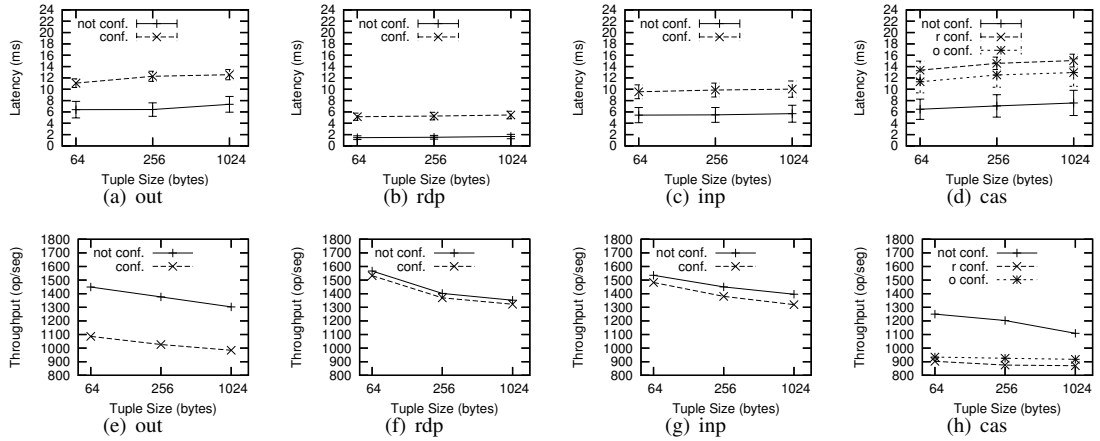

Figure 2: Latency and throughput of DEPSPACE operations considering different tuple sizes.

hashes always have 128 bits), consequently the message size has little effect over the agreement protocol. With feature *(ii.)* we can execute all the required PVSS cryptographic in the same, relatively small algebraic field of 192 bits, which means that the tuple size has no effect in these computations and the use of the confidentiality scheme implies almost the same overhead regardless the size of the tuple.

The second set of experiments measured the throughput of DEPSPACE. For these experiments we used a modified client process that pre-processes C requests for the operation of interest (executing the client-side processing) and send then one-by-one to the servers. We measured the time T taken to process all these requests at the leader replica, from the moment it receives the first request to the moment it sends the response for the last one. The throughput of the system is calculated as C/T.

Figures 2(e) to 2(h) show that the system provides a high throughput with few servers. Even with larger tuples, the decrease in throughput is reasonable small, e.g. increasing the tuple size 16 times (64 to 1024 bytes) causes a decrease of about 10% in the system throughput. Therefore, the good throughput of the system is due to the low processing required at server side and the batch message ordering implemented in BYZANTINE PAXOS [6].

6 Final Remarks

The paper presents a solution for the implementation of an intrusion-tolerant tuple space. The proposed architecture integrates several dependability and security mechanisms in order to enforce the required properties. This architecture was implemented in a system called DEPSPACE.

Another interesting aspect of this work is the integration of replication with confidentiality. To the best of our knowledge, this is the first paper to integrate state machine replication and confidentiality of data stored in the servers. Somewhat surprisingly, this integration is not trivial and the use of secret sharing fundamentally weakens the semantics of state machine replication in a Byzantine-prone environment (linearizability is not unconditionally ensured).

All code used in DEPSPACE is available at the *JITT (Java Intrusion Tolerance Tools)* project homepage: `http://www.das.ufsc.br/~neves/jitt`.

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. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, July 2006.

[4] N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. SecSpaces: a data-driven coordination model for environments open to untrusted agents. *Elect. Notes in Theoretical Computer Science*, 68(3), 2003.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2), 2000.

[6] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4), 2002.

[7] J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, 1985.

[8] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.

[9] 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.

[10] 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 - SAC 2000*, 2000.

[11] A. Mostefaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. E. Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems - SRDS 2005*, pages 109–118, Oct. 2005.

[12] A. Murphy, G. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3), 2006.

[13] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[14] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164, Aug. 1999.

[15] E. J. Segall. Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, pages 320–327, Oct. 1995.

[16] P. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[17] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89*, pages 199–206, June 1989.

[18] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, June 2004.