DepSpace: A Byzantine Fault-Tolerant Coordination Service

Alysson Neves Bessani[†], Eduardo Pelison Alchieri[‡],
Miguel Correia[†], Joni da Silva Fraga[‡]

[†]LaSIGE, University of Lisbon, Lisbon, Portugal
[‡]DAS, Federal University of Santa Catarina, Florianópolis, Brazil

ABSTRACT

The tuple space coordination model is one of the most interesting coordination 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 through the use of replication for fault tolerance and access control for security. However, many practical applications in the Internet require both fault tolerance and security. This paper describes the design and implementation of DepSpace, a Byzantine fault-tolerant coordination service that provides a tuple space abstraction. The service offered by DepSpace is secure, reliable and available as long as less than a third of service replicas are faulty. Moreover, the content-addressable confidentiality scheme developed for DepSpace bridges the gap between Byzantine fault-tolerant replication and confidentiality of replicated data and can be used in other systems that store critical data.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*; D.4.5 [**Software**]: Operating Systems—*Reliability*; D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Algorithms, Design, Reliability, Security

Keywords

Byzantine Fault Tolerance, Confidentiality, Tuple Space

1. INTRODUCTION

Open distributed systems are a fundamental component of our Information Society. These systems are typically composed by an unknown number of processes running in heterogeneous hosts connected by heterogeneous networks like the Internet. Albeit many distributed applications are still programmed using simple primitives like TCP/IP sockets and remote procedure calls, there is an important demand for more powerful tools that allow the design of complex applications with low time-to-market. This requirement is more stringent due to the need of guaranteeing tolerance to disconnections, recuperation from server crashes and security against malicious actions.

The *tuple space coordination* model, originally introduced in the Linda programming language [22], 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 [11], providing some level of synchronization at the same time. The tuple space is a kind of storage that stores *tuples*, i.e., finite sequences of values. The operations supported are essentially inserting a tuple in the space, reading a tuple from the space and removing a tuple from the space. The programming model supported by tuple spaces is regarded as simple, expressive and elegant, being supported by middleware platforms like Sun's JavaSpaces [39] and IBM's TSpaces [30].

There has been some research on fault-tolerant [5, 43] and secure tuple spaces [9, 33, 41], but these works 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 systems dependability that applies fault tolerance concepts and mechanisms in the domain of security, *intrusion tolerance* (or *Byzantine fault tolerance*) [21, 14, 40]. 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 [4], despite the occurrence of Byzantine faults, like attacks and intrusions in some servers.

The implementation of a dependable tuple space with the above-mentioned attributes presents some interesting challenges. Our design is based on the *state machine approach*, a classical solution for implementing Byzantine fault-tolerant systems [35]. 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,

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

instead of just one. Therefore, combining the state machine approach with confidentiality is a 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 we want to guarantee confidentiality and the way to guarantee this property is by encrypting data? In this paper we present DepSpace, a dependable tuple space system that addresses these challenges using a secret sharing scheme together with standard cryptographic primitives in such a way that it guarantees that a tuple stored in the system will have its content revealed only to authorized parties. The proposed confidentiality scheme for content-based storage developed for DepSpace is novel and interesting by itself since it bridges the gap between confidentiality of replicated data and Byzantine fault-tolerant (BFT) replication and can be used in other systems that require confidentiality.

Algorithms and services based on a dependable tuple space like DepSpace are well suited for coordination of non-trusted processes in practical dynamic systems. Instead of trying to compute some distributed coordination task considering a dynamic distributed system, we pursue a more pragmatic approach where a tuple space is deployed on a fixed and small set of servers and is used by an unknown, dynamic and unreliable set of processes that need to coordinate themselves. An example of scenario where this kind of system can be deployed are peer-to-peer systems and infrastructured wireless networks.

This paper has three main contributions. The first is the design and implementation of a dependable intrusion-tolerant tuple space[1]. This design involves a non-trivial systematic combination of security and fault tolerance mechanisms: BFT state machine replication, space and tuple level access control, and cryptography. The second contribution is the development of a content-aware confidentiality scheme for replicated data that can be combined with any BFT replication protocols to ensure that critical data stored in a BFT service will not be revealed as long as at most f out of n servers are compromised. This is the first work to integrate BFT state machine replication with a confidentiality scheme. The last contribution is the development of several generic services – lock service, partial barrier, secret storage and name service – with DepSpace, showing the simplicity of developing distributed systems abstractions using the tuple space model.

1.1 Why a dependable tuple space?

Since our objective with DepSpace is to provide a coordination infrastructure for dynamic and untrusted distributed systems, the attentive reader may ask: Why someone would use a service like a tuple space for coordinating processes, instead of using a fault-tolerant synchronization library that implement consensus and other distributed algorithms? A recent paper about Google's Chubby lock service lists several reasons to have a coordination service instead of a distributed algorithm implemented in a client library [8]. Some of these reasons and others specific for BFT systems are:

1. *Ease of use:* in many ways, using a service with a well defined interface is easier than integrating a library with a system. The use of a service makes easier to convert programs that initially are not fault-tolerant into dependable ones [8].
2. *Less resources:* encapsulating the replication and synchronization requirements in a service, namely, the algorithm's required number of replicas and/or specialized components, makes the applications that use the service simpler and less expensive in terms of resources used. Moreover, if the service is shared by several applications, its cost is also shared.
3. *Ease of Management:* the "Achilles' heel" of Byzantine fault tolerance is the assumption of fault independence (or the non existence of correlated failures in the system) and how to cover this assumption in a real system [34]. Encapsulating the core of the synchronization protocols in a small set of diverse, well-protected and well-managed servers that implement the coordination service makes it easier to achieve fault independence for critical services than assuming that each and every process of an open system fails independently one from another.

At this point, another question may be asked: Why a tuple space instead of some other more popular abstraction? In this paper we argue that the tuple space abstraction is adequate for dealing with any coordination task required in distributed systems due to its simplicity and generality. At least three theoretical results support this statement. First, the set of operations supported by the tuple space model is known to be a *Turing powerful language* [10], which means that any sequential program can be expressed using them. Second, if augmented with a special operation (conditional atomic swap, *cas*), the tuple space is an *universal shared memory object* [26, 37], i.e., it has synchronization power to solve the important consensus problem between any number of processes and, consequently, to emulate any other shared memory object or synchronization primitive [26]. Third, it was shown that in an untrusted and fault-prone environment in which processes are subject to Byzantine failures, an augmented tuple space with a policy-enforcement mechanism called PEATS (Policy-Enforced Augmented Tuple Space), from which DepSpace is an instantiation, can be orders of magnitude more efficient than other abstractions like read/write storage and sticky bits protected by access control lists [6]. Besides this theoretical expressiveness and efficiency, from a practical point of view there are three other motivations for using tuple spaces: *simplicity* as only four basic operations (and some variants) are supported; *content-addressable*, i.e., the fact that tuples are accessed by their contents gives a high flexibility to the model; *decoupled communication*, the time and space decoupling of the tuple space model are extremely attractive for dynamic distributed systems.

2. 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 in which all fields have a defined value is called an *entry*. A tuple with one or more undefined fields is called a *template* (usually denoted by a bar, e.g.,

[1]Freely available at the DepSpace project homepage: `http://www.navigators.di.fc.ul.pt/software/depspace`.

$\bar{t}$). An undefined field is represented by a *wild-card* ('*'). Templates are used to allow content-addressable access to tuples in the tuple space. An entry t and a template $\bar{t}$ *match* if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. For example, template $\langle 1, 2, * \rangle$ matches any tuple with three fields in which 1 and 2 are the values of the first and second fields, respectively. A tuple t can be inserted in the tuple space using the $out(t)$ operation. The operation $rd(\bar{t})$ is used to read tuples from the space, and returns any tuple in the space that *matches* the template $\bar{t}$. 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 [22].

There are two other extensions provided by modern tuple space implementations [23, 39, 30]: *multiread operations*, versions of *rd*, *in*, *rdp* and *inp* in which all (or a given maximum number of) tuples that match a given template are returned; and *tuple leases*, a validity time for inserted tuples such that the tuple is removed from the space after this time.

This paper presents the implementation of a policy-enforced augmented tuple space (PEATS) [6] so we provide another operation usually not considered by most tuple space works: $cas(\bar{t}, t)$ (conditional atomic swap) [5, 37]. 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., if there is no tuple in the space that matches $\bar{t}$[2]. The *cas* operation is important mainly because a tuple space that supports it is capable of solving the consensus problem [37], which is a building block for solving many important distributed synchronization problems like atomic commit, total order multicast and leader election. Table 1 presents a summary of the tuple space operations supported by DepSpace.

A tuple space is dependable if it satisfies the *dependability attributes* [4]. 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 tuple fields can not 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 assumes the most generic class of faults – arbitrary or Byzantine faults. In this scenario, a tuple space built using BFT state machine replication (e.g., [1, 14, 18, 28]) together with a simple access control scheme can provide all dependability attributes except confidentiality. Therefore, most research in DepSpace was related to the problem of maintaining tuple confidentiality in presence of faulty servers.

[2]Notice that the meaning of the tuple space *cas* operations is the opposite of the well known register *compare&swap* operation, where the object state is modified if its current state is *equal* to the value compared.

Oper.	Description
$out(t)$	inserts tuple t in the space
$rdp(\bar{t})$	reads a tuple that matches $\bar{t}$ from the space (returning *true*); returns *false* if no tuple is found
$inp(\bar{t})$	reads and removes a tuple that matches $\bar{t}$ from the space (returning *true*); returns *false* if no tuple is found
$rd(\bar{t})$	reads a tuple that matches $\bar{t}$ from the space; stays blocked until some matching tuple is found
$in(\bar{t})$	reads and removes a tuple that matches $\bar{t}$ from the space; stays blocked until some matching tuple is found
$cas(\bar{t}, t)$	if there is no tuple that matches $\bar{t}$ on the space, inserts t and returns *true*; otherwise returns *false*

Table 1: DepSpace supported operations.

3. SYSTEM MODEL

The system is composed by an unlimited set of *clients* which interact with a set of n *servers* that implement a dependable tuple space with the properties described in the previous section. We consider that each client and each server have an unique identifier (id).

All communication between clients and servers is made over *reliable authenticated point-to-point channels*. These channels can be implemented using TCP sockets and message authentication codes (MACs) with session keys under the common assumption that the network can drop, corrupt and delay messages, but can not disrupt communication between correct processes indefinitely.

The dependable tuple space does not require any explicit time assumption. However, since we use a total order multicast primitive based on the Byzantine Paxos consensus protocol [14, 32, 45] to ensure that all replicas execute the same sequence of operations, an *eventually synchronous system* model [19] is required for liveness.

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. We assume *fault independence* for servers, i.e., that the failures of the servers are uncorrelated. This assumption can be substantiated in practice using several types of diversity [34].

4. ARCHITECTING DEPSPACE

The architecture of the dependable tuple space consists in a series of integrated layers that enforce each one of the dependability attributes listed 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

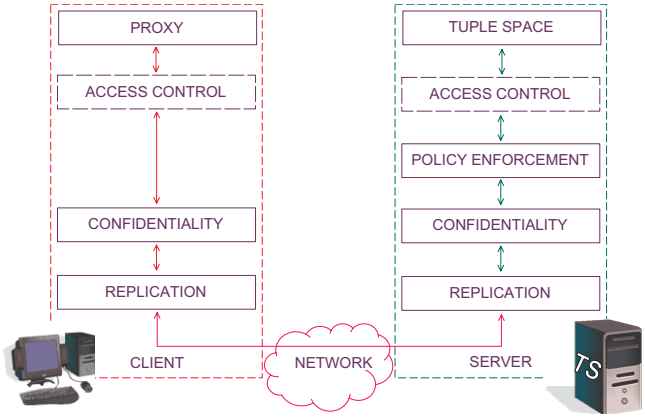


Figure 1: DepSpace architecture

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 (Section 4.3). After, there is a layer that takes care of confidentiality (Section 4.2) and then one that handles replication (Section 4.1). The server-side is similar, except that there is a new layer to check the access policy for each operation requested (Section 4.4).

We must remark that not all of these layers must be used in every tuple space configuration. The idea is that the layers are added or removed according to the quality of service desired for the tuple space.

4.1 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 ones. A simple approach for replication is *state machine replication* [35]. This approach guarantees *linearizability* [27], 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 in the same order [35]. 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 sends an operation request using total order multicast and waits 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.

4.2 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. Any solution that requires key sharing between clients contradicts the anonymity property of the tuple space model [22], which states that communicating processes do not need to know each other. Since we assume Byzantine failures, no server individually can have access to the tuples, so the solution must rely on a set of servers.

The basic tool we use to implement confidentiality for DepSpace is a special kind of *secret sharing scheme* [38]. In a secret sharing scheme, a special party called dealer distributes a secret to n players, but each player gets only a share of this secret. The basic property of the scheme is that it is needed at least $f + 1 \leq n$ different shares of a secret to recover it and no information about the secret is disclosed with f or less shares. More specifically, our solution is based on a $(n, f+1)$–*publicly verifiable secret sharing* scheme (PVSS) [36]. Each server 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 the tuple with the public keys of each server and obtaining a set of tuple *shares* (function share of the PVSS scheme). Any tuple can be decrypted with $f + 1$ shares (function combine), therefore a collusion of malicious servers can not disclose the contents of confidential tuple fields (we assume at most f servers can be faulty). A server can use a function prove to 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 (function verifyD) and another for the client/combiner to verify if the shares collected from servers are not corrupted (function verifyS).

The confidentiality scheme has also to handle the problem of matching encrypted tuples with templates. When a client inserts a tuple in the space, it chooses one of three types of protection for each tuple field:

- *public*: the field is not encrypted so it can be compared arbitrarily but its content may be disclosed if a server is faulty;
- *comparable*: the field f_i is encrypted but a cryptographic *hash* of the field obtained with a *collision-resistant hash function* $\mathsf{H}(f_i)$ is also stored;
- *private*: the field is encrypted and no hash is stored so no comparisons are possible.

The type of protection for each field of a tuple is defined in a *protection type vector*. Given a tuple t, we define its protection type vector v_t as a sequence of protection types, one for each field of t. The possible values for the fields of a protection type vector are PU, CO and PR, indicating if the corresponding tuple field is public, comparable or private, respectively. Therefore, each field of v_t describes the protection type required by the corresponding field of t. For example, if a tuple $t = \langle 7, 8 \rangle$ has a protection type vector $v_t = \langle \mathrm{CO}, \mathrm{PR} \rangle$, we known that the first field is comparable and that the second field is private.

The idea behind *comparable fields* is to allows tuple matching without disclosing tuple contents. For example, suppose

client c_1 wants to insert in the space a tuple t with a single comparable field f_1. c_1 sends t encrypted and $\mathsf{H}(f_1)$ to the servers. Suppose later a client c_2 requests $rd(\overline{t})$ and the tuple space needs to check if t and $\overline{t}$ match. c_2 calculates $\mathsf{H}(\overline{f}_1)$ and sends it to the tuple space that verifies if this hash is equal to $\mathsf{H}(f_1)$. This scheme works for equalities but clearly does not work with more complex comparisons. 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 8 bit values. An attacker can simply calculate the hashes of all 2^8 possible values to 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 can not be disclosed.

4.2.1 Protocol

The confidentiality scheme is implemented by the confidentiality layers in the client and server sides. Before presenting the algorithms to insert and retrieve tuples, we have to define the concept of *fingerprint*. Given a tuple $t = \langle f_1, ..., f_m \rangle$ and a protection type vector $v_t = \langle p_1, ..., p_m \rangle$, the fingerprint $t_h = \langle h_1, ..., h_m \rangle$ of t is calculated calling function $\mathsf{fingerprint}(t, v_t) = t_h$. This function calculates each field of the fingerprint using:

$$h_i = \begin{cases} * & \text{if } f_i = * \\ f_i & \text{if } v_i = \text{PU} \\ \mathsf{H}(f_i) & \text{if } v_i = \text{CO} \\ \text{PR} & \text{if } v_i = \text{PR} \end{cases}$$

The fingerprint function ensures that if a tuple t matches a template $\overline{t}$, the fingerprint t_h of t matches the fingerprint $\overline{t}_h$ of $\overline{t}$ if both are generated using the same protection type vector v_t. Consequently, there should be a vector v_t that must be known (and used) by all clients that insert and read certain kinds of tuple.

Besides the PVSS and the hash function, the confidentiality scheme uses a *symmetric cryptography scheme* and a *digital signature scheme*. The cryptography scheme provides two functions $\mathsf{E}(k, v)$ (encrypt) and $\mathsf{D}(k, v')$ (decrypt). The signature scheme includes a signing function and a verification function that use pairs of public and private keys. We assume that each correct server has a private key know only by itself, and that its public key is known by all client processes and servers. We represent a message signed by a server i with a subscript σ_i.

A fundamental idea of the confidentiality scheme is that, on the contrary to a straightforward implementation of a dependable tuple space using BFT state machine replication, the replicas *do not* have the same state since each server stores the fingerprint of the tuple, a share of the tuple and the public data generated by the PVSS scheme. This data set is called the *tuple data* and our scheme ensures that all DepSpace replicas have *equivalent states*, i.e., that for each tuple inserted in the tuple space, each correct replica stores some tuple data in its local tuple space. This is ensured using the total order multicast provided by the replication layer.

Tuple insertion. Algorithm 1 presents the procedure for storing a tuple with confidentiality. All shares are sent encrypted together with the tuple fingerprint and its validity proof by the client using total order multicast. The encryption of each share t_i addressed to server i is made through symmetric cryptography, using the session key shared between the client c and the server i. Notice that all servers will receive all encrypted shares, but each server will have access only to its share, which is the only one it has a shared key to decrypt.

Algorithm 1 Storing tuple t (client c and server i)

{*Client*}

C1. Client c generates n shares $t_1, ..., t_n$ of t and the proof of correctness $PROOF_t$ using the function $\mathsf{share}(y_1, ..., y_n, t)$.

C2. c computes the fingerprint t_h of the tuple t using the protection vector v_t calling $\mathsf{fingerprint}(t, v_t)$.

C3. c encrypts each share t_i generating the encrypted share t'_i with the symmetric shared key $k_{c,i}$ between client c and replica i using $\mathsf{E}(k_{c,i}, t_i) = t'_i$.

C4. c executes the procedure for total order multicast the message $\langle \text{STORE}, t'_1, ..., t'_n, t_h, PROOF_t \rangle$ to the n servers.

C5. c waits for acknowledgements from $f + 1$ different servers.

{*Server*}

S1. When a server i receives the message $\langle \text{STORE}, t'_1, ..., t'_n, t_h, PROOF_t \rangle$ from client c, it first retrieves its share executing $\mathsf{D}(k_{c,i}, t'_i) = t_i$.

S2. i calculates $PROOF^i_t = \mathsf{prove}(t_i, x_i, PROOF_t)$ and store the tuple data $\langle t_i, t_h, PROOF_t, PROOF^i_t, c \rangle$.

S3. i sends an acknowledge to c.

Tuple reading/removal. Algorithm 2 presents the procedure for reading a tuple. The same algorithm is executed for removing the tuple, with the difference that the chosen tuple data is removed in step S1. To access a tuple, the client sends the fingerprint of the template and then waits for encrypted replies from the servers containing the same tuple fingerprint that matches the template fingerprint sent, the share of the server for this tuple and its corresponding proof of validity (produced by the server). Each reply 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 below). The client decrypts the received shares, verifies their validity, and combines $f + 1$ of them to obtain the stored tuple.

Repair procedure. Nothing prevents a malicious client from inserting a tuple with a fingerprint that does not correspond to it. Consequently, as show in step C5 of Algorithm 2, after obtaining a stored tuple, the client has to verify if the tuple corresponds to the fingerprint. If it does not correspond, the client must clear the tuple from the space (if it is not removed yet) and reissue the operation to the space. This repair procedure is described in Algorithm 3. The cleaning of the tuple 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.)* if the replies were produced by the servers and the tuple returned does not correspond to the fingerprint, the servers remove the tuple from their local tu-

Algorithm 2 Reading template $\overline{t}$ (client c and server i)

{*Client*}

C1. c computes the fingerprint $\overline{t}_h$ of the template $\overline{t}$ using the protection type vector v_t executing $\mathsf{fingerprint}(\overline{t}, v_t)$.

C2. c executes the total order multicast primitive to send a message $\langle \text{READ}, \overline{t}_h \rangle$ to the servers.

C3. c waits for a set containing at least $f + 1$ valid replies from different servers. A reply m_i from server i is valid if $\mathsf{D}(k_{c,i}, m_i) = \langle \text{TUPLE}, t_h, PROOF_t, t_i, PROOF_t^i \rangle_{\sigma_i}$ with the same t_h and $PROOF_t$ fields and, for each reply from a server i, $\mathsf{verifyS}(t_i, y_i, PROOF_t, PROOF_t^i)$ is *true*.

C4. c combines the $f + 1$ correct shares received calling $\mathsf{combine}(t_1, ..., t_{f+1})$ and obtains the tuple t.

C5. c verifies if $t_h = \mathsf{fingerprint}(t, v_t)$. If *true*, the operation is finished, otherwise the repair procedure is called and the operation is repeated.

{*Server*}

S1. When receiving $\langle \text{READ}, \overline{t}_h \rangle$ from client c, a server i *chooses deterministically* some tuple data $\langle t_i, t_h, PROOF_t, PROOF_t^s, c' \rangle$ such that t_h *matches* $\overline{t}_h$.

S2. i sends the signed and encrypted message $\mathsf{E}(k_{c,i}, \langle \text{TUPLE}, t_h, PROOF_t, t_i, PROOF_t^i \rangle_{\sigma_i})$ to c and stores the id of the client c' that inserted the tuple read by c together with the hash of the tuple fingerprint in $last\_tuple[c]$.

ple space. Furthermore, the client that inserted the invalid tuple is put on a black list and its further requests ignored. This ensures that a malicious client can not insert tuples after some of its invalid insertions have been cleaned.

Algorithm 3 Tuple space repair (client c and server i)

{*Client*}

C1. The set S of signed TUPLE messages that the shares are used to build the tuple t (the invalid tuple) are used to justify the tuple space repairing. c calls the total order multicast primitive to send the message $\langle \text{REPAIR}, S \rangle$ to all servers.

{*Server*}

S1. When server i receives $\langle \text{REPAIR}, S \rangle$ from client c it verifies if the repair is justified: *(i.)* all messages in S are correctly signed; *(ii.)* all messages in S have the same t_h and $PROOF_t$; and *(iii.)* the fingerprint of the tuple t which was built from the shares from the messages in S is different from t_h.

S2. If the tuple data corresponding to the invalid tuple is still present in the tuple space of server i, it is deleted.

S3. Finally, server i adds the process that inserted the invalid tuple in the space (it's id was stored together with the in $last\_tuple[c]$) in a black list and its future requests are ignored.

4.2.2 Discussion

There are several comments that deserve to be made concerning the proposed confidentiality scheme.

Linearizability. The confidentiality scheme weakens the semantics of BFT state machine replication since it no longer satisfies linearizability. The problem is that a malicious client writing a data item can insert invalid shares in some servers and valid shares in others, so it is not possible to ensure that the same read operation executed in the same state of the system will have the same result depending on the $f + 1$ responses collected on step C3 of Algorithm 2, a client can access a tuple or call the repair procedure. However, the confidentiality scheme ensures that this condition is satisfied for all data items that have been inserted by correct processes.

Lazy recovery. The repair procedure uses a blacklist to ensure that a malicious client can not write data items after some of their invalid writings have been recovered. This means that the damage caused by malicious clients is *recoverable* and *bounded*. A question one can ask is that if it is possible to integrate the fingerprint calculation inside the PVSS scheme in such a way that the servers can verify if the share delivered to them corresponds to the data item that originated the fingerprint without revealing the whole data item to them. We investigated this question and the answer is *no* for the general case. The PVSS scheme requires a symmetric cryptography algorithm in order to be used to share arbitrary secrets (like tuples) [36]. This fact, together with the hash function used to compute fingerprints and required by the PVSS scheme means that, in order for such verification to be executed in the servers, it would be needed that functions share (PVSS shares' generation function), E (encryption function) and H (hash function) has some mathematical relation such as $\mathsf{share}(v) = \mathsf{E}(v) + \mathsf{H}(v)$. Clearly, there is no such relationship for arbitrary functions share, E and H. We do not know if there is some set of cryptographic functions with these properties but, even if there are, we believe that a lazy and recover-oriented approach like the one used in our scheme is efficient in the common case (no malicious faults) since the recover is executed only when data items written by malicious processes are read from the system (once per invalid data item).

Generality of the scheme. For space constraints and to maintain the focus of the paper, we chose to present the confidentiality scheme in terms of tuples and tuple spaces. However, the presented scheme can be used (with little modification) in any content-based BFT data store built above a BFT replication protocol. The key modification that must be made to implement this scheme in other data stores concerns the fingerprint function. This function is used to generate the "address information" of data items and the "query information" used for accessing these items. In adapting the scheme for other data stores, this function must be changed to ensure that a query can find a data item that matches its condition.

4.3 Access Control

Access control ia a fundamental security mechanism for tuple spaces [9]. We do not make a decision about the specific access control model to be used, since the better one depends on the application. 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 systems. 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 TS, a client must provide credentials that match C^{TS}. Analogously, to read (resp. remove) 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, C^{TS} is the set of processes allowed to insert tuples in TS and the C^t_{rd} and C^t_{in} associated with t would be the sets of processes allowed to read and remove the tuple, respectively.

In our architecture, the access control is implemented by the access control layers in the clients and servers (Figure 1). 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. In the server side, it is verified if the operation can be executed. If the operation is the insertion of a tuple t, the client credentials appended to t must be sufficient for inserting t in the space. If the requested operation is a read/remove, the credentials associated with the template $\bar{t}$ passed must be sufficient for executing the operation.

4.4 Policy Enforcement

The idea behind *policy enforcement* is that the tuple space is governed by a *fine-grained access policy* [6]. This kind of policies takes into account three types of parameters to decide if an operation is approved or denied: identifier of the invoker; operation and arguments; and the tuples currently in the space.

A tuple space has a single access policy. It should be defined 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. Correct servers correctly verify the access policy, while faulty servers can behave arbitrarily. The verification itself is a simple local evaluation of a logical 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 accepts the rejection if it receives $f+1$ copies of the same error code.

4.5 Properties and Correctness

In this section we present a sketch of the main safety properties satisfied by our design. The liveness of the architecture is satisfied as long as the total order multicast is live.

The DepSpace architecture ensures four safety properties. First, the replication layer ensures that every tuple space operation issued by a correct client is executed in accordance with the formal linearizable tuple space specification [7]. This happens because the total order multicast ensures that all operations are executed in the same order by all replicas, and, together with the determinism of the tuple space, makes all replicas execute the same state transitions. The second property is due to the confidentiality and access control layers and can be summarized as following: given a tuple t inserted in a tuple space with required access credentials C^t_{rd} and C^t_{in}, it is infeasible for an adversary that does not have credentials that satisfy C^t_{rd} or C^t_{in} to disclose the contents of comparable or private fields as long as at most f servers are faulty. This property holds mainly because the PVSS scheme requires $f+1$ tuple shares to reconstruct a tuple. Since we assume at most f faulty servers, no correct server will send the tuple share to a client that did not present credentials that match the required credentials for accessing the tuple. The third safety property concerns the amount of visible damage that a faulty client can do is recoverable and bounded. By visible damage we mean the insertion of an invalid tuple (i.e., a tuple that can not be decrypted or that is different from the fingerprint) that is read by a correct client. Visible damage is recoverable because, independently of the problem of the invalid tuple inserted by the malicious client, a correct server only stores the tuple data associated with it if it is delivered by the total order multicast protocol (Algorithm 1, steps S1 and S2). This implies that all correct servers will store some tuple data for the tuple. When this tuple is read by a correct client (i.e., it is passed a template fingerprint that matches the tuple fingerprint), the client executes the verification of tuple consistency (Algorithm 2, steps C5), and when it fails, invokes the repair procedure. A repair issued by a correct client will be accepted by correct servers that will remove the tuple from the space (if it was not removed) and the client that inserted this tuple (stored in the *last_tuple* variable) is put in a black list, and has its subsequent requests ignored. This fact proves also that the visible damage is bounded: after the tuple space is repaired from an invalid tuple, the faulty client that inserted this tuple can not insert more invalid tuples on the system. The final property is that every operation is executed on a given tuple space if and only if this operation is in accordance with the policy defined for this space. This property is satisfied because the policy enforcement layer approves or denies the operation in each correct replica and since the state of the servers are equivalent, this layer will give the same result on each correct replica.

4.6 Optimizations

This section presents a set of optimizations that can be applied to the basic DepSpace protocols to improve the performance of the system.

Read-only operations. An optimization of the *rd*/*rdp* operations that can be made in the replication layer is to try to execute them first without total order multicast and wait for $n-f$ responses. If all responses are equal (or equivalent, in case the confidentiality layer is used), the returned value is the result of the operation, otherwise the normal protocol must be executed. This optimization is quite effective when there is no faulty servers in the system.

Avoiding verification of shares. One of the most costly steps of the confidentiality scheme is the share verification operation executed by clients when a tuple is read/removed (Algorithm 2, step C3). If the tuple is inserted by a correct client and there are no faulty servers, the first $f+1$ replies received by a reading client will contain correct shares that suffice to generate the original tuple. So, we change the algorithm to make the client first try to combine the shares without verifying them and, if the recovered secret is not the tuple corresponding to the stored fingerprint, the normal algorithm is used (i.e., the shares are verified and only $f+1$ correct shares are used to recover the tuple). This optimization drastically reduces the cryptographic processing time required for the client read a tuple, in the fault-free case.

Laziness in share extraction/proof generation. When a tuple is inserted, the servers execute steps S1-S2 of Algorithm 1 to decrypt their share and generate the corresponding proofs. However, Section 6 shows that tuple insertion involves the most expensive cryptographic operations. In order to alleviate this processing, we delay this operation until the tuple is first read.

Signatures in tuple reading. Algorithm 2 shows that the responses for read operations have to be signed because these messages can be used as justifications in the repair procedure. Since the processing cost of asymmetric cryptography is high, instead of signing all responses, the servers can first send them without signatures and the clients request signed responses if they find that the read tuple is invalid. Since it is expected that invalid tuples will be rare, in most cases digital signatures will not be used.

5. DEPSPACE IMPLEMENTATION

DepSpace was implemented in the Java programming language, and at present it is a simple but fully functional dependable tuple space. The system has about 12K lines of code, most of them on the replication (5K) and confidentiality (2.5K) layers. Here we present some interesting points about its implementation.

Replication protocol. The BFT total order multicast algorithm implemented is based on the Paxos at War protocol [45] that makes consensus in two communication steps as long as there are no faulty servers in the system, which is optimal [32]. The protocol is modified to work as a total order multicast following ideas of PBFT [14]. There are two important differences between our protocol and PBFT, though: (1.) it does not use checkpoints but works under the assumption of authenticated reliable communication (checkpoints can be implemented to deal with cases where these channels are disrupted); and (2.) it does not employ authenticators (i.e., MAC vectors) [14] in its critical path but only MACs, resulting in much less cryptographic processing in synchronous executions with correct leaders (from $3+8f$ [28] to 4 MACs per consensus executed on the bottleneck server). Two common optimizations are implemented in the total order protocol: agreement over hashes (the consensus protocol is executed using message hashes instead of full messages) and batch agreement (a consensus protocol is executed to order a batch of messages, instead of just one).

Cryptography. Authentication was implemented using the SHA-1 algorithm for producing HMACs (providing an approximation of authenticated channels on top of Java TCP Sockets). SHA-1 was also used for computing the hashes. For symmetric cryptography we employed the 3DES 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 [36], using algebraic groups of 192 bits (more than the 160 bits recommended). This implementation makes extensive use of the `BigInteger` class, provided by the Java API, which provides several utility methods for implementing cryptography.

Tuples and tuple space. All client-side (resp. server side) layers in DepSpace provide the same interface to the layer above (resp. below). Tuples are represented by a common tuple class and all fields are represented as generic objects to simplify the design (fields are not typed). DepSpace supports multiple logical tuple spaces with different configurations. Therefore, it provides an administrative interface for creating, destroying and managing logical tuple spaces.

Serialization. During DepSpace's development we discovered that the default Java serialization mechanism was very inefficient, specially when the confidentiality layer was enabled. To overcome this, we implemented serialization methods for all classes that have to be serialized using the `java.io.Externalizable` Java interface, decreasing all message sizes significatively. For example, the STORE message (step C4 of Algorithm 1) for a 64-byte tuple with four comparable fields has 2313 bytes when transformed using standard Java serialization. The main problem is that the `BigInteger` class breaks 192 bits numbers in many fields, which makes the serialized version of this class much bigger than 24 bytes. Our custom serialization only stores 24 bytes for each big integer, and thus improved the serialized form of the described tuple from 2313 to 1300 bytes.

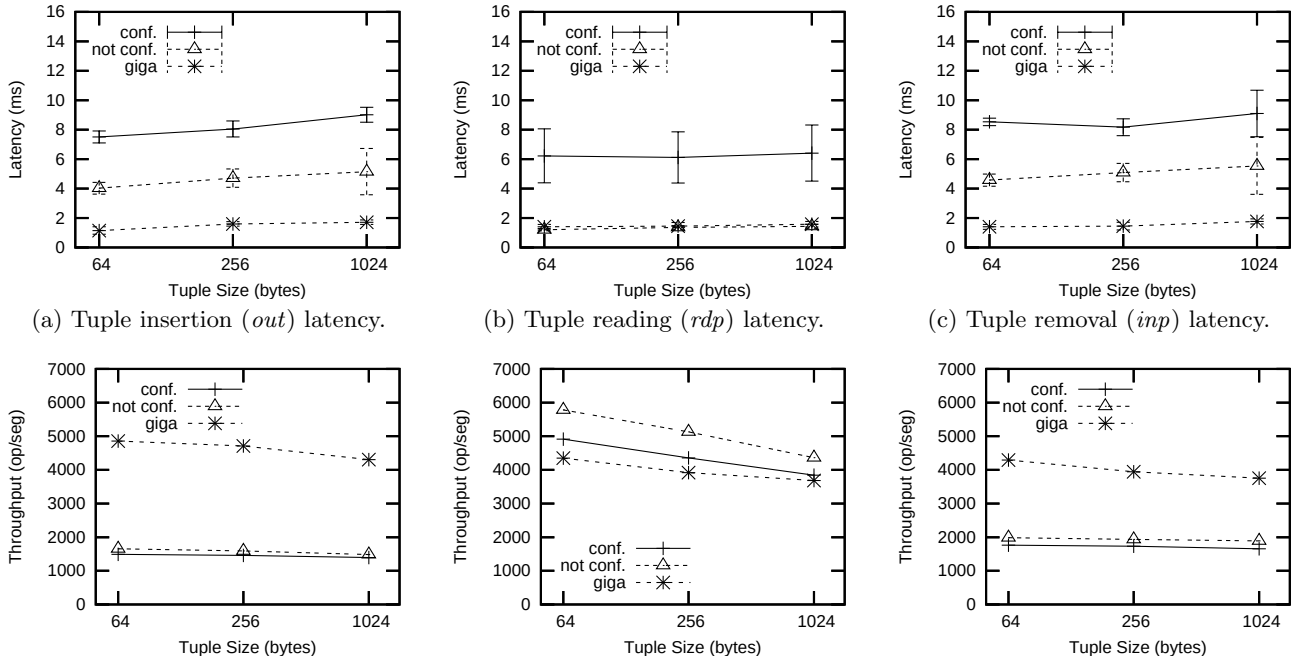
Access control. Our current implementation of the access control layer is based on ACLs. We assume that each client has an unique id (a 32 bit integer) that is obtained by a server when the client establishes an authenticated channel with it. When a logical space is created, the administrator creates an ACL with the ids of the clients that can insert tuples. When a tuple is inserted, it takes two ACLs that define the clients that can read and remove it.

Policy enforcement. The DepSpace implementation requires that the policy of the tuple space is passed as an argument when the space is created. Currently, the system accepts policies specified in the Groovy programming language [17]. A policy is defined in a Groovy class and then passed as a string to the servers. At a server, the string is transformed in a binary Java class and instantiated as a policy enforcer. Every time an operation is requested to a server, an authorization method is called on the policy enforcer. If the method returns *true*, the operation is executed in the upper layer, otherwise access is denied. Notice that after the script being processed during the tuple space creation, its execution involves a normal Java method invocation, i.e., no script interpretation is made. To ensure that no malicious code will be executed by the script[3], the class loader used to instantiate the policy enforcer is protected by a security manager that only gives it the right to read the contents of the tuple space. No I/O and no external calls can be executed by the script.

6. EXPERIMENTAL EVALUATION

We executed a set of experiments to assess the performance of DepSpace under different conditions. We used Emulab [42] to run the experiments, allocating 15 *pc3000* machines and a 1Gbps switched network. The machines were 3.0 GHz 64-bit Pentium Xeon with 2GB of RAM and

[3]For example, a `System.exit(0)` call that shutdowns the server.



(a) Tuple insertion (*out*) latency. (b) Tuple reading (*rdp*) latency. (c) Tuple removal (*inp*) latency.

(d) Tuple insertion (*out*) throughput. (e) Tuple reading (*rdp*) throughput. (f) Tuple removal (*inp*) throughput.

Figure 2: DepSpace operations performance with (conf) and without (not-conf) confidentiality enabled for different tuple sizes and $n = 4$. We report also performance values for a commercial non-fault-tolerant tuple space (giga).

gigabit network cards. The network is emulated as a VLAN configured in a Cisco 4509 switch with near zero latency. The software installed on the machines was Red Hat Linux 6 with kernel 2.4.20 and Sun's 32-bit JRE version 1.6.0_02. All experiments were done with the Just-In-Time (JIT) compiler enabled, and run a warm-up phase to transform the bytecode into native code. We concluded that JIT compilation cuts down the cryptographic processing delays around 10 times.

Two logical tuple space configurations were deployed for the experiments: the complete system with all layers (conf) and the system with the confidentiality layer deactivated (not-conf). We did not measure configurations with or without policy enforcement and access control layers because these layers do only modest local computations, which are not significative when compared to replication and confidentiality.

To better understand the costs of making the tuple space model dependable, we compared the performance of our system (DepSpace) with GigaSpaces XAP Community version 6.0 (*giga* in the graphics) [23]. GigaSpaces is a commercial highly-scalable production-level tuple space system developed in Java that is used in mission critical applications. This version of GigaSpaces is implemented as a non-replicated application server and thus does not tolerate faults. Neither the two configurations of DepSpace nor GigaSpaces execute disk access during the experiments. Although DepSpace does tolerates faults, all performance values were obtained in fault-free executions.

Figure 2 presents experimental results considering tuples with 4 comparable fields, with sizes of 64, 256, and 1024 bytes, running on a system with 4 servers in the case of DepSpace (tolerating 1 failure) or a single tuple space server for GigaSpaces. We report latency and throughput values for the main tuple space operations: tuple insertion (*out*), tuple read (*rdp*), and tuple removal (*inp*). To execute latency experiments, we deployed a client in one machine and servers in other four (or one, in the case of GigaSpaces). We executed each operation 1000 times and obtained the mean time and standard deviation discarding the 5% values with greater variance. To execute throughput experiments, we deployed clients in one to ten machines executing operations on DepSpace (or GigaSpaces). We varied the number of clients and measured the maximum throughput obtained in each configuration.

The results presented in the figure show that *out* and *inp* have almost the same latency on DepSpace when the confidentiality layer is not used (Figures 2(a) and 2(c)). This is mainly due to the latency imposed by the total order multicast protocol (about 3.5 ms). *rdp*, on the other hand, is much more efficient (less than 2 ms) due to the optimization presented in Section 4.6, which avoids running the total order multicast protocol in read-only operations (Figure 2(b)). The confidentiality scheme adds from 3 ms to 4 ms in all operations. GigaSpaces latency is always less than 2 ms, which is the same latency for the optimized tuple reading in *not-conf* configurations.

From the latency results (Figures 2(a)-2(c)), it is clear that the size of the tuple being inserted has almost no effect on the latency experienced by the protocols. This happens due to two implementation features: *(i.)* the BFT agreement is made 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 total order multicast protocol, but only its hash, which always have the same size. With feature *(ii.)* we can execute all the required PVSS cryptography 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 of the size of the tuple.

DepSpace provides about a third of *out*-throughput and a half of *inp*-throughput, when compared with GigaSpaces (a non-fault-tolerant tuple space). This is a consequence of the maximum throughput attained by our BFT total order multicast implementation. We believe that if more aggressive BFT replication protocols are used (e.g., Zyzzyva [28]), the attained throughput can be better. The confidentiality scheme has little impact on DepSpace's throughput because most of the cryptographic processing happens at the client side (see Table 2). Considering the perceived *rdp*-throughput, both DepSpace configurations outperform GigaSpaces. We suspect that this happens because we use manual serialization, which is more efficient than standard Java serialization, which is used in GigaSpaces.

The figure shows also that, even with large tuples, the reduction in throughput is reasonably small, e.g., increasing the tuple size 16 times 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 the total order multicast protocol [14].

We do not report results for configurations with more than four servers for two reasons. First, in our opinion it is currently infeasible to implement BFT systems with more replicas due to the difficulty of justifying the fault-independence assumption [34]. Second, the fault-scalability limitations of the kind of replication protocol used by DepSpace are well studied [1, 18, 28]. However, we present the cost of the most important cryptographic operations used in the confidentiality scheme, for three different configurations of n and f in Table 2.

Operation	4/1	7/2	10/3	Side
share	2.94	4.91	6.90	client
prove	0.47	0.49	0.48	server
verifyS	1.48	1.51	1.50	client
combine	0.12	0.14	0.23	client
RSA sign	9.79			server
RSA verify	0.52			client

Table 2: Cryptographic costs (in ms) of operations used in the confidentiality scheme for different n/f and a 64 bytes tuple and the side in which it is used.

The table shows several interesting issues. First, the operation share is the only one that increases its costs significantly as n increases in the system, but it is executed at the client side. Second, almost all significant confidentiality processing costs are at the client side. The small share extraction cost (function prove) is the only cryptographic processing at server side in the optimized protocol (where RSA sign is not used). Moreover, it is executed only once during a tuple lifetime. This shows that the confidentiality scheme proposed in this paper is scalable, i.e., it is not expected to have a significant impact on a BFT system throughput. Third, all PVSS operations are less costly than a standard 1024-bit RSA signature generation. This means that our confidentiality scheme will not turn a system significantly less responsive than secure systems that rely on standard cryptographic primitives. Finally, since operation verifyS must be executed at the client at least $f+1$ times in the non-optimized version of the read protocol (Algorithm 2, step C3), shows that our optimization to avoid this verification is crucial to the responsiveness of the system.

7. USING DEPSPACE

In this section we present the design of several services that can be built over DepSpace. Due to space constraints, we only sketch the design of these services, in order to give a feeling of how to use our system to program useful abstractions. Other examples of how to build distributed systems abstractions using tuple spaces can be found in [6, 9, 13, 23, 30, 33].

Partial barrier. A barrier is an abstraction accessed by a set of distributed processes through an enter operation. The processes stay blocked until all invoke this operation. A *partial barrier* is a weaker abstraction in which it is not required that all processes enter the barrier for the barrier to be released [3]. This abstraction is more adequate for dynamic and fault-prone environments.

With DepSpace, we implemented a partial barrier with the same semantics as the one proposed in [3]. The idea is very simple: to create a barrier, a tuple $\langle \text{BARRIER}, N, P, I \rangle$ defining the barrier's name N, the set of processes that can enter the barrier P and other info I (e.g., the percentage of processes from P that is required to enter the barrier for it to be released). To enter a barrier, a process p first reads the barrier tuple and them inserts a tuple $\langle \text{ENTERED}, N, p \rangle$ with the name of the barrier N and its id p. Then, process p keeps reading the tuple space until it sees that the required number of processes inserted "entered" tuples in the space. This reading can be implemented with a single blocking operation $rdAll(\langle \text{ENTERED}, N, * \rangle, k)$ in which k corresponds to the number of required tuples to be read. A security policy is deployed in the space to ensure that *(i.)* no two barriers with the same name can be created, *(ii.)* only the processes in P can insert entered tuples, *(iii.)* a process can insert at most one entered tuple per barrier and the id field must contain its id. Notice that this design tolerates Byzantine faults on clients and servers, which is not the case of the barrier presented in [3].

Lock service. Another example of synchronization service that can be build using DepSpace is a *lock service*. It provides lock and unlock operations for stored objects, in the same way as the lock service provided by Chubby [8]. The idea is that the presence of a "locked" tuple in the space represents that the object is locked, while the non-existence of this tuple means that the object is not locked. To lock an object the *cas* operation is executed to insert the "locked" tuple: if there is no locked tuple for the object in the space, a tuple of this type is inserted. To unlock an object, the "locked" tuple is removed from the space by its owner (using

inp). Since there are several opportunities for a malicious processes to disrupt this algorithm, a policy must be deployed in the space to ensure that the state of the tuple space is always valid. Additionally, "locked" tuples should have a lease time associated to guarantee that after a certain time the tuple is deleted from the space, i.e., the lock is released.

Secret Storage. Since our design assumes that tuples are relatively small pieces of data, we do not expect that general purpose storage systems are built using DEPSPACE. However, it can be used to implement metadata services like key management systems, file systems metadata servers and naming and directory services.

As an example, it is almost trivial to implement a secret storage service with the same guarantees offered by the CODEX system [31]. This service provides three main operations: $create(N)$, to create a name N; $write(N, S)$, to bind a secret S to a name N; and $read(N)$, to read the secret associated with the name N. A fundamental property of CODEX is that a name can not be deleted and the binding of a name and a secret have at-most-once semantics, i.e., if the secret S is bound to name N, no other secret S' can be bound to that name. The secret storage service uses secret sharing to ensure that a bounded secret will not be revealed to a malicious adversary as long as at most f out of n servers are faulty.

The confidentiality scheme of DEPSPACE makes it very easy to implement a service like CODEX: to create a name, a client executes $out(\langle \text{NAME}, N\rangle)$ with a protection type vector $\langle PU, CO\rangle$ to insert a name tuple in the space; to bind a secret to a name, a client executes $out(\langle \text{SECRET}, N, S\rangle)$ with a protection type vector $\langle PU, CO, PR\rangle$; to read a secret a client executes $rdp(\langle \text{SECRET}, N, *\rangle)$. A policy must be deployed in the space to ensure that *(i.)* there is at most one name tuple in the space with some name N, *(ii.)* at most one secret tuple can be inserted in a space for a name N, and only if there is a name tuple for N; and *(iii.)* no name or secret tuple can be removed. Access control is used to define what are the required credentials to read secret tuples.

Naming service. A more complex storage service that can be built using DEPSPACE is a naming service. The idea is to have tuples to describe naming trees stored in the service. A tuple $\langle \text{DIRECTORY}, N, D\rangle$ represents a directory with name N and with parent directory D. A tuple $\langle \text{NAME}, N, V, D\rangle$ represents a name N bound with a value V associated with a parent directory D. The most complex operation to be implemented in a naming service is the update operation, in which the value associated to a name is updated. The problem is that the tuple space abstraction does not support updates on stored tuples: the tuple must be removed, updated and inserted. In high level, the solution for this problem is to insert a temporary name tuple in the space; then remove the outdated name tuple. A policy must be deployed in the space to avoid directory tree corruption by faulty processes.

8. LESSONS LEARNED

In this section we list some lessons learned from implementing, testing and using DEPSPACE.

BFT replication library. Apart from PBFT [14] (which is outdated in terms of OS support), none of the subsequent BFT replication protocols (e.g., [1, 18, 28]) have complete implementations available to be used. Most of these systems only implement the contention- and fault-free parts of the protocol with several optimizations to obtain best-case performance figures for the proposed algorithm. When implementing the Byzantine Paxos protocol, we found it very easy to implement the optimized fault-free case (about 2K lines of code), but much harder to implement the controls needed to support all fault scenarios (our complete replication layer have about 5K lines of code). Moreover, as already pointed in [16], converting a published fault-tolerant algorithm into a production-ready library requires a lot of effort implementing features and optimizations that do not appear in the original protocol.

Providing confidentiality. Our effort to build a confidentiality scheme to be used on top of BFT state machine replication let us draw the following observations. First, as showed by the reasonable complexity of our confidentiality scheme, a secret sharing scheme alone is not sufficient to give confidentiality for data. Second, it is interesting to note that despite the large body of research on secret sharing schemes, there is no publicly available implementation of a non-trivial scheme; we had to implement it from scratch. Finally, a secret sharing scheme is not as slow as we thought. Our experiments showed that the latency of generating and combining shares is about half the time needed to sign a message using 1024-bit RSA.

Pursuing fault independence. During the development of DEPSPACE we investigated how to make the fault independence assumption valid for our system [34]. Is it worth to implement multiple versions of DEPSPACE? Is it worth to exploit opportunistic diversity [15] using different tuple spaces implementations on each DEPSPACE replica? Both these possibilities require a high implementation effort, which is obvious for multiple version programming, but less obvious for opportunistic diversity. In the case of tuple spaces, the most common problem in using diverse local tuple spaces would be the non-determinism of read/remove operations. A solution for this problem would be the use of abstract specifications [15], but it would impose a high performance loss in our system. Investigating the sources of diversity and the vulnerabilities associated with security attacks, we found that the most common problems are due to faults in infrastructure software (operating systems, services like naming, file systems, etc.) or bad system management, not on application software. Given the costs on supporting diverse implementations, and the sources of security problems, we found out that it is more interesting to spend resources trying to improve a single implementation of a tuple space and deploy the different replicas in diverse locations, with different operating systems and administrators. The fact that our system is built using Java make this pragmatical approach even more attractive.

The power of an abstraction. The tuple space model allows the development of a large spectrum of services and applications using the abstraction of a bag of tuples and some operations to store and retrieve them. Given the current trend of the increasing complexity of computer sys-

tems, we believe that pursuing the use of powerful and efficient abstractions should be a priority in our current research agenda, specially in areas that are complex by nature, like Byzantine fault tolerance. Concerning the tuple space model, we found that revising and extending this abstraction allow us to work efficiently in research areas as diverse as fault-tolerant grid scheduling [20], BFT agreement/message ordering layers (in the sense of [44]) and web services collaboration.

9. RELATED WORK

As a general purpose coordination service, the system that most resembles DepSpace is Google's Chubby lock service [8]. However, there are at least three important differences between these systems. First, Chubby provides an (extended) file system abstraction while DepSpace implements the tuple space model. Second, despite the fact that both systems provide high availability, DepSpace tolerates Byzantine faults and provides data confidentiality, while Chubby assumes a non-malicious fault model. Finally, Chubby is a production-level system used by many clients, while DepSpace is a proof-of-concept prototype. Another recent system that resembles DepSpace in some aspects is Sinfonia [2]. Sinfonia provides a transactional shared memory abstraction that can be used to store data with different qualities of service and synchronize processes using mini-transactions, which are constrained transactions that can be implemented very efficiently. The main difference between Sinfonia and DepSpace are in the abstractions provided and the goals they achieve. Regarding the abstraction, Sinfonia provides a low-level address-based memory that supports constrained transactions while DepSpace provides a content-addressable tuple space extended with a synchronization operation (*cas*). Considering the objectives of the two systems, while Sinfonia provides a scalable solution to coordinate processes in a relatively safe environment (data centers) and tolerates only crash faults, DepSpace does not address scalability but tries to provide an ultra-dependable solution for coordinating process in unreliable and untrusted environments. The secret storage service CODEX [31] has also some similarities with DepSpace since both systems provides confidentiality for generic data. The main difference is that while CODEX provides simple operations to deal with secret storage and retrieval, our system focus on providing a generic abstraction that offers storage and synchronization operations.

There are several works that aim to improve the dependability of the tuple space coordination model using security or fault tolerance mechanisms. Several papers have proposed the integration of security mechanisms in tuple spaces. Amongst these works, some try to enforce security policies that depend on the application [33], while others provide integrity and confidentiality using access control at tuple space level, tuple level or both [9, 41]. However, these works consider centralized servers so they are not fault-tolerant, neither propose a confidentiality scheme like the one proposed in this paper. Some of the work on fault tolerance for tuples spaces aims to increase the dependability of the tuple space using replication [5, 43]. However, all of these proposals are concerned with crash failures and do not tolerate malicious faults, the main requirement for intrusion tolerance. DepSpace is the first system to employ state-of-the-art BFT replication to implement a tuple space.

Recently, some of the authors have proposed a quorum-based linearizable Byzantine-resilient tuple space – LBTS [7]. The work presented in this paper differs from LBTS since it presents a complete solution for building a dependable tuple space while LBTS focus on a novel replication algorithm, and thus does not provides access control, policy enforcement and tuple confidentiality. Additionally, LBTS does not implement the *cas* operation and thus can not be used to solve important synchronization problems like consensus. Moreover, LBTS was not fully implemented on the contrary to DepSpace.

Despite the existence of several proposals of BFT replication protocols (e.g., [1, 14, 18, 28, 44]), only the privacy firewall described in [44] tries to improve the confidentiality of replicated services. The idea is to introduce a privacy firewall between the service replicas and the agreement layer (that gives total order to requests issued to the BFT replicated service) to avoid that faulty service replicas disclose service information to other parties. The privacy firewall works only if all communication between service replicas and other system components is inspected by the firewall. The confidentiality scheme used in DepSpace considers that service data is shared between servers in such a way that at least $f + 1$ servers are required to disclose encrypted data.

There are several works that advocate the use of secret sharing to ensure data confidentiality. The seminal paper in the area had a partial solution for confidentiality in synchronous systems since data was scattered in several servers [21]. We are aware of only two works that guarantee the confidentiality of replicated data in asynchronous systems. The first is the Secure Store, which uses a secret sharing scheme to split data among the servers [29]. To reconstruct the original data, the collaboration of at least $f+1$ servers is needed. The scheme is not very efficient so the data shares created are replicated in several servers, thus using a large number of servers. Moreover, the system does not tolerate malicious clients. The second solution is used to store (large) files efficiently in a set of servers so it uses an information dispersal scheme [12]. The idea is essentially to encrypt the file with a random secret key, split it in parts using an erasure code scheme, and send these parts to the servers. The secret key is also split in shares and sent to the servers, which use a threshold cryptography scheme to reconstruct the key (requires the collaboration of $f + 1$ servers). In the same line, there are several other works that use information dispersal based on erasure codes for efficient data storage (e.g., [24, 25]). Despite the fact that both secret sharing and erasure codes have similarities (both generate n "shares" of a data block and at least $f+1 \leq n$ of these shares are required to recover the block), the properties ensured by these techniques are very different: erasure code shares are smaller than the original data block and do provide some information about their content (provide efficient storage but not confidentiality), while secret sharing shares have usually the same size of the original block and do not reveal any information about their content (provide confidentiality but not efficient storage). So, in works that use erasure codes for efficient storage confidentiality is usually not addressed. DepSpace does not employ information dispersal since tuples are expected to be reasonable small pieces of data. Our solution is closer to the Secure Store since it uses secret sharing, but does not have an impact in terms of number of servers and tolerates any number of malicious clients. Moreover, ours is the first work

to combine state machine replication and a confidentiality scheme, presenting also a practical assessment of the performance of this kind of scheme.

10. CONCLUSIONS

The paper presents a solution for the implementation of a dependable – fault- and intrusion-tolerant – tuple space. The proposed architecture integrates several dependability and security mechanisms in order to enforce a set of dependability properties. This architecture was implemented in a system called DepSpace, a coordination service for untrusted and dynamic environments. DepSpace implements an abstraction called PEATS (Policy-Enforced Augmented Tuple Space), an object that can be used to build distributed computing abstractions efficiently even in Byzantine-prone environments [6].

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 BFT state machine replication and data confidentiality. Experimental evaluation shows that despite the fact the confidentiality scheme has some impact in the performance of the system, it has little impact on the system perceived throughput. Although the confidentiality scheme was developed for tuple spaces, it can be adapted to be used in any content-based data storage system.

DepSpace source code (including the total order multicast and the PVSS scheme) is available at project homepage: `http://www.navigators.di.fc.ul.pt/software/depspace`.

Acknowledgments

We would like to thank Lau Cheuk Lung, Fabio Favarim, Wagner Saback Dantas and the EuroSys'08 reviewers for their contributions to improve this paper. This work was partially supported by the EC, through projects IST-2004-27513 (CRUTIAL) and NoE IST-4-026764-NoE (ReSIST), by the FCT, through the Multiannual and the CMU-Portugal Programmes, and by CAPES/GRICES (project TISD).

11. REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, Oct. 2005.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, Oct. 2007.

[3] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of the 2006 Usenix Annual Technical Conference – Usenix'06*, 2006.

[4] 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.

[5] 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.

[6] A. N. Bessani, M. Correia, J. S. 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.

[7] A. N. Bessani, M. Correia, J. S. Fraga, and L. C. Lung. Decoupled quorum-based Byzantine-resilient coordination in open distributed systems. Technical Report DI-FCUL TR 07–9, Dep. of Informatics, University of Lisbon, May 2007.

[8] M. Burrows. The chubby lock service. In *Proceedings of 7th Symposium on Operating Systems Design and Implementations - OSDI 2006*, Nov. 2006.

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

[10] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of Linda coordination primitives. *Information and Computation*, 156(1-2):90–121, Jan. 2000.

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

[12] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems - SRDS 2005*, Oct. 2005.

[13] N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989.

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

[15] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3), Aug. 2003.

[16] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing - PODC'07*, Aug. 2007.

[17] Codehaus. Groovy programing language homepage. Avaliable at *http://groovy.codehaus.org/*, 2006.

[18] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th Symposium on Operating Systems Design and Implementations - OSDI 2006*, Nov. 2006.

[19] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, Apr. 1988.

[20] F. Favarim, J. S. Fraga, L. C. Lung, and M. Correia. GridTS: A new approach for fault-tolerant scheduling in grid computing. In *Proceedings of 6th IEEE Symposium on Network Computing and Applications - NCA 2007*, July 2007.

[21] J. S. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd*

International Conference on Computer Security, 1985.

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

[23] GigaSpaces. GigaSpaces – write once, scale anywere. Avaliable at *http://www.gigaspaces.com/*, 2007.

[24] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of Dependable Systems and Networks - DSN 2004*, June 2004.

[25] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, Oct. 2007.

[26] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, Jan. 1991.

[27] 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.

[28] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, Oct. 2007.

[29] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):818–828, Sept. 2003.

[30] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, Mar. 2001.

[31] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, Jan. 2004.

[32] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[33] N. H. Minsky, Y. M. Minsky, and V. Ungureanu. Making tuple-spaces safe for heterogeneous distributed systems. In *Proceedings of the 15th ACM Symposium on Applied Computing - SAC 2000*, 2000.

[34] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, University of Lisbon, Sept. 2006.

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

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

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

[38] A. Shamir. How to share a secret. *Communications of ACM*, 22(11):612–613, Nov. 1979.

[39] Sun Microsystems. JavaSpaces service specification. Availiable at *http://www.jini.org/standards*, 2003.

[40] P. Verissimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

[41] J. Vitek, C. Bryce, and M. Oriol. Coordination processes with Secure Spaces. *Science of Computer Programming*, 46(1-2):163–193, Jan. 2003.

[42] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of 5th Symposium on Operating Systems Design and Implementations - OSDI 2002*, Dec. 2002.

[43] 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*, June 1989.

[44] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, Oct. 2003.

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