Sharing Memory between Byzantine Processes using Policy-Enforced Tuple Spaces

Alysson Neves Bessani† Miguel Correia‡ Joni da Silva Fraga† Lau Cheuk Lung§
† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
‡ LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
§ Prog. de Pós-Grad. em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brazil
neves@das.ufsc.br mpc@di.fc.ul.pt fraga@das.ufsc.br lau@ppgia.pucpr.br

Abstract

Despite the large amount of Byzantine fault-tolerant algorithms for message-passing systems designed through the years, only recently algorithms for the coordination of processes subject to Byzantine failures using shared memory have appeared. This paper presents a new computing model in which shared memory objects are protected by fine-grained access policies, and a new shared memory object, the policy-enforced augmented tuple space (PEATS). We show the benefits of this model by providing simple and efficient consensus algorithms. These algorithms are much simpler and use less memory bits than previous algorithms based on ACLs and sticky bits. We also prove that PEATSs are universal (they can be used to implement any shared memory object), and present an universal construction.

1. Introduction

Despite the large amount of Byzantine fault-tolerant algorithms for message-passing systems designed through the years [4, 8, 9, 10, 16], only recently algorithms for the **coordination of processes subject to Byzantine failures using shared memory** have appeared [1, 3, 15]. The motivation for this line of research is the current availability of several solutions for the emulation of dependable shared memory objects on message-passing distributed systems subject to Byzantine failures [7, 8, 9, 10, 16]. The fundamental question regarding this research is: what is the power of shared memory objects to coordinate processes that can fail in a Byzantine way, i.e., arbitrarily [15]? The question is specially relevant since this kind of failures can be used to model the behavior of malicious hackers and malware [9, 8, 10]. In a nutshell, the objective is to mask these failures using shared memory objects.

The first works in this area have made several important theoretical contributions. They have shown that simple objects like registers and sticky bits [19] when combined with access control lists (ACLs) are enough to implement consensus [15], that the optimal resilience for strong consensus is $n \geq 3t+1$ in this model [1, 15] (t is an upper bound on the number of faulty processes and n the total number of processes), and that sticky bits with ACLs are universal, i.e., they can be used to implement any shared memory object [15], to state only some of those contributions.

Despite the undeniable importance of these theoretical results, on the practical side these works also show the limitations of combining simple objects like sticky bits with ACLs: the amount of objects required and the amount of operations requested in these objects is enormous, making the developed algorithms impractical for real systems. The reason for this is that the algorithms fall in a combinatorial problem. There are n processes and k shared memory objects for which we have to setup ACLs associating objects with processes in such way that faulty processes cannot invalidate the actions of correct processes.

The present paper contributes to this area by modifying this model in two aspects. First, the paper proposes the use of **fine-grained security policies** to control the access to shared memory objects. These policies allow us to specify when an invocation to an operation in a shared memory object is to be allowed or denied in terms of who invokes the operation, what are the parameters of the invocation and what is the state of the object. We call the objects protected by these policies **policy-enforced objects** (PEOs).

Second, the paper uses only one type of shared memory object: an **augmented tuple space** [5, 21]. This object, which is an extension of the tuple space introduced in LINDA [11], stores generic data structures called tuples. It provides operations for the inclusion, removal, reading and conditional inclusion of tuples.

The paper shows that **policy-enforced augmented tuple spaces** (PEATSs) are an attractive solution for the coordi-

nation of Byzantine processes. The paper provides algorithms for consensus and an universal construction that are much simpler than previous ones based on sticky bits and ACLs [1, 15]. They are also more efficient in terms of number of bits and objects needed to solve a certain problem. This comparison of apparently simple objects like sticky bits with apparently complex objects like tuple spaces may seem unfair but in reality the implementation of linearizable versions of both (the case we consider here) involves similar protocols with similar complexities. For instance, both can be implemented similarly using the above mentioned Byzantine fault-tolerant systems based on state machine replication [8, 9, 10].

2. Model and Definitions

2.1. System Model

The model of computation consists of an asynchronous set of n processes $\mathscr{P} = \{p_1, p_2, ..., p_n\}$ that communicate via a set of k shared memory objects $\mathscr{O} = \{o_1, ..., o_k\}$ (e.g., registers, sticky bits, tuple spaces). Each of these processes may be either **faulty** or **correct**. A correct process is constrained to obey its specification, while a faulty process, also called a **Byzantine** process, can deviate arbitrarily from its specification. We assume that a malicious process cannot impersonate a correct process when invoking an operation on a shared memory object. This limitation is important in our model since we will use a reference monitor [2] to enforce the access policy. This monitor must know the correct identity of the process invoking operations on the object in order to grant or deny access to the operation.

A **configuration** of a shared memory distributed system with n processes communicating using k shared memory objects is a vector $C = \langle q_1, ..., q_n, r_1, ..., r_k \rangle$ where q_i is the state of the process p_i and r_i is the state of the object o_i. A **step** of a process is an action of this process that changes the system configuration (the state of a process and/or object). An **execution** of a distributed system is an infinite sequence $C_0, a_0, C_1, a_1, ...$ where C_0 is an initial configuration and each a_i is the step that changes the system state from C_i to C_{i+1}.

Each shared memory object is accessed through a set of operations made available through its interface. An object operation is executed by a process when it makes an **invocation** to that operation. An operation ends when the process receives a **reply** for the corresponding invocation. An operation that has been invoked but not replied to is called a **pending operation**. We assume that all processes (even the faulty ones) invoke an operation on a shared memory object only after receiving the reply for its last operation on this object. This condition is sometimes called **well formedness** or **correct interaction** [4][1].

The shared memory objects used in this paper are assumed to be dependable (they do not fail) and to satisfy the **linearizability** correctness condition [13]: although they are accessed concurrently, every operation executed on them appears to take effect instantaneously at some point between its invocation and reply, in such a way that they appear to have been accessed sequentially.

In term of liveness, all operations provided by the shared memory objects used in this paper satisfy one of the following termination conditions (x is a shared memory object):

- **lock-freedom**: an operation x.op is lock-free if, when invoked by a correct process at any point in an execution in which there are pending operations invoked by correct processes, some operation (either x.op or any of the pending operations) will be completed;
- t-**resilience** [15]: an operation x.op is t-resilient if, when executed by a correct process, it eventually completes in any execution in which at least $n-t$ correct processes infinitely often have a pending invocation for some operation of x;
- t-**threshold** [15]: an operation x.op is t-threshold if, when executed by a correct process, it eventually completes in any execution in which at least $n-t$ correct processes invoke x.op;
- **wait-freedom** [12]: an operation x.op is wait-free if, when executed by a correct process, it eventually completes in any execution (despite the failures of other processes).

The difference between t-threshold and t-resilience, is the fact that in the first an operation completes only if $n-t$ correct processes invoke the **same** operation, and in the second an operation completes only if $n-t$ correct processes keep invoking some operation on the object. Notice that t-threshold implies t-resilience, but not vice-versa.

For any of these liveness conditions, we say that an object satisfies the condition if all its operations satisfy the condition.

2.2. Augmented Tuple Space

The **tuple space** coordination model, originally introduced in the LINDA programming language [11], allows distributed processes to interact through a shared memory object called a tuple space, where generic data structures called **tuples** are stored and retrieved.

[1] This is just a simplification to improve the presentation of the algorithms. This assumption can be easily enforced by making the objects ignore invocations made by processes that have pending invocations.

Each tuple is a sequence of typed fields. A tuple in which all fields have their values defined is called an **entry**. A tuple that has one or more fields with undefined values is called a **template** (indicated by a bar, e.g., $\bar{t}$). An undefined value can be represented by the wildcard symbol '$*$' (meaning "any value") or by a **formal field**, denoted by a variable name preceded by the character '?' (e.g., ?v).

The **type of a tuple** t is the sequence of types of each field of t. An entry t and a template $\bar{t}$ **match**, denoted $m(t,\bar{t})$, iff *(i.)* they have the same type and *(ii.)* all defined field values of $\bar{t}$ are equal to the corresponding field values of t. The variable in a formal field (e.g., v in ?v) is set to the value in the corresponding field of the entry matched to the template.

There are three basic operations on a tuple space [11]: $out(t)$, which outputs the entry t in the tuple space (write); $in(\bar{t})$, which removes a tuple that matches $\bar{t}$ from the tuple space (destructive read); and $rd(\bar{t})$, which is similar to $in(\bar{t})$ but does not remove the tuple from the space (non-destructive read). The *in* and *rd* operations are blocking, i.e., if there is no tuple in the space that matches the specified template, the invoking process will wait until a matching tuple becomes available.

A common extension to this model, which we adopt in this paper, is the inclusion of non-blocking variants of these read operations, called *inp* and *rdp* respectively. These operations work in the same way as their blocking versions but return even if there is no matching tuple for the specified template in the space (signaling the operation's result with a boolean value). Notice that according to the definitions above, the tuple space works just like an associative memory: tuples are accessed through their contents, not through their address. This programming model allows expressive interactions to be described with few lines of code [11].

In Herlihy's hierarchy of shared memory objects [12], the tuple space object has consensus number 2 [21], i.e., it can be used to solve consensus between at most two processes. In this paper we want to present algorithms to solve consensus and build universal constructions for any number of processes, so we need universal shared memory objects (consensus number n) [4, 12]. Therefore, we use an **augmented tuple space** [5, 21] which provides an extra **conditional atomic swap** operation. This operation, denoted by $cas(\bar{t},t)$ for a template $\bar{t}$ and an entry t, works like an atomic (indivisible) execution of the instruction:

$$\textbf{if } \neg rdp(\bar{t}) \textbf{ then } out(t)$$

The meaning of this instruction is "if the reading of $\bar{t}$ fails, insert the entry t in the space"[2]. This operation returns *true* if the tuple is inserted in the space and *false* otherwise.

[2]Notice that *cas* is similar to the register compare&swap operation [4] but in some sense does the opposite, because compare&swap modifies the register if its value is **equal** to the value compared.

The augmented tuple space is a universal shared memory object, since it can solve wait-free consensus trivially in the crash fault model [5, 21] as well as in the Byzantine model (as will show in this paper) for any number of processes.

All algorithms proposed in this paper are based on a single linearizable wait-free augmented tuple space.

3. Policy-Enforced Objects

Previous works on objects shared by Byzantine processes consider that the access to operations in these objects is protected by ACLs [1, 3, 15]. In that model, each operation provided by an object is associated to a list of processes that have access to that operation. Only processes that have access to an operation can execute it. Notice that this model requires a kind of reference monitor [2] to protects the objects from unauthorized access. The implementation of this monitor is not problematic since, in general, it is assumed that the shared memory objects are implemented using replicated servers [8, 9, 10, 16], which have processing power.

In this paper, we also assume this kind of implementation but extend the notion of protection to more powerful security polices than access control based on ACLs. We define **policy-enforced objects** (PEOs), which are objects whose access is governed by a fine-grained security policy. Later, we argue that the use of these policies make possible the implementation of simple and efficient algorithms that solve several important distributed problems, for instance, consensus.

A reference monitor permits the execution of an operation on a PEO if the corresponding invocation satisfies the access policy of the object. The **access policy** is composed by a set of rules. Each rule is composed by an invocation pattern and a logical expression. An execution is allowed (predicate $execute(\text{op})$ set to *true*) only if its associated logical expression is satisfied by the invocation pattern. Following the principle of fail-safe defaults, any invocation that does not fit in any rule is always denied [20]. A logical value *false* is returned by the operation whenever the access is denied.

The reference monitor has access to three pieces of information in order to evaluate if an invocation $invoke(p,\text{op})$ to a protected object x can be executed:

- the invoker process identifier p;
- the operation op and its arguments;
- the current state of x.

An example is a policy-enforced numeric atomic register r in which only values greater than the current value can be written and in which only processes p_1, p_2 and p_3 can write.

The access policy for that PEO is represented in Figure 1. We use the symbol ':- ' taken from the PROLOG programming language to state that the predicate in the left hand side is true if the condition in the right hand side is true. The *execute* predicate (left hand side) indicates if the operation is to be executed, and the predicate *invoke* (right hand size) indicates if the operation was invoked.

Object State r
R_{read}: $execute(read()) \text{ :- } invoke(p, read())$
R_{write}: $execute(write(v)) \text{ :- } invoke(p, write(v)) \wedge$
$p \in \{p_1, p_2, p_3\} \wedge v > r$

Figure 1. An example of access policy for an atomic register.

In the access policy in Figure 1, we initially define the elements of the object's state that can be used in the rules. In this case, the register state is specified by its current value, denoted r. Then, one or more access rules are defined. The first rule (R_{read}) says that all register readings are allowed. The second rule (R_{write}) states that a $write(v)$ operation invoked by a process p, can only be executed if *(i.)* p is one of the processes in the set $\{p_1, p_2, p_3\}$ and *(ii.)* the value v being written is greater than the current value of the register r. Notice that condition *(i.)* is nothing more than a straightforward implementation of an ACL in our model.

The algorithms presented in the paper are based on a **policy-enforced augmented tuple space** object (PEATS). The implementation of this kind of object (or another PEO in general) on distributed message-passing systems could be based on interceptors [14], that would grant or deny access to the operations according to the access policy of the object. A straightforward implementation would be to replicate the PEATS in a set of servers. There would be one interceptor in each replica, which would be in charge of enforcing the policy in that replica. The access policy could be hard-coded in the interceptor, or a more generic policy enforcer like the one in [18] might be used. Notice that the policy is enforced strictly locally; there is no need for communication between the interceptors. All the interceptors in correct replicas always take the same decision to grant/deny an operation because policies are deterministic and evaluate the same data in all correct replicas. A complete implementation of a dependable PEATS is described in [6].

4. Solving Consensus

In this section we illustrate the benefits of using a PEATS to solve two variants of the consensus problem. This problem concerns a set of n processes proposing values from a set of possible values $\mathscr{V}$ and trying to reach agreement about a single decision value. A consensus object is a shared memory object that encapsulates a consensus algorithm. Next, we present algorithms to implement two kinds of consensus objects:

- **Weak Consensus** [15]: A weak consensus object x is a shared memory object with a single operation x.propose(v), with $v \in \mathscr{V}$, satisfying the properties: (**Agreement**) in any execution, x.propose returns the same value, called the **consensus value**, to every correct process that invokes it; (**Validity**) in any finite execution in which all participating processes are correct, if the consensus value is v, then some process invoked x.propose(v).

- **Strong Consensus** [15]: A strong consensus object x is defined by a stronger Validity condition than weak consensus objects: (**Strong Validity**) if the consensus value is v, then some correct process invoked x.propose(v).

Both objects have also to satisfy one of the termination conditions given in Section 2.1.

4.1. Weak Consensus Object

In a weak consensus object, the consensus value can be any of the proposed values. With this validity condition it is perfectly legal that a value proposed by a faulty process becomes the consensus value.

Algorithm 1 Weak Byzantine consensus (process p_i).

Shared variables:
1: $ts = \emptyset$ {PEATS object}
procedure x.propose(v)
2: **if** $ts.cas(\langle \text{DECISION}, ?d \rangle, \langle \text{DECISION}, v \rangle)$ **then**
3: $d \leftarrow v$ {decision value (v) inserted}
4: **end if**
5: **return** d

Algorithm 1 presents the algorithm that implements weak consensus using a PEATS. The algorithm is very simple: a process tries to insert its proposal in the PEATS object using the *cas* operation. It succeeds only if there is no decision tuple in the space. If there is already a decision tuple, this is the value to be decided and it is returned.

Object State TS
R_{cas}: $execute(cas(\langle \text{DECISION}, x \rangle, \langle \text{DECISION}, y \rangle)) \text{ :-}$
$invoke(p, cas(\langle \text{DECISION}, x \rangle, \langle \text{DECISION}, y \rangle))$
$\wedge formal(x)$

Figure 2. Access policy for Algorithm 1.

The access policy for the PEATS used in Algorithm 1 is presented in Figure 2. The predicate $formal(x)$ is *true* if x is a formal field, otherwise it is *false*. This access policy permits only executions of the *cas* operation. The tuple must have two fields, the first with a constant DECISION and the second must be formal. Only one decision tuple can be inserted in the PEATS.

Besides its simplicity and elegance, this algorithm has several interesting properties: first it is **uniform** [4], i.e., it works for any number of processes and the processes do not need to know how many other processes are participating. Second, it can solve **multi-valued consensus**, since the range of values proposed can be arbitrary. Finally, the algorithm is **wait-free**, i.e., it always terminates despite the failure of any number of processes. An interesting point about this algorithm is that our PEATS with the access policy specified in Figure 2 behaves like a persistent object, so our result is in accordance with Theorem 4.1 of [15].

Theorem 1 *Algorithm 1 provides a wait-free weak consensus object.*

Proof: Omitted due to lack of space.

4.2. Strong Consensus Object

A strong consensus object enforces the validity condition by requiring that the consensus value be proposed by a correct process even in the presence of faulty ones. This strict condition results in a more complex (but still simple) algorithm. However, this algorithm does not share some of the benefits of the algorithm presented in the previous section: *(i.)* it is not uniform since a process has to know who are the other processes in order to read their input values and decide a consensus value proposed by some correct process; *(ii.)* it solves only **binary consensus**, also due to the fact that a process needs to know if a value has been proposed by one correct process before deciding it; *(iii.)* it is not wait-free since it requires $n-t$ processes to take part in the algorithm. Nevertheless, the number of processes needed is optimal: $n \geq 3t+1$.

Algorithm 2 presents the strong binary consensus protocol. The algorithm works as follows: a process p_i first inserts its proposal in the augmented tuple space ts using a PROPOSE tuple (line 2). Then, p_i queries ts continuously trying to read proposals (line 7) until it finds that some value has been proposed by at least $t+1$ processes (loop of lines 5-11). The rationale for the amount of $t+1$ is that at least one correct process must have proposed this value, since there are at most t failed processes. The first value that satisfies this condition is then inserted in the tuple space using the *cas* operation. This commitment phase is important since different processes can collect $t+1$ proposals for different values and we must ensure that there is a single decision value. All further invocations of *cas* return this value (lines 12-14).

Algorithm 2 Strong Byzantine consensus (process p_i).

Shared variables:

1: $ts = \emptyset$ {PEATS object}

procedure x.propose(v)

2: $ts.out(\langle \text{PROPOSE}, p_i, v\rangle)$
3: $S_0 \leftarrow \emptyset$ {set of processes that proposed 0}
4: $S_1 \leftarrow \emptyset$ {set of processes that proposed 1}
5: **while** $|S_0| < t+1 \wedge |S_1| < t+1$ **do**
6: **for all** $p_j \in \Pi \setminus (S_0 \cup S_1)$ **do**
7: **if** $ts.rdp(\langle \text{PROPOSE}, p_j, ?v\rangle)$ **then**
8: $S_v \leftarrow S_v \cup \{p_j\}$ {p_j proposed v}
9: **end if**
10: **end for**
11: **end while**
12: **if** $ts.cas(\langle \text{DECISION}, ?d, *\rangle, \langle \text{DECISION}, v, S_v\rangle)$ **then**
13: $d \leftarrow v$ {decision value (v) inserted}
14: **end if**
15: **return** d

The access policy for the PEATS used in Algorithm 2 is presented in Figure 3. This access policy specifies that any process can read any tuple; that each process can introduce only one PROPOSE entry in the space; that the second field of the template used in the *cas* operation must be a formal field; and that the decision value v must appear in proposals of at least $t+1$ processes. These simple rules, that can easily be implemented in practice, effectively constrain the power of Byzantine processes, thus allowing the simplicity of the consensus presented in Algorithm 2.

Object State TS
R_{rdp}: $execute(rdp(t))$:- $invoke(p, rdp(t))$
R_{out}: $execute(out(\langle \text{PROPOSE}, p, x\rangle))$:-
$invoke(p, out(\langle \text{PROPOSE}, p, x\rangle)) \wedge$
$\nexists y : \langle \text{PROPOSE}, p, y\rangle \in TS$
R_{cas}: $execute(cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, P\rangle))$:-
$invoke(p, cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, P\rangle)) \wedge$
$formal(x) \wedge |P| \geq t+1 \wedge$
$\forall q \in P : \langle \text{PROPOSE}, q, v\rangle \in TS$

Figure 3. Access policy for Algorithm 2.

Our algorithm requires only $n(\lceil \log n \rceil + 1) + (1 + (t+1)\lceil \log n \rceil)$ bits in the PEATS object[3] (n PROPOSE tuples plus one DECISION tuple). The consensus algorithm with the same resilience presented in [1] requires $(n+1)\binom{2t+1}{t}$ sticky bits[4].

[3] E.g., only 68 bits are needed for $t = 4$ and $n = 13$.

[4] It is a lot of memory. For example, if we want to tolerate $t = 4$ faulty processes, we need at least $n = 13$ processes and 1764 sticky bits.

Theorem 2 *Algorithm 2 provides a t-threshold strong binary consensus object.*

Proof: Omitted due to lack of space.

5. A Lock-free Universal Construction

A fundamental problem in shared memory distributed computing is to find out if an object X can be used to implement (or **emulate**) another object Y. This section proves that PEATSs are **universal objects** [12], i.e., that they can be used to emulate any other shared memory object. Herlihy has shown that an object is universal in a system with n processes if and only if it has **consensus number** n, i.e., if it can solve consensus for n processes [12].

The proof that PEATSs are universal is made by providing one universal construction based on this kind of object. A **universal construction** is an algorithm that uses one or more universal objects to emulate any other shared memory object [12]. There are several wait-free universal constructions for the crash fault model, using consensus objects [12], sticky bits [19], compare and swap registers [4] and several other universal objects. A universal construction for the Byzantine fault model using sticky bits was defined in [15]. However, this construction is not wait-free but t-resilient, which is a more appropriate termination condition for Byzantine fault-tolerant algorithms. The possibility of algorithms with stronger liveness properties is still an open problem [15].

In order to define a universal construction that emulates a deterministic object o of a certain type, we have to start by defining the type of the object. A type T is defined by the tuple $\langle STATE_T, S_T, INVOKE_T, REPLY_T, apply_T \rangle$ where $STATE_T$ is the set of possible states of objects of type T, $S_T \in STATE_T$ is an initial state for objects of this type, $INVOKE_T$ is the set of possible invocations of operations provided by objects of type T, $REPLY_T$ is the set of possible replies for these invocations, and $apply_T$ is a function defined as:

$$apply_T : STATE_T \times INVOKE_T \rightarrow STATE_T \times REPLY_T.$$

The function $apply_T$ represents the state transitions of the object. Given a state S_i and an invocation inv for an operation op, $apply_T(S_i, inv)$ gives a new state S_j (the result of the execution of operation op in state S_i) and a reply rep for the invocation. This definition is enough for showing the universality of tuple spaces, although Malkhi et al. have shown that a (trivial) generalization is needed for emulating non-deterministic types and some objects that satisfy weak liveness guarantees [15].

Our **lock-free universal construction** follows previous constructions [4, 12, 15]. The idea is to make all correct processes execute the sequence of operations invoked in the emulated object in the same order. Each process keeps a replica of the state of the emulated object S_i. An invocation inv is executed by applying the function $apply_T(S_i, inv)$ to that state. The problem boils down essentially to define a total order for the execution of the operations.

The operations to be executed in the emulated object can be invoked in any of the processes, so the definition of an order for the operations requires a consensus among all processes. Therefore, we need an object with consensus number n, i.e., a universal object.

The solution is to add (to "thread") the operations to be executed in the emulated object to a list where each element has a sequence number. The element with the greater sequence number represents the last operation to be executed on the emulated object. The consistency of the list, i.e., the property that each of its elements (each operation) is followed by one other element, is guaranteed by the universal object, a PEATS in our case. Given this list, each process executes the operations of the object emulated in the same order.

The list of operations is implemented using a PEATS object. The key idea is to represent each operation as a SEQ tuple containing a position field, and to insert each of these tuples in the space using the *cas* operation. When a process wants to execute an operation, it invokes the *cas* operation: if there is no SEQ tuple with the specified sequence number in the space, then the tuple is inserted. Algorithm 3 presents this universal object.

Algorithm 3 Lock-free universal construction (process p_i).

```
Shared variables:
 1: ts = ∅                                              {PEATS object}
Local variables:
 2: state = S_T                                  {current state of the object}
 3: pos = 0                    {position of the tail of the operations list}
invoked inv
 4: loop
 5:    pos ← pos + 1
 6:    if ts.cas(⟨SEQ, pos, ?pos_inv⟩, ⟨SEQ, pos, inv⟩) then
 7:       ⟨state, reply⟩ ← apply_T(state, inv)
 8:       return reply
 9:    end if
10:    ⟨state, reply⟩ ← apply_T(state, pos_inv)
11: end loop
```

The algorithm assumes that each process p_i begins its execution with an initial state composed by the initial state of the emulated object ($state = S_T$, line 2) plus an empty list ($pos = 0$, line 3). When an operation is invoked (denoted by inv), p_i iterates through the list updating its *state* variable (loop in lines 4-11) and trying to thread its operation by appending it to the end of the list using the *cas* operation (line 6). If *cas* is executed successfully by p_i, the *state* vari-

able is updated and the reply to the invocation is returned (lines 7 and 8).

The algorithm is lock-free due to the *cas* operation: when two processes try concurrently to put tuples at the end of the list, at least one of them succeeds. However, the algorithm is not wait-free since some processes might succeed in threading their operations again and again, delaying other processes forever.

Object State TS
R_{cas}: $execute(cas(\langle SEQ, pos, x\rangle, \langle SEQ, pos, inv\rangle))$:-
$invoke(p, cas(\langle SEQ, pos, x\rangle, \langle SEQ, pos, inv\rangle)) \wedge$
$formal(x) \wedge$
$(pos = 1 \vee \exists y : \langle SEQ, pos-1, y\rangle) \in TS)$

Figure 4. Access policy for Algorithm 3.

The access policy for our universal construction (Figure 4) states that a SEQ tuple with the second field *pos* can only be inserted in the space (using *cas*) if there is a SEQ tuple with the second field with value $pos - 1$.

The proof of the correctness of the algorithm is based on the following lemmas:

Lemma 1 *For any execution of the system, the following properties are invariants of the PEATS used in Algorithm 3:*

1. *For any $pos \geq 1$, there is at most one tuple $\langle SEQ, pos, inv\rangle$ in the tuple space;*
2. *For any tuple $\langle SEQ, pos, inv\rangle$ in the tuple space with $pos > 1$, there is exactly one tuple $\langle SEQ, pos-1, inv\rangle$ in the space.*

Proof: Omitted due to lack of space.

Lemma 2 *The universal construction of Algorithm 3 is lock-free.*

Proof: This lemma is proved by contradiction. Let α be an execution with only two correct processes p_1 and p_2 (without loss of generality) which invoke operations inv_1 and inv_2, respectively. Suppose that they stay halted forever, not receiving replies. We have to show that α does not exist. An inspection of the algorithm shows that the processes keep updating their copies of the object state until they execute the most recent threaded operation (with position field value equal to *pos*, without loss of generality). At this point, p_1 and p_2 will try to thread their invocations to the list in position $pos+1$ executing *cas* (line 6). Since the PEATS is assumed to be linearizable, the two *cas* invocations will happen one after another in some order, so either the inv_1 or the inv_2 SEQ tuples will be inserted in position $pos+1$. The process that succeeds in executing *cas* will thread its invocation and will return its reply (lines 7 and 8). This is a contradiction with the definition of α. ■

Theorem 3 *Algorithm 3 provides a lock-free universal construction.*

Proof: Lemma 1 implies that there is a total order on the operations executed in the emulated object. Through an inspection of the algorithm, it is easy to see that a process updates its copy of the state of the emulated object by applying the deterministic function $apply_T$ to all SEQ tuples in the order defined by the sequence number. In this way, all operations are executed in the same order by all correct processes, and this order is according to the sequential specification of the object provided by the function $apply$. This proves that the universal construction satisfies linearizability. Lemma 2 proves that the construction is lock-free. ■

6. Related Work

In this paper we present several shared memory algorithms that tolerate Byzantine faults using an augmented tuple space. To the best of our knowledge, the only other works which use this type of object to resolve fundamental distributed computing problems are [5, 21]. However these works address only the wait-free consensus problem in fail-stop systems (no Byzantine failures).

Asynchronous shared memory systems with processes that can fail in a Byzantine way have been first studied independently by Attie [3] and Malkhi et al. [15]. The work in [3] shows that weak consensus cannot be solved using only resettable objects[5]. This result implies that algorithms for solving consensus in this model must use some kind of persistent (non-resettable) object like sticky bits. The PEATS used in our algorithms can be viewed as a persistent object since the specified access policies do not allow processes to reset the state of the object.

The work presented in [15] uses shared memory objects with ACLs to define a t-threshold strong binary consensus algorithm and a t-resilient universal construction. The former uses $2t+1$ sticky bits and requires $n \geq (t+1)(2t+1)$ processes. The paper also shows that there can be no strong binary consensus algorithm with $n \leq 3t$ processes in this model of computation.

In a more recent work, Alon et al. [1] extend previous results by presenting a strong binary consensus algorithm that attains optimal resiliency ($n \geq 3t+1$) using an exponential number of sticky bits and requiring also an exponential number of rounds. That work proves several lower bounds related to the number of objects required to implement consensus, including a tight trade-off characterizing the number of objects required to implement strong consensus: a polynomial number of processes needs an exponen-

[5] An object o is **resettable** if, given any of its reachable states, there is a sequence of operations that can return the object back to its initial state [3].

tial number of objects and vice-versa. This result emphasizes the power of ACLs in limiting malicious processes but also shows the limitations of this model, specially in terms of the large number of objects required to attain optimal resilience. The approach proposed in the present paper uses a different model so this trade-off does not apply.

The type of policy enforcement used in this paper was inspired by the LGI (Law-Governed Interaction) approach [18] and its use in protecting centralized tuple spaces [17].

7. Concluding Remarks

The approach for distributed computing with shared memory accessed by Byzantine processes presented in this paper differs from the previous model where objects are protected by access control lists. Our approach is based on the use of fine-grained access policies that specify rules that allow or deny an operation invocation to be executed in an object based on the arguments of the operation, its invoker, and the state of the object. The constructions presented in this paper (consensuses and universal object) demonstrate that this approach allows the development of simple and elegant algorithms

An inherent characteristic of the proposed approach is that its utility is limited when used to implement simple persistent objects like sticky bits (its use would be equivalent to using ACLs). The full potential of PEOs appears when more elaborated objects such as augmented tuple spaces are considered.

Acknowledgements

We warmly thank Isabel Nunes for her assistance with the formalism of the policies. This work was supported by CNPq (Brazilian National Research Council) through processes 200569/2005-8 and 550114/2005-0, the EU through project IST-4-027513-STP (CRUTIAL) and by the FCT through project POSI/EIA/60334/2004 (RITAS).

References

[1] N. Alon, M. Merrit, O. Reingold, G. Taubenfeld, and R. Wright. Tight bounds for shared memory systems accessed by Byzantine processes. *Distributed Computing*, 18(2):99–109, Nov. 2005.

[2] J. P. Anderson. Computer security technology planning study. ESD-TR 73-51, U. S. Air Force Electronic Systems Division, Oct. 1972.

[3] P. C. Attie. Wait-free Byzantine consensus. *Information Processing Letters*, 83(4):221–227, Aug. 2002.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 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. da Silva Fraga, and L. C. Lung. Towards a dependable tuple space. DI/FCUL TR 06–4, Department of Informatics, University of Lisbon, March 2006.

[7] A. N. Bessani, J. da Silva Fraga, and L. C. Lung. BTS: A Byzantine fault-tolerant tuple space. In *Proceedings of the 21st ACM Symposium on Applied Computing - SAC 2006*, Apr. 2006.

[8] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2002*, Washington, DC, USA, June 2002.

[9] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

[10] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, Oct. 2004.

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

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

[13] 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.

[14] R. K. Joshi, N. Vivekananda, and D. J. Ram. Message filters for object-oriented systems. *Software - Practice and Experience*, 27(6):677–699, June 1997.

[15] D. Malkhi, M. Merrit, M. Reiter, and G. Taubenfeld. Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37–48, Feb. 2003.

[16] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, Apr. 2000.

[17] 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*, pages 218–226, Mar. 2000.

[18] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[19] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, June 1989.

[20] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the ACM Symposium on Operating System Principles*, Oct. 1973.

[21] 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.