Sharing Memory between Byzantine Processes Using Policy-Enforced Tuple Spaces

Alysson Neves Bessani, Miguel Correia, *Member*, *IEEE*,
Joni da Silva Fraga, *Member*, *IEEE*, and Lau Cheuk Lung

Abstract—Despite the large amount of Byzantine fault-tolerant algorithms for message-passing systems designed through the years, only recent 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 require less shared memory operations, using also less memory bits than previous algorithms based on access control lists (ACLs) and sticky bits. We also prove that PEATS objects are universal, i.e., that they can be used to implement any other shared memory object, and present lock-free and wait-free universal constructions.

Index Terms—Byzantine fault-tolerance, shared memory algorithms, tuple spaces, consensus, universal constructions.

1 Introduction

Despite the large amount of Byzantine fault-tolerant algorithms for message-passing systems designed through the years (e.g., [1], [2], [3], [4], [5], [6], [7], and [8]), only recent algorithms for the *coordination of processes subject to Byzantine failures using shared memory* have appeared [9], [10], [11]. This line of research complements the current availability of several solutions for the construction of dependable *services* on message-passing distributed systems subject to Byzantine failures [1], [2], [3], [4], [6], [8]. These services can be seen as *shared memory objects* emulated over message-passing systems, and the clients that access the services can be seen as the processes accessing the shared memory. The motivation for this research is to answer a fundamental question: what is the power of these shared memory objects to coordinate processes that can fail in a Byzantine way, i.e., arbitrarily [11]? From a more practical point of view, we are interested in knowing if it is possible to elect a leader among these processes or to solve fundamental problems like consensus or mutual exclusion even if some processes are faulty. A complementary question is—how costly is it to solve these problems in terms of resilience and shared memory bits and operations required? These questions are especially relevant since Byzantine failures can be used to model the behavior of malicious hackers and malware [12].

- *A.N. Bessani and M. Correia are with the Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Bloco C6 Piso III, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: {bessani, mpc}@di.fc.ul.pt.*
- *J. da Silva Fraga is with the Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, DAS/UFSC, CP 476-88.040-900 Florianópolis, SC, Brazil. E-mail: fraga@das.ufsc.br.*
- *L.C. Lung is with the Departamento de Informática e Estatistica, Universidade Federal de Santa Catarina, INE/UFSC, CP 476-88.040-900 Florianópolis, SC, Brazil. E-mail: lau.lung@inf.ufsc.br.*

Manuscript received 6 Nov. 2007; revised 14 Apr. 2008; accepted 16 May 2008; published online 28 May 2008.
Recommended for acceptance by A. Pietracaprina.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-11-0408.
Digital Object Identifier no. 10.1109/TPDS.2008.96.

The first works in this area made several important theoretical contributions. They have shown that simple objects like registers and sticky bits [13] when protected by access control lists (ACLs) are enough to solve consensus [9], that the optimal resilience for strong consensus is $n \geq 3t + 1$ in this model [9], [11] (t is an upper bound on the number of faulty processes and n is 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 [11], 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 and registers with ACLs: the amount of objects required and the amount of requested operations in these objects are 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 set up ACLs, associating objects with processes in such a way that faulty processes cannot invalidate the actions of correct processes [9].

This paper contributes to advancing the study of Byzantine shared memory by modifying this model in two aspects. First, this 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 current state of the object. We call the objects protected by these policies *policy-enforced objects* (PEOs).

Second, this paper uses only one type of shared memory object: an *augmented tuple space* [14], [15]. This object, which is an extension of the tuple space introduced in LINDA [16], stores generic data structures called tuples. It provides operations for the inclusion, removal, reading and conditional inclusion of tuples.

This paper shows that *policy-enforced augmented tuple spaces* (PEATS) are an attractive solution for the coordination of Byzantine processes. This paper provides algorithms that are much simpler than previous ones based on sticky bits and ACLs [9], [11]. They are also more efficient in terms of number of bits, objects, and operations 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 when considering shared memory emulation over message-passing [17]. For instance, both can be implemented similarly using the aforementioned Byzantine fault-tolerant systems based on state machine replication [2], [3], [4].

The results presented have two main consequences on the broad Byzantine fault tolerance research area. First, they show that a well-designed shared memory object makes it much easier to program synchronization protocols in the asynchronous Byzantine fault model. Second, they show that fine-grained policy enforcement is a much more efficient model for protecting dependable services/objects than ACLs, which are the standard mechanism used to protect Byzantine fault-tolerant objects from faulty clients that access them.

1.1 Summary of the Contributions

The main contributions of this paper are the following:

- we present a new computing model where shared memory objects are protected by fine-grained access policies;
- we present a new shared memory object, the PEATS;
- we show the benefits of this model by providing simple and efficient consensus algorithms with resilience $n \geq 3t + 1$, and we prove that this is the optimal resilience for strong binary consensus in our system model; our strong binary consensus algorithm uses only $O((n + t) \log n)$ bits as compared to the $(n + 1)\binom{2t+1}{t}$ sticky bits of the algorithm in [9];
- we show also how the PEATS can be used to solve multivalued strong consensus and default multivalued consensus in our model;
- we prove that PEATS are universal [18], i.e., that they can be used to implement any other shared memory object, by providing two universal constructions based on PEATS: a uniform lock-free construction and a wait-free construction. The wait-free construction is the first for a model in which the memory is shared by Byzantine processes (there is only one previous universal construction for this case and it is t-resilient, not wait-free [11]).

1.2 Paper Organization

This paper is organized as follows: Section 2 presents the system model and the augmented tuple space. PEOs are presented in Section 3. Some details about the feasibility of the PEATS are presented in Section 4. Section 5 presents consensus algorithms based on this object. Section 6 provides the two universal constructions based on a PEATS object. Finally, Sections 7 and 8 summarize related work and present conclusions.

2 Model and Definitions

2.1 System Model

The model of computation consists of an asynchronous set of n processes $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ that communicate via a set of k shared memory objects $\mathcal{O} = \{o_1, \ldots, o_k\}$ (e.g., registers, sticky bits, and 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 [7], can deviate arbitrarily from it. In the same way as previous works on Byzantine shared memory [9], [10], [11], 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 [19] to enforce the access policy (see Section 4). This monitor must know the correct identity of the process invoking operations on the object in order to grant or deny access for the invocation. It is worth to notice that without authenticated access to shared memory, it is impossible to implement access control, and thus, it is impossible to solve any nontrivial problem in the Byzantine asynchronous setting since a faulty process can always write invalid/inconsistent values to the memory.

A *configuration* of a shared memory distributed system with n processes communicating using k shared memory objects is a vector $C = \langle q_1, \ldots, q_n, r_1, \ldots, r_k \rangle$, 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 their last operation on this object. This condition is sometimes called *well formedness* or *correct interaction* [17].[1]

The shared memory objects used in this paper are assumed to be dependable (they do not deviate from their specification) and to satisfy the *linearizability* correctness condition [20]: 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 concurrent operations appear to be executed sequentially.

1. This is just a simplification to improve the presentation of the algorithms. The enforcement of this assumption can be easily implemented making the objects ignore invocations made by processes that have pending invocations.

2.2 Termination Conditions

In terms 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 $x.\mathrm{op}$ 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 $x.\mathrm{op}$ or any of the pending operations) will be completed;
- *t-resilience* [11]: an operation $x.\mathrm{op}$ 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* [11]: an operation $x.\mathrm{op}$ 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 $x.\mathrm{op}$; and
- *wait-freedom* [18]: an operation $x.\mathrm{op}$ is wait-free if, when executed by a correct process, it eventually completes in any execution (despite the failure of other processes).

The main difference between t-threshold and t-resilience is the fact that 1) an operation is guaranteed to complete only if $n-t$ correct processes invoke the *same* operation and 2) 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.

2.3 Augmented Tuple Space

The *tuple space* coordination model, originally introduced in the LINDA programming language [16], allows distributed processes to interact through a shared memory object called a tuple space, where generic data structures called *tuples* are stored and retrieved.

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., $\overline{t}$). 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 $\overline{t}$ *match*, denoted $m(t,\overline{t})$, iff 1) they have the same type and 2) all defined field values of $\overline{t}$ 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 [16]: $out(t)$, which outputs the entry t in the tuple space (write); $in(\overline{t})$, which removes a tuple that matches $\overline{t}$ from the tuple space (destructive read); and $rd(\overline{t})$, which is similar to $in(\overline{t})$ but does not remove the tuple from the space (nondestructive 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 nonblocking 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 using addresses. This feature leads to a simple programming model where more expressive interactions can be described with very few lines of code.

In Herlihy's hierarchy of shared memory objects [18], the tuple space object has consensus number 2 [15], 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; therefore, we need universal shared memory objects (consensus number n) [18], [17]. Therefore, we use an *augmented tuple space* [14], [15] which provides an extra *conditional atomic swap* operation. This operation, denoted by $cas(\overline{t},t)$ for a template $\overline{t}$ and an entry t, works like an atomic (indivisible) execution of the instruction:

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

The meaning of this instruction is "if the reading of $\overline{t}$ fails, insert the entry t in the space."[2] This operation returns $true$—we say it *succeeded*—if the tuple is inserted in the space, and $false$ otherwise. The augmented tuple space is a universal shared memory object, since it can solve wait-free consensus trivially in the crash fault model [14], [15] 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 [9], [10], [11]. 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. This model requires a kind of reference monitor [19] to protect 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 [1], [2], [3], [4], [6], [8], 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 PEOs, which are objects whose access is governed by a fine-grained security policy. Later, we argue that the use

2. Notice that the meaning of the tuple space cas is the opposite of the well-known register *compare&swap* operation [17], where the object state is modified if its current state is *equal* to the value compared.

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

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

of these policies makes 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 $execute(\mathrm{op})$ 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 [21]. 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 $invoke(p, \mathrm{op})$ to a protected object x can be executed:

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

An example of a PEO 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 Fig. 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 side) indicates if the operation was invoked.

In the access policy in Fig. 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 1) p is one of the processes in the set $\{p_1, p_2, p_3\}$ and 2) the value v being written is greater than the current value of the register r. Notice that condition 1 is nothing more than a straightforward implementation of an ACL in our model.

4 Policy-Enforced Augmented Tuple Space

The algorithms presented in this paper are based on a PEATS object. The implementation of this kind of object (or another PEO in general) on distributed message-passing systems could be based on interceptors [22], that would grant or deny access to the operations according to the access policy defined for the tuple space and the identity of the client, which is available due to the use of authenticated channels (implemented using standard technologies like IPSec or SSL). A straightforward resilient implementation would be to replicate the PEATS in a set of servers, e.g., using the CL-BFT library [3] or any other Byzantine fault-tolerant state machine replication support [23]. The interceptor would be deployed in every PEATS replica to make possible the local enforcement of policies, working as a reference monitor. The access policy could be hard-coded in the interceptor, or a more generic policy enforcer system like the one presented in [24] might be used. Fig. 2 illustrates this design.

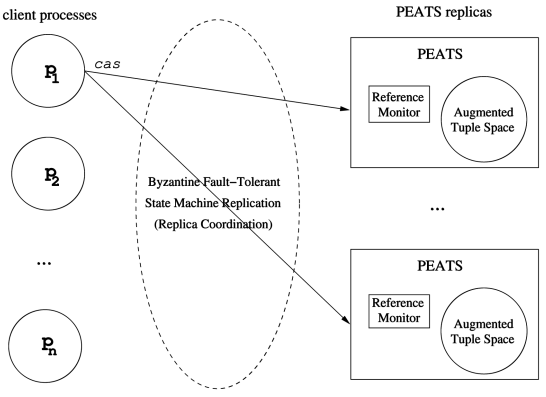

Fig. 2. A Byzantine fault-tolerant PEATS implementation.

In this figure, it can be seen that the set of processes that access the fault-tolerant replicated PEATS do it using a replica coordination protocol which ensures that all requests are executed in all PEATS replicas in the same order (usually through an atomic multicast protocol, which is known to be equivalent to consensus [25]). This, together with the fact that both the augmented tuple space and the reference monitor are deterministic objects (i.e., their outputs depend only on their previous state and the operation issued by the client), basic voting protocols can be executed by the processes to determine the operation results. The DEPSPACE system [26] is a complete implementation of a PEATS that follows the architecture described in Fig. 2.

It is worth to notice that any fault-tolerant implementation of a PEATS requires consensus for replica coordination simply because this object has consensus number greater than one. Otherwise, the FLP result [27] would not be valid: a distributed system that would solve consensus in asynchronous systems (the PEATS fault-tolerant implementation) would be built without solving consensus. Given this note, a question that can be made is—why build synchronization algorithms based on fault-tolerant objects built using other synchronization algorithms? There are two answers for this question, a theoretical and a practical one. The theoretical one is that, in this paper, as well as in the previous ones in this field [11], [9], consensus and universal constructions are used as reference problems to determine the power of the model and objects used to solve it. The practical answer is that, for many applications, it is much simpler to develop a resilient object/service built on top of a fixed set of "controlled" servers, and make this object available to be used by an open and unknown set of processes that need to coordinate between themselves.

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

Fig. 3. Access policy for the PEATS used in Algorithm 1.

5 Solving Consensus

In this section, we illustrate the benefits of using a PEATS to solve several variants of the consensus problem.

The consensus problem concerns a set of n processes proposing values from a set $\mathcal{V}$ of possible values 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 three kinds of consensus objects (or, to solve three consensus variants):

- *Weak consensus* [11]. A weak consensus object x is a shared memory object with a single operation $x.\text{propose}(v)$, with $v \in \mathcal{V}$, satisfying the properties: (*Agreement*) in any execution, $x.\text{propose}$ 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 $x.\text{propose}(v)$.
- *Strong consensus* [11]. 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 $x.\text{propose}(v)$.

Another variant of consensus that we define and implement in this paper is the default (multivalued) consensus [5], which is a slightly weakened version of strong consensus:

- *Default consensus.* A default consensus object x is defined by a weaker Validity condition than strong consensus objects: (*Default Strong Validity*). The consensus value must satisfy two conditions: 1) if all correct processes invoke $x.\text{propose}(v)$, then v is the consensus value and 2) if the consensus value is v, then some correct process invoked $x.\text{propose}(v)$ or $v = \perp$.

The idea behind default consensus is that the consensus value should be a value proposed by some correct process or a default value $\perp \notin \mathcal{V}$ [5]. This idea is related to the *quittable consensus* problem [28]. In this problem, a process can decide a "quit" value (Q) when some failure is detected. In the default consensus problem, the default value $(\perp)$ can be decided even in executions without faulty processes if not enough processes propose the same value.

We remark that all these objects, as all other objects used in this paper, must satisfy some of the termination conditions given in Section 2.2.

5.1 Weak Consensus

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 object (process p_i).
Shared variables:
1: $ts = \emptyset$ {PEATS object}
procedure $x.\text{propose}(v)$
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 returned. In the former case, line 3 is executed, setting variable d to the decision value, while in the latter, it is the cas operation that sets d.

The access policy for the PEATS used in Algorithm 1 is presented in Fig. 3. 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* [17], i.e., it works for any number of processes and the processes do not need to know how many other processes are participating in the distributed computation. 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 occurrence of failures of any number of processes running it.

An interesting point about this algorithm is that our PEATS with the access policy specified in Fig. 3 behaves like a persistent object, so our result is in accordance with [11, Theorem 4.1].

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

Proof. From the access policy, we know that the only way to insert a tuple in the space is by invoking the cas operation. This operation can be executed successfully only once since there is no way to remove a tuple from the space (operations in and inp are not allowed). This way, the Agreement property must be satisfied since the first process that successfully executes the cas operation will insert a DECISION tuple with its consensus value v in the space. Other processes will read v (through the formal field $?d$) since their invocation of the cas operation will return $false$ (the DECISION tuple will not be inserted in the space).

The Validity property holds because, in any execution with only correct processes, the consensus value must have been proposed by some process (the one that inserted the DECISION tuple with its proposal in the space). The algorithm is wait-free because the cas operation is wait-free. □

5.2 Strong Consensus

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:

- *Nonuniform.* The strong consensus algorithm 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.
- *Binary consensus.* Our algorithm solves only binary consensus. This limitation is also due to the fact that a process needs to know if a value has been proposed by one correct process before deciding it.
- *t-threshold object.* The algorithm for strong consensus is not wait-free since it requires $n - t$ processes to take part in the algorithm. However, the number of processes needed in our algorithm is optimal: $n \geq 3t + 1$ (see Corollary 1 in the next section).

Algorithm 2: Strong Byzantine consensus object (process p_i).
Shared variables:
1: $ts = \emptyset$ {PEATS object}
procedure $x.\text{propose}(v)$
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** $p_j \in \mathcal{P} \setminus (S_0 \cup S_1)$ **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

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 a single decision value will be defined. All further invocations of cas return this value (lines 12-14).

The access policy for the PEATS used in Algorithm 2 is presented in Fig. 4. 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

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

Fig. 4. Access policy for the PEATS used in Algorithm 2.

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 could easily be implemented in practice effectively constrain the power of Byzantine processes, thus allowing the simplicity of the consensus presented in 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 [9] requires $(n + 1)\binom{2t+1}{t}$ sticky bits.[4]

Theorem 2. *Algorithm 2 provides a t-threshold strong binary consensus object if $n \geq 3t + 1$.*

Proof. From the access policy, we know that the only way to insert a DECISION tuple in the space is by invoking a cas operation. This operation can be invoked successfully (returning $true$) only once, since neither an inserted tuple can be removed (the operations in and inp are not allowed by the policy), nor two decision tuples can be inserted (the second field of the template of cas must be formal). In any execution of the algorithm, the first process that executes cas after reading $t + 1$ PROPOSE tuples with the same value v will manage to insert a DECISION tuple with v (if it satisfies the rule R_{cas}), thus making this the decision value (lines 13 and 15). The Agreement property is always satisfied since the value v associated with the DECISION tuple in the space will be read by all correct processes that do not succeed in the cas operation, i.e., all that receive $false$ in reply. Their decision values will be v (lines 12 and 15).

The algorithm satisfies also Strong Validity since the DECISION tuple can only be inserted if its value v is justified by a set of $t + 1$ processes (at least one correct) that proposed v. This condition is enforced by the rule R_{cas} of the access policy.

In terms of termination conditions, our algorithm is a t-threshold protocol. This property is satisfied since for a process to decide a value v, this value must have been proposed by $t + 1$ processes. Assuming $n \geq 3t + 1$, it can be easily shown that if $n - t$ correct processes (at least $2t + 1$) invoke $x.\text{propose}$ with some value $v' \in \{0, 1\}$, there will be always at least $t + 1$ PROPOSE tuples for some value (0 or 1) and one process will insert a justified DECISION tuple in the space. Since the cas operation is wait-free, the algorithm terminates. □

3. For example, 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 1, 764 sticky bits.

5.3 Strong Multivalued Consensus

A strong multivalued consensus can be obtained with little modifications to the strong binary consensus algorithm following the same ideas in [9]. In fact, if we consider a *k-valued consensus* problem, in which there are k possible inputs for processes to propose,[5] i.e., $|\mathcal{V}| = k$ (binary consensus is a 2-valued consensus), we can use the same algorithm and collect different proposition values in different sets S_v, with $v \in \mathcal{V}$ and $|\mathcal{V}| = k$. The algorithm works exactly in the same way as Algorithm 2: a process proposes its value and keeps reading the values of other processes until there is some value that was proposed by $t+1$ processes (some correct process proposed it). This value will be assumed as a possible decision.

Unfortunately, this algorithm requires more processes to resist t faulty processes, as shown by the following theorem:

Theorem 3. *The algorithm implements a* t-threshold *strong* k-valued *consensus object if* $n \geq (k+1)t + 1$.

Proof. The proofs for Agreement and Validity are similar to the proof of Theorem 2. Now, suppose the worst possible execution for a system running the algorithm described above: each of the k possible values is proposed by t processes and t faulty processes do not propose (they crash or stay silent during the whole execution). To guarantee the termination of the algorithm, we need one process to break the tie of t proposals for each value. Consequently, we need $n \geq kt + t + 1 = (k+1)t + 1$. □

A direct consequence of this theorem is that the number of processes needed to solve strong k-valued consensus in the presence of Byzantine faults using the described algorithm is always $n > k$. This result rules out the possibility of using this algorithm in applications where every process must propose some process identifier to a consensus. Examples of such applications are consensus-based mutual exclusion [8] and leader-election [17].

Our strong multivalued consensus algorithm requires only $O(n(\log n + \log |\mathcal{V}|))$ bits of shared memory.

The following theorem proves that $n \geq (k+1)t + 1$ is the minimum number of processes needed to solve the k-valued strong consensus problem tolerating t Byzantine faults.

Theorem 4. *The* k-valued *strong consensus problem can only be solved in an asynchronous system in which the processes communicate through PEOs and at most* t *processes can be faulty if the number of processes is* $n \geq (k+1)t + 1$.

Proof. Theorem 3 proves the existence of an algorithm with this resilience. We have to prove now that there is no algorithm that solves the k-valued strong consensus problem with $n \leq (k+1)t$. Assume that there is an algorithm A that solves this problem with this number of processes. We will present an execution in which A does not terminate.

Let $\mathcal{V} = \{v_1, \ldots, v_k\}$ be the domain of k values. Suppose an execution α of A in which no faulty process participates in the distributed computation (t processes stay silent) and each of the k values of $\mathcal{V}$ is proposed by at most t correct processes.

Independently of the shared memory object(s) used by A and their access policy(s), to satisfy the strong consensus Validity property, a correct process can only consider a value for decision if it knows that this value was proposed by at least $t+1$ processes. If this condition is not true, it is very easy to show an execution of A in which a correct process will decide a value proposed only by faulty processes (due to the system absence of synchrony), violating the Validity property.

Turning back to the execution α, the system will reach a configuration in which all correct processes will have read at most t proposals for each one of the k values and cannot read more proposals (there are no more processes, since all correct processes—at most kt—already proposed and the t faulty ones will stay silent). Consequently, no value will be proposed by at least $t+1$ processes and the algorithm will not terminate. This means that there cannot be an algorithm A that solves the k-valued strong consensus problem with $n \leq (k+1)t$. □

Given this theorem, we can define the optimal resilience for strong binary consensus.

Corollary 1. *The optimal resilience for the strong binary consensus problem in asynchronous systems where the processes communicate using PEOs is* $t = \lfloor \frac{n-1}{3} \rfloor$ *of* n *processes.*

Proof. The proof that such algorithm exists is given by Theorem 2. The proof that this resilience is optimal is a direct consequence of Theorem 4 if we take $k = 2$. □

5.4 Default Multivalued Consensus

A default multivalued consensus object can be obtained by making some simple modifications to the strong binary consensus in Algorithm 2. The objective here is to show that a PEATS allows to solve multivalued consensus stronger than weak consensus with optimal resilience, i.e., with $n \geq 3t + 1$.

The required modifications to Algorithm 2 are the following:

- There has to be one set S_v for every different value v in a tuple $\langle \text{PROPOSE}, *, v \rangle$ obtained from the PEATS in line 7 (instead of only S_0 and S_1).
- After a process reads $n - t$ proposed values, if there is no value v proposed by at least $t+1$ processes, the value to be put in the DECISION tuple is $\perp$.
- If the value put in the DECISION tuple is $\perp$, the third field in the DECISION tuple has to be a set with all the sets S_v filled in line 8.

In order to rule out the possibility of malicious processes forcing the consensus value to be always $\perp$, we have to ensure that the default value is put in the PEATS (using cas) by a process p only if this process has read $n - t$ PROPOSE tuples and none of the values was proposed by $t+1$ processes. This condition is enforced by the access policy in Fig. 5.

There are two main differences between this policy and the one used for strong consensus (see Fig. 4). First, all proposed values must be different from $\perp$ (rule R_{out}). Second, the rule R_{cas} now states that if p wants to output a DECISION tuple in the PEATS with $v = \perp$, it has to show that it did not find a value proposed by $t+1$ processes. More precisely, the rule states that if the second argument of the cas operation

5. This problem is completely different from the well-known *k-set consensus*, where the consensus value of the processes can be different, but belonging to a set of k values [29].

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

Fig. 5. Access policy for the PEATS used in the default multivalued consensus.

executed by p (Algorithm 2, line 12) takes $v = \perp$, then the third argument has to contain a set of sets S_v satisfying the following conditions: 1) the union of all sets S_v must contain at least $n-t$ processes, 2) no set S_v can have more than t processes, and 3) all processes q in all sets S_v must correspond to a PROPOSE tuple $\langle \text{PROPOSE}, q, v\rangle$ in TS.

Theorem 5. *The algorithm implements a t-threshold default multivalued consensus object if $n \geq 3t+1$.*

Proof. The Agreement property is satisfied due to the access policy in Fig. 5 that does not allow two different DECISION tuples to be inserted in the space. The two conditions of the Default Strong Validity property are also satisfied:

1. If all correct processes invoke $x.\text{propose}(v)$ with the same value v, then a different value v' can be proposed by at most other t processes, and clearly, the cas operation of the PEATS will not allow the insertion of a DECISION tuple with v'. A malicious process will also not be allowed to insert a DECISION tuple with $\perp$ due to the last two lines of the rule R_{cas}.
2. If a value $v \neq \perp$ is decided, it must have been proposed by at least $t+1$ processes (one of which is correct). □

6 Universal Constructions

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 PEATS are *universal objects* [18], 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 [18].

The proof that PEATS are universal is made by providing two universal constructions 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 [18]. There are several wait-free universal constructions for the crash fault model, using consensus objects [18], sticky bits [13], compare and swap registers [17], and several other universal objects. A universal construction for the Byzantine fault model using sticky bits was defined in [11]. However, this construction is not wait-free but only t-resilient.

In order to define a universal construction that emulates a deterministic object o, 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 the 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, $apply_T(S_i, inv)$ gives a new state S_j (the result of the execution of inv in state S_i) and a reply $reply$ for the invocation. This definition is enough for showing the universality of tuple spaces, although Malkhi et al. have shown that a simple generalization is needed for emulating nondeterministic types and some objects that satisfy weak liveness guarantees [11].

In this section, we present a simple nonblocking universal construction, which shows the power and simplicity of our PEATS objects. Then, we present a wait-free universal construction.

6.1 Uniform Lock-Free Universal Construction

Our lock-free universal construction follows previous constructions [17], [18]. 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 essentially boils down to the definition of 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. Fig. 6 illustrates the main idea.

In this figure, the process p_1 tries to thread a tuple containing an invocation $inv1$ with sequence number 3 in the PEATS, while process p_2 executes cas trying to insert an invocation $inv2$ with sequence number 5. In the PEATS, there are tuples with sequence numbers from 1 to 4, so process p_1

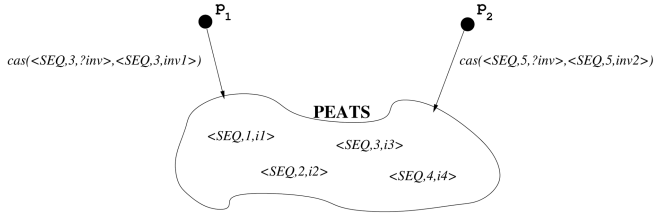

Fig. 6. PEATS-based universal construction.

will not insert its tuple and process p_2 will have success in its insertion. Algorithm 3 presents this universal object.

Algorithm 3: Lock-free universal construction (process p_i).
Shared variables:
1: $ts = \emptyset$ {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: $pos \leftarrow pos + 1$
6: **if** $ts.cas(\langle \text{SEQ}, pos, ?einv \rangle, \langle \text{SEQ}, pos, inv \rangle)$ **then**
7: $\langle state, reply \rangle \leftarrow apply_T(state, inv)$
8: **return** $reply$
9: **end if**
10: $\langle state, reply \rangle \leftarrow apply_T(state, einv)$
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 the $state$ 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 $state$ variable 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. A very interesting property of this algorithm is that it is *uniform*: processes executing operations on the emulated object do not need to know each other. This means that this algorithm works even with an unknown and dynamic set of processes.

The access policy for our universal construction (Fig. 7) 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 $pos - 1$. No other operations are allowed.

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 $\langle \text{SEQ}, pos, inv \rangle$ in the tuple space.*

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

Fig. 7. Access policy for the PEATS used in Algorithm 3.

2. *For any tuple $\langle \text{SEQ}, pos, inv \rangle$ in the tuple space with $pos > 1$, there is exactly one tuple $\langle \text{SEQ}, pos - 1, inv \rangle$ in the space.*

Proof. These two invariants follow directly from Algorithm 3 and its access policy (Fig. 7):

1. From the access policy, we can see that a tuple can be inserted in the space only by a cas operation, which is executed with a template and an entry with the same sequence number pos, and with a template invocation field (x) that is formal. With that property and the behavior of the cas operation, it becomes clear that there can never be two SEQ tuples with the same sequence number in the tuple space.
2. From the access policy, it is possible to see that a cas operation can be executed by trying to insert a tuple in position pos only if there is a SEQ tuple in the space with position $pos - 1$. This guarantees that there is one tuple $\langle \text{SEQ}, pos - 1, inv \rangle$ in the space. That there is no more than one is a direct consequence of the first part of the lemma. □

Lemma 2. *The universal construction of the 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) that 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 6. *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 $apply_T$. This suffices for proving that the universal construction satisfies linearizability. Lemma 2 proves that the construction is lock-free. □

6.2 Wait-Free Universal Construction

The wait-free universal construction follows the same basic idea as the previous construction of building a list of operations to be executed in the emulated object. However, here, we need a helping mechanism that allows a process to thread an operation even if in contention with $n-1$ faulty processes. This mechanism works as follows: When a process wants to thread an operation, it inserts an ANN (announcement) tuple with the invocation it wants to execute on the emulated object. After this insertion, the invocation is said to be *announced*. For each position of the invocation list, there is a *preferred process* for the position. The preferred process for position pos is p_i such that $i = pos \mod n$. If the preferred process for a position has an invocation that is announced but not threaded, then the policy of the space does not permit any other invocation to be threaded in that position. The invocation can be threaded either by the process or by any other process willing to "help" it. A consequence of the use of this mechanism is that the algorithm is not uniform: processes must be aware of the ID of each other in order to help.

Algorithm 4 presents this universal construction. For simplicity, it assumes there are no two identical invocations, something that can be trivially enforced by adding a unique timestamp to the invocation (including the invoker's identification).

Algorithm 4: Wait-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: ts.out(⟨ANN, i, inv⟩)
 5: repeat
 6:    pos ← pos + 1
 7:    preferred ← pos mod n
 8:    if ¬ts.rdp(⟨SEQ, pos, ?einv⟩) then
 9:       if (i≠preferred) ∧ ts.rdp(⟨ANN, preferred, ?tinv⟩)
          then
10:          if ts.rdp(⟨SEQ, *, tinv⟩) then
11:             tinv ← inv
12:          end if
13:       else
14:          tinv ← inv
15:       end if
16:       if ts.cas(⟨SEQ, pos, ?einv⟩, ⟨SEQ, pos, tinv⟩) then
17:          einv ← tinv
18:       end if
19:    end if
20:    ⟨state, reply⟩ ← apply_T(state, einv)
21: until einv = inv
22: ts.inp(⟨ANN, i, inv⟩)
23: return reply
```

Each process p_i begins its execution with the initial state of the emulated object ($state = S_T$, line 2) and an empty list ($pos = 0$, line 3). When an operation (denoted by inv) is invoked on the emulated object, p_i first announces its invocation with an ANN tuple and then iterates through the list updating its $state$ variable and trying to thread inv (lines 5-21). When inv is executed ($einv = inv$), p_i removes the announcement tuple from the PEATS and returns the reply for the invocation (lines 20-23).

The most important part of the algorithm is the loop in lines 5-21. This loop has two main parts: the verification if the preferred process needs help (lines 8-15) and the threading of an invocation (lines 16-18). The main idea is to find an *invocation to be threaded* (stored in variable $tinv$) and try to insert it in the PEATS in a SEQ tuple. The *invocation to be executed* is stored in variable $einv$. The two parts of the loop are executed for some position pos only if this position is not already occupied (line 8). Otherwise, the invocation for this position is read and executed, updating the state of the emulated object (line 20).

The "help" part of the loop works as follows: If there is an ANN tuple from the preferred process for position pos (line 9, second condition) that is not already threaded (line 10), then the invocation $tinv$ in the ANN tuple must be threaded in that position (line 11). Otherwise, inv can be threaded (line 14). Notice that p_i verifies if some process needs help only if it is not the preferred process for position pos (first condition of line 9). The operation is threaded using a cas: if there is no SEQ tuple in the position, then $einv$ is inserted in a SEQ tuple.

The access policy for this universal construction is responsible for ensuring that the total order of the list is always satisfied and the helping mechanism is respected. This policy is presented in Fig. 8.

The policy is an extension of the policy of the lock-free universal construction (see Fig. 7). There are now two simple rules to allow the insertion and removal of ANN tuples (R_{out}, R_{inp}). There are also three new lines related to ANN tuples in the rule R_{cas} (the bottom three lines). These lines enforce the behavior of the helping mechanism by defining exactly in which conditions a cas invocation can be executed (respecting the helping mechanism). One invocation can be threaded if one of the following conditions are satisfied: 1) the preferred process for the position has not announced an invocation, 2) the preferred process for the position announced an invocation but it had already been threaded, or 3) the invocation being threaded is the one that was announced by the preferred process.

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

Lemma 3. *For any execution of the system, the following properties are invariants of the PEATS used in Algorithm 4:*

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

Proof. The proof of this lemma is identical to the proof of the equivalent lemma for the lock-free construction (Lemma 1). The invariants are enforced by the policies,

Object State TS
R_{out}: $execute(out(\langle \text{ANN}, i, inv\rangle)) \ :-$
$\quad invoke(p_i, out(\langle \text{ANN}, i, inv\rangle)) \wedge$
$\quad \nexists x : \langle \text{ANN}, i, x\rangle \in TS$
R_{rdp}: $execute(rdp(\bar{t})) \ :- \ invoke(p, rdp(\bar{t}))$
R_{inp}: $execute(inp(\langle \text{ANN}, i, inv\rangle)) \ :-$
$\quad invoke(p_i, inp(\langle \text{ANN}, i, inv\rangle))$
R_{cas}: $execute(cas(\langle \text{SEQ}, pos, x\rangle, \langle \text{SEQ}, pos, inv\rangle)) \ :-$
$\quad invoke(p, cas(\langle \text{SEQ}, pos, x\rangle, \langle \text{SEQ}, pos, inv\rangle)) \wedge$
$\quad formal(x) \wedge$
$\quad (pos = 1 \vee \exists y : \langle \text{SEQ}, pos - 1, y\rangle \in TS) \wedge$
$\quad ((\nexists y : \langle \text{ANN}, pos \bmod n, y\rangle \in TS) \vee$
$\quad\ (\exists y : (\langle \text{ANN}, pos \bmod n, y\rangle \in TS \wedge$
$\quad\ \exists z : \langle \text{SEQ}, z, y\rangle \in TS)) \vee$
$\quad\ (\langle \text{ANN}, pos \bmod n, inv\rangle \in TS))$

Fig. 8. Access policy for the PEATS used in Algorithm 4.

which are similar for both constructions. The specific part of the rule R_{cas} that enforces the invariants $(invoke(p, cas(\langle \text{SEQ}, pos, x\rangle, \langle \text{SEQ}, pos, inv\rangle)) \wedge formal(x) \wedge (pos = 1 \vee \exists y : \langle \text{SEQ}, pos - 1, y\rangle \in TS))$ is identical in the policies of both algorithms (Figs. 7 and 8). □

Lemma 4. *An invocation inv made by a correct process p_i is threaded in position pos in a universal construction with $pos - 1$ threaded operations if either 1) there is no announced but not threaded invocation inv' made by process $p_j \neq p_i$, with $j = pos \bmod n$ or 2) $i = pos \bmod n$. If an invocation does not satisfy any of these two conditions, then it cannot be threaded.*

Proof. Let us prove the first part of the lemma. Consider a process p_i trying to thread inv. An invocation inv threaded in position pos is represented by the tuple $\langle \text{SEQ}, pos, inv\rangle$. The policy defined in Fig. 8 allows those tuples to be inserted in the PEATS in a single way: using the cas operation. If there are exactly $pos - 1$ threaded tuples, then there is no tuple in position pos, so process p_i will enter the "if" clause of lines 8-19. We have to prove that if one of the conditions of the lemma is satisfied, then the rule R_{cas} allows the insertion of the SEQ tuple, i.e., allows inv to be threaded:

- *Condition 1.* This condition assumes $i \neq preferred = pos \bmod n$. If there is no invocation announced by process p_j with $j = pos \bmod n$ (case 1), then the "if" condition of line 9 will not be satisfied, p_i will execute line 14 and the invocation to be threaded (variable $tinv$) will be set to inv. If there is such an announcement, but the invocation was already threaded (case 2), then both "if" conditions of lines 9 and 10 will be satisfied and the invocation to be threaded ($tinv$) will also be set to inv. Therefore, in both cases, process p_i in the cas of line 16 tries to insert the invocation inv in the position pos. This operation will succeed only if the conditions of rule R_{cas} are satisfied, but this happens in both cases:
  - case 1: $(\nexists y : \langle \text{ANN}, pos \bmod n, y\rangle \in TS)$ will be satisfied;
  - case 2: $(\exists y : (\langle \text{ANN}, pos \bmod n, y\rangle \in TS \wedge \exists z : \langle \text{SEQ}, z, y\rangle \in TS))$ will be satisfied.
- *Condition 2.* If p_i is the preferred process to thread an operation in pos ($i = preferred = pos \bmod n$), then it will execute lines 9 and 14, then execute the cas trying to insert a tuple $\langle \text{SEQ}, pos, inv\rangle$. It will succeed because, if p_i is correct, it has announced its invocation using an ANN tuple, and consequently, the conditions of rule R_{cas} will be satisfied. In particular, the last condition will be satisfied since $(\langle ANN, pos \bmod n, inv\rangle \in TS)$.

Let us now prove the second part of the lemma. If some process tries to insert a tuple without satisfying any of the conditions of the lemma, it will not succeed simply because the only way to insert a SEQ tuple is through the operation cas and the R_{cas} rule will only allow an insertion that satisfies these conditions. □

Lemma 5. *The universal construction of the Algorithm 4 is wait-free.*

Proof. This lemma is proved by contradiction. Let α be an execution with n processes in which a correct process p_i invokes an operation inv but never receives the reply. We have to show that α does not exist.

When p_i starts executing Algorithm 4, it inserts an ANN tuple for its invocation inv in the PEATS (line 4). Suppose that after the insertion of this announcement, the last SEQ tuple inserted in the space has sequence number pos. We have to consider two cases:

- If there is a configuration in α in which there is no process $p_j \neq p_i$ with an invocation that is announced but not threaded, then p_i will execute the cas successfully due to Lemma 4 (Condition 1).
- If all processes (correct or faulty) keep executing operations on the emulated object after the announcement of inv, then eventually there will be a position pos' such that all positions until $pos' - 1$ will have operations threaded and $pos' \bmod n = i$. Lemma 4 (Condition 2) guarantees that inv is threaded.

In both cases, inv is threaded, then p_i executes lines 17, 20, and 23 and returns $reply$. This fact contradicts the definition of α. □

The next theorem proves that Algorithm 4 is a wait-free universal construction.

Theorem 7. *Algorithm 4 provides a wait-free universal construction.*

Proof. Lemma 3 implies that there is a total order on the operations executed in the emulated object. An inspection of the algorithm shows 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. 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 $apply$, which satisfies linearizability. Lemma 5 proves that the construction is wait-free. □

7 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 that use this type of object to resolve fundamental distributed computing problems are [14] and [15]. However, in contrast to this paper, 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 [10] and Malkhi et al. [11]. The work in [10] shows that weak consensus cannot be solved using only resettable objects.[6] This result implies that algorithms for solving consensus in this model must use some kind of persistent (nonresettable) object like sticky bits. The PEATS used in our algorithms can be viewed as a persistent object since the access policies do not allow processes to reset the state of the object.

The work presented in [11] 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. This 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. [9] 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 required objects 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 exponential number of objects and vice-versa. This result not only emphasizes the power of ACLs in limiting malicious processes but also shows the limitations of this model, especially in terms of the large number of objects required to attain optimal resilience. The approach proposed in this paper uses a more powerful protection model than ACLs so this trade-off does not apply since our objects cannot be subverted by faulty processes.

As stated in this paper, our algorithms are much more simple and efficient than those in [11] and [9]. It could be argued that this happens because we assume a more powerful model and shared memory object (PEATS and policy enforcement instead of sticky bits and ACLs) which would be much more costly to implement. Obviously, it is more difficult to implement a PEATS than sticky bits and ACLs in hardware or at operating system level, but as argued in this paper and in previous papers in the area [11], [9], Byzantine shared memory only makes sense when considering shared memory emulation on message-passing systems[7] like, for instance, [1], [2], [3], [4], [6], and [8]. In that case, shared memory is implemented by a set of servers, so implementing the sticky bit set operation or the PEATS cas operation (both with consensus number n) requires exactly the same: a Byzantine fault-tolerant atomic multicast protocol that delivers the requests to all servers in the same order. Therefore, in terms of the costs relevant in Byzantine protocols for this emulation—time complexity, communication complexity, cryptography used—they are identical in both cases since the protocols required are also the same. Implementing simple ACLs or policy enforcement requires the same additional resource: a reference monitor deployed in each replica to verify access policies. The big difference here is that an ACL monitor only verifies if a process trying to execute an operation on an object has its ID on the operation ACL while a policy enforcement monitor has to evaluate a predicate. As can be seen in the algorithms presented in this paper, the predicates are, in general, very simple and can be implemented efficiently with little (local) processing overhead. If one takes into account the improvements of our protocols when compared with previous ones, this little extra processing is worth it.

To show the feasibility of our approach, we have implemented the DEPSPACE system [26], a complete Byzantine fault-tolerant PEATS developed in Java. Experimental results have shown that its performance is competitive with nondependable tuple space implementations.

The type of policy enforcement used in this paper was inspired by the *law-governed interaction* approach [24] and its use in protecting centralized tuple spaces [30].

8 Concluding Remarks

The proposal for distributed computing with shared memory accessed by Byzantine processes presented in this paper differs from the previous model where objects are protected by ACLs. 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 (consensus and universal objects) demonstrate that this approach allows the development of simple and elegant algorithms, at the cost of defining access policies for the shared memory objects they use.

We show that a particular type of PEO—the PEATS—is an attractive choice as support for coordinating processes that can be subjected to Byzantine failures due to its programming simplicity (few versatile operations), flexibility (can implement almost any data structure), and power (in terms of the wait-free hierarchy [18]). The combination of these advantages with fine-grained security policies allows the implementation of simple algorithms for the Byzantine fault model, especially if compared with previous solutions for this model [9], [11]. A consequence of this result is that fine-grained policy enforcement is a more adequate protection mechanism for dependable services/shared memory objects when compared with ACLs, which are considered the standard protection mechanism for these kinds of systems.

Regarding the implementation of PEATS (or any other PEO), the amount of resources required to implement these objects is the same of objects protected by ACLs previously used to solve problems in our model: a local reference

6. 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 [10].

7. The study of Byzantine faults in real shared memory systems is considered uninteresting because as these systems are implemented on tightly coupled architectures, the presence of a malicious process usually indicates that the whole system is compromised.

monitor deployed in every replica of the object implementation [26]. We remark that both the PEO model and the algorithms based on the PEATS are well suited for coordination of nontrusted processes in practical systems. We envision system models where the PEATS (or another PEO) 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. Programming synchronization primitives on these system models should be much simpler than using Byzantine fault-tolerant synchronization protocols for message-passing systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments that improved 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). A preliminary version of this paper appeared in the *26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*.

REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP '05),* pp. 59-74, Oct. 2005.

[2] C. Cachin and J.A. Poritz, "Secure Intrusion-Tolerant Replication on the Internet," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* pp. 167-176, June 2002.

[3] M. Castro and B. Liskov, "Practical Byzantine Fault-Tolerance and Proactive Recovery," *ACM Trans. Computer Systems,* vol. 20, no. 4, pp. 398-461, Nov. 2002.

[4] M. Correia, N.F. Neves, and P. Veríssimo, "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," *Proc. 23rd IEEE Symp. Reliable Distributed Systems (SRDS '04),* pp. 174-183, Oct. 2004.

[5] M. Correia, N.F. Neves, and P. Veríssimo, "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures," *The Computer J.,* vol. 49, no. 1, pp. 82-96, Jan. 2006.

[6] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance," *Proc. Seventh Symp. Operating Systems Design and Implementations (OSDI '06),* pp. 177-190, Nov. 2006.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382-401, July 1982.

[8] D. Malkhi and M. Reiter, "An Architecture for Survivable Coordination in Large Distributed Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 2, pp. 187-202, Apr. 2000.

[9] N. Alon, M. Merritt, O. Reingold, G. Taubenfeld, and R. Wright, "Tight Bounds for Shared Memory Systems Accessed by Byzantine Processes," *Distributed Computing,* vol. 18, no. 2, pp. 99-109, Nov. 2005.

[10] P.C. Attie, "Wait-Free Byzantine Consensus," *Information Processing Letters,* vol. 83, no. 4, pp. 221-227, Aug. 2002.

[11] D. Malkhi, M. Merritt, M. Reiter, and G. Taubenfeld, "Objects Shared by Byzantine Processes," *Distributed Computing,* vol. 16, no. 1, pp. 37-48, Feb. 2003.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Mar. 2004.

[13] S.A. Plotkin, "Sticky Bits and Universality of Consensus," *Proc. Eighth ACM Symp. Principles of Distributed Computing (PODC '89),* pp. 159-175, June 1989.

[14] D.E. Bakken and R.D. Schlichting, "Supporting Fault-Tolerant Parallel Programming in Linda," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 3, pp. 287-302, Mar. 1995.

[15] E.J. Segall, "Resilient Distributed Objects: Basic Results and Applications to Shared Spaces," *Proc. Seventh Symp. Parallel and Distributed Processing (SPDP '95),* pp. 320-327, Oct. 1995.

[16] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 1, pp. 80-112, Jan. 1985.

[17] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* second ed. Wiley-Interscience, 2004.

[18] M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems,* vol. 13, no. 1, pp. 124-149, Jan. 1991.

[19] J.P. Anderson, *Computer Security Technology Planning Study,* US Air Force Electronic Systems Division, ESD-TR 73-51, Oct. 1972.

[20] M. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 3, pp. 463-492, July 1990.

[21] J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE,* vol. 63, no. 9, pp. 1278-1308, Sept. 1975.

[22] R.K. Joshi, N. Vivekananda, and D.J. Ram, "Message Filters for Object-Oriented Systems," *Software—Practice and Experience,* vol. 27, no. 6, pp. 677-699, June 1997.

[23] F.B. Schneider, "Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* vol. 22, no. 4, pp. 299-319, Dec. 1990.

[24] N.H. Minsky and V. Ungureanu, "Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems," *ACM Trans. Software Eng. and Methodology,* vol. 9, no. 3, pp. 273-305, July 2000.

[25] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, Mar. 1996.

[26] A.N. Bessani, E.P. Alchieri, M. Correia, and J.S. Fraga, "DepSpace: A Byzantine Fault-Tolerant Coordination Service," *Proc. Third ACM SIGOPS/EuroSys European Systems Conf. (EuroSys '08),* pp. 163-176, Apr. 2008.

[27] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, Apr. 1985.

[28] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing," *Proc. 23rd Ann. ACM Symp. Principles of Distributed Computing (PODC '04),* pp. 338-346, July 2004.

[29] S. Chaudhuri, "More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems," *Information and Computation,* vol. 105, no. 1, pp. 132-158, July 1993.

[30] N.H. Minsky, Y.M. Minsky, and V. Ungureanu, "Making Tuple-Spaces Safe for Heterogeneous Distributed Systems," *Proc. 15th ACM Symp. Applied Computing (SAC '00),* pp. 218-226, Mar. 2000.

Alysson Neves Bessani received the BS degree in computer science from Maringá State University, Brazil, in 2001 and the MSE and PhD degrees in electrical engineering from Santa Catarina Federal University (UFSC), Brazil, in 2002 and 2006, respectively. He is a visiting assistant professor in the Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, and a member of LASIGE research unit and the Navigators research team. His research interests include distributed algorithms, Byzantine fault tolerance, coordination, middleware, and systems architecture.

Miguel Correia received the PhD degree in computer science from the University of Lisboa, Lisboa, Portugal, in 2003. He is an assistant professor in the Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, and an adjunct faculty member in the Carnegie Mellon Information Networking Institute. He is a member of the LASIGE research unit and the Navigators research team. He has been involved in several international and national research projects related to intrusion tolerance and security, including the MAFTIA and CRUTIAL EC-IST projects, and the ReSIST NoE. He is currently the coordinator of the University of Lisboa's degree on Informatics Engineering and an instructor in the joint Carnegie Mellon University and University of Lisboa MSc in Information Technology-Information Security. His main research interests include intrusion tolerance, security, distributed systems, and distributed algorithms. He is a member of the IEEE.

Joni da Silva Fraga received the BS degree in electrical engineering from the University of Rio Grande do Sul (UFRGS), in 1975, the MSE degree in electrical engineering from the Federal University of Santa Catarina (UFSC), Florianópolis, SC, Brazil, in 1979, and the PhD degree in computing science (Docteur de l'INPT/LAAS) from the Institut National Polytechnique de Toulouse/Laboratoire d'Automatique et d'Analyse des Systémes, France, in 1985. He was a visiting researcher at University of California, Irvine (UCI) from 1992 to 1993. Since 1977, he has been a research associate and later as a professor in the Departamento de Automação e Sistemas, UFSC. His research interests include distributed systems, fault tolerance, and security. He has more than 100 scientific publications. He is a member of the IEEE and the Brazilian scientific societies.

Lau Cheuk Lung received the PhD degree in electrical engineering from the Universidade Federal de Santa Catarina (UFSC), Florianópolis, SC, Brazil, in 2001. He is an associate professor in the Departamento de Informática e Estatistica (INE), UFSC. He is currently conducting research in fault tolerance, security in distributed systems, and middleware. From 2003 to 2007, he was an associate professor in the Department of Computer Science, Pontifical Catholic University of Paraná, Brazil. From 1997 to 1998, he was an associate research fellow at the University of Texas at Austin, working in the Nile Project. From 2001 to 2002, he was a postdoctoral research associate in the Department of Informatics, University of Lisbon, Portugal.