Decoupled Quorum-Based Byzantine-Resilient Coordination in Open Distributed Systems

Alysson Neves Bessani† Miguel Correia† Joni da Silva Fraga‡ Lau Cheuk Lung§
† LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
‡ Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
§ Prog. de Pós-Grad. em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brazil

Abstract

Open distributed systems are typically composed by an unknown number of processes running in heterogeneous hosts. Their communication often requires tolerance to temporary disconnections and security against malicious actions. Tuple spaces are a well-known coordination model for this sort of systems. They can support communication that is decoupled both in time and space. There are currently several implementations of distributed fault-tolerant tuple spaces but they are not Byzantine-resilient, i.e., they do not provide a correct service if some replicas are attacked and start to misbehave. This paper presents an efficient implementation of LBTS, a linearizable Byzantine fault-tolerant tuple space. LBTS uses a novel Byzantine quorum systems replication technique in which most operations are implemented by quorum protocols while stronger operations are implemented by more expensive protocols based on consensus. LBTS is linearizable and wait-free, showing interesting performance gains when compared to a similar construction based on state machine replication.

1. Introduction

The *generative coordination model*, originally introduced in the LINDA programming language [11], uses a shared memory object called a *tuple space* to support coordination between distributed processes. Tuple spaces can support communication that is decoupled both in time – processes do not have to be active at the same time – and space – processes do not need to know each others' addresses [5]. The tuple space can be considered to be a kind of storage that stores *tuples*, i.e., finite sequences of values. The operations supported are essentially three: inserting a tuple in the space, reading a tuple from the space and removing a tuple from the space. The programming model supported by tuple spaces is regarded as simple, expressive and elegant, being supported by middleware platforms like Sun's JAVASPACES and IBM's TSPACES.

There has been some research about fault-tolerant tuple spaces (e.g., [24, 2]). The objective of those works is essentially to guarantee the availability of the service provided by the tuple space, even if some of the servers that implement it crash. This paper goes one step further by describing a tuple space that tolerates Byzantine faults. More specifically, this work is part of a recent research effort in *intrusion-tolerant systems*, i.e., on systems that tolerate malicious faults, like attacks and intrusions [23]. These faults can be modeled as arbitrary faults, also called Byzantine faults in the literature.

The proposed tuple space is dubbed LBTS since it is a *Linearizable Byzantine Tuple Space*. LBTS is implemented by a set of distributed servers and behaves according to its specification if up to a number of these servers fail, either accidentally (e.g., crashing) or maliciously (e.g., by being attacked and starting to misbehave). Moreover, LBTS also tolerates accidental and malicious faults in an unbounded number of clients accessing it. LBTS has two important properties. First, it is *linearizable*, i.e., it provides a strong concurrency semantics in which operations invoked concurrently appear to take effect instantaneously sometime between their invocation and the return of their result [13]. Second, it is *wait-free*, i.e., every correct client process that invokes an operation in LBTS eventually receives a response, independently of the failure of other client processes or the contention in the system [12].

Another distinguished feature of LBTS is its novel use of the *Byzantine quorum systems* replication technique. Most operations on the tuple space are implemented by pure asynchronous Byzantine quorum protocols [15]. However, a tuple space is a shared memory object with consensus number higher than one [22], according to Herlihy's wait-free hierarchy [12], so it cannot be implemented using only asynchronous quorum protocols. In this paper we identify the tuple space operations that require stronger protocols, and show how to implement them using a *Byzantine* PAXOS consensus protocol [6, 17]. The philosophy behind our design is that simple operations are implemented by "cheap" quorum-based protocols, while stronger operations are implemented by more expensive protocols based on consensus. Although there are other recent works that use quorum-based protocols to implement objects stronger than registers [1] and to optimize state machine replication [7], LBTS is the first to mix these two approaches supporting wait freedom and being efficient even in the presence of contention.

Although this is the first linearizable Byzantine tuple space that we are aware of, there are several domains in which it might be interesting to use this service. For example, application domains with frequent disconnections and mobility like *ad hoc networks* [19] and *mobile agents* [5] can benefit from the time and space decoupling provided by LBTS. Another domain is *grid computing,* where a large number of computers are used to run complex computations. These applications are decoupled in space and time since the computers that run the application can enter and leave the grid dynamically [10].

The main contributions of the paper are the following: *(i.)* it presents the first linearizable tuple space that is Byzantine fault-tolerant; the tuple space requires $n \geq 4f+1$ servers, from which f can be faulty, and tolerates any number of faulty clients; *(ii.)* it introduces a new design philosophy to implement shared memory objects with consensus number higher than 1 [12], by using asynchronous quorum protocols for the weaker operations and consensus protocols (which require synchrony assumptions) for stronger operations; to implement this philosophy several new techniques are developed; and *(iii.)* it compares the proposed approach with Byzantine state machine replication [21, 6] and shows that LBTS presents several benefits: some operations are much cheaper and it supports the concurrent execution of operations, instead of executing them in total order.

2. Preliminaries

2.1. Tuple Spaces

The *generative coordination* model, originally introduced in the LINDA programming language [11], uses a shared memory object called a *tuple space* to support the coordination between processes. This object essentially allows the storage and retrieval of generic data structures called *tuples*.

Each tuple is a sequence of fields. A tuple t in which all fields have a defined value is called an *entry*. A tuple with one or more undefined fields is called a *template* (usually denoted by a bar, e.g., $\bar{t}$). An entry t and a template $\bar{t}$ *match* — $m(t,\bar{t})$ — if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. Templates are used to allow content-addressable access to tuples in the tuple space (e.g., template $\langle 1,2,* \rangle$ matches any tuple with three fields in which 1 and 2 are the values of the first and second fields, respectively).

A tuple space provides three basic operations [11]: $out(t)$ that outputs/inserts the entry t in the tuple space; $inp(\bar{t})$ that reads and removes some tuple that matches $\bar{t}$ from the tuple space; $rdp(\bar{t})$ that reads a tuple that matches $\bar{t}$ without removing it from the space. The $inp()$ and $rdp()$ operations are *non-blocking*, i.e., if there is no tuple in the space that matches the template, an error code is returned. Most tuple spaces also provide blocking versions of these operations, *in* and *rd*. These operations work in the same way of their non-blocking versions but stay blocked until there is some matching tuple available on the space.

These few operations together with the content-addressable capabilities of generative coordination provide a simple and powerfull programming model for distributed applications. The drawback of this model is that it depends of an infrastructure object (the tuple space), which is usually implemented as a centralized server, being a single point of failure, the main problem addressed in this paper.

2.2. System Model

The system is composed by an infinite set of *client processes* $\Pi = \{p_1, p_2, p_3, ...\}$ which interact with a set of n *servers* $U = \{s_1, s_2, ..., s_n\}$ that simulates a tuple space with certain dependability properties. We consider that each client process and each server has an unique id.

All communication between client processes and servers is made over *reliable authenticated point-to-point channels*. All servers are equipped with a local clock used to compute message timeouts. These clocks are not synchronized so their values can drift.

In terms of failures, we assume that an arbitrary number of client processes and a bound of up to $f \leq \lfloor \frac{n-1}{4} \rfloor$ servers can be subject to *Byzantine failures*, i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Clients or servers that do not follow their algorithm in some way are said to be *faulty*. A client/server that is not faulty is said to be *correct*. We assume *fault independence*, i.e., that the probability of each server failing is independent of another server being faulty. This assumption can be substantiated in practice using several kinds of diversity [20].

We assume an *eventually synchronous system model* [8]: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time), so that every message sent by a correct server to another correct server at instant $u >$ GST is received before $u + \Delta$. Δ and GST are unknown. The intuition behind this model is that the system can work asynchronously (with no bounds on delays) most of the time but there are stable periods in which the communication delay is bounded (assuming local computations take negligible time)[1]. This assumption is needed to guarantee the termination of the Byzantine PAXOS [6, 17]. An execution of a distributed algorithm is said to be *nice* if the bound Δ always holds and there are no server failures.

Additionally, we use a *digital signature scheme* that includes a signing function and a verification function that use

[1] In practice this stable period has to be long enough for the algorithm to terminate, but does not need to be forever.

pairs of public and private keys. A message is signed using the signing function and a private key, and this signature is verified with the verification function and the corresponding public key. We assume that each correct server has a private key known only by itself, and that its public key is known by all client processes and servers. We represent a message signed by a server s with a subscript σ_s, e.g., m_{σ_s}.

2.3. Byzantine Quorum Systems

Byzantine quorum systems [15] are a technique for implementing dependable shared memory objects in message passing distributed systems that can suffer Byzantine failures. Given a universe of data servers, a quorum system is a set of server sets, called *quorums*, that have a non-empty intersection. The intuition is that if a shared variable is stored replicated in all servers, any read or write operation has to be done only in a quorum of servers, not in all servers. Formally, a Byzantine quorum system is a set of server quorums $\mathscr{Q} \subseteq 2^U$ in which each pair of quorums intersect in sufficiently many servers (*consistency*) and there is always a quorum in which all servers are correct (*availability*).

The servers can be used to simulate one or more shared memory objects. In this paper the servers simulate a single object – a tuple space. The servers form a *f-masking quorum system*, which tolerates at most f faulty servers [15]. This type of Byzantine quorum systems requires that the majority of the servers in the intersection between any two quorums are correct, thus $\forall Q_1, Q_2 \in \mathscr{Q}, |Q_1 \cap Q_2| \geq 2f+1$. Given this requirement, each quorum of the system must have $q = \lceil \frac{n+2f+1}{2} \rceil$ servers and the quorum system can be defined as: $\mathscr{Q} = \{Q \subseteq U : |Q| = q\}$. This implies that $|U| = n \geq 4f+1$ [15]. With these constraints, a quorum system with $4f+1$ servers will have quorums of size $3f+1$.

2.4. Byzantine Paxos

Since LBTS requires some modifications to the basic Byzantine Paxos total order protocol [6], this section briefly presents this protocol.

The protocol begins with a client sending a signed message m to all servers. One of the servers, called the leader, is responsible for ordering the messages sent by the clients. The leader then sends a PRE-PREPARE message to all servers giving a sequence number i to m. A server accepts a PRE-PREPARE message if the proposal of the leader is *good*: the signature of m verifies and no other PRE-PREPARE message was accepted for sequence number i. When a server accepts a PRE-PREPARE message, it executes two steps of message exchange with the other servers to commit m as the i-th message to be delivered.

When the leader is detected to be faulty, a leader election protocol is used to freeze the current round of the protocol, elect a new leader and start a new round. When a new leader is elected, it collects the protocol state (messages exchanged) from $\lceil \frac{n+f}{2} \rceil$ servers. This information is signed and allows the new leader to verify if some message was already committed with some sequence number. Then, the new leader continues to order messages. For a complete description of the Byzantine Paxos protocol and its many subtleties, we refer the reader to [6, 17].

3. Linearizable Byzantine Tuple Space

This section presents LBTS. Since we are interested only in wait-free operations, we concentrate our discussion only in tuple space non-blocking operations. The tuple space correctness condition and the protocols correctness proofs, as well as some optimizations and improvements are omitted due to lack of space. We refer the interested reader to the extended version of this paper [3].

3.1. Design Rationale and New Techniques

As already discussed in the introduction, the design philosophy of LBTS is to use quorum-based protocols for read (*rdp*) and write (*out*) operations, and an agreement primitive for the read-remove operation (*inp*). The implementation of this philosophy requires the development of some new techniques, described in this section.

To better understand these techniques let us recall how basic quorum-based protocols work. Traditionally, the objects implemented by quoruns are read-write registers (e.g., [14, 15, 16, 18]). The state of a register in each replica is represented by its current value and a timestamp (a kind of "version number"). The write protocol usually consists in *(i.)* reading the register current timestamp from a quorum, *(ii.)* incrementing it, and *(iii.)* writing the new value with the new timestamp in a quorum (deleting the old value). In the read protocol, the standard procedure is *(i.)* reading the pair timestamp-value from a quorum and *(ii.)* applying some read consolidation rule such as *"the current value of the register is the one associated with the greater timestamp that appears $f+1$ times"* to define what is the current value stored in the register. To ensure register linearizability (a.k.a. atomicity) two techniques are usually employed: *write-backs* – the read value is written again in the system to ensure that it will be the result of subsequent reads (e.g., [16, 14]) – or the *listener communication pattern* – the reader registers itself with the quorum system servers for receiving updates on the register values until it receives the same register state from a quorum, ensuring that this state will be observed in subsequent reads (e.g., [18]).

In trying to develop a tuple space object using these techniques two differences between this object and a register were observed: *(1.)* the state of the tuple space (the tuples it contains) can be arbitrarily large and *(2.)* the *inp* operation cannot be implemented by read and write protocols due to the requirement that the same tuple cannot be removed by two concurrent operations. Difference *(1.)* turns difficult the applicability of timestamps for defining what is the current state of the space while difference *(2.)* requires that concurrent *inp* operations are executed in total order by all servers. The challenge is how to develop quorum protocols for implementing an object that does not use timestamps for versioning and, at the same time, requires a total order protocol in one operation. To solve these problems, we developed three algorithmic techniques.

The first technique introduced in LBTS serves to avoid timestamps in a collection object (one that its state is composed by a set of items added to it): we partition the state of the tuple space in infinitely many simpler objects, the tuples, that have three states: not inserted, inserted, and removed. This means that when a process invokes a read operation, the space chooses the response from the set of matching tuples that are in the inserted state. So, it does not need the timestamp of the tuple space, because the read consolidation rule is applied to tuples and not to the space state.

The second technique is the application of the listener communication pattern in the *rdp* operation, to ensure that the usual quorum reasoning (e.g., a tuple can be read if it appears in $f+1$ servers) can be applied in the system even in parallel with executions of Byzantine PAXOS for *inp* operations. In the case of a tuple space, the *inp* operation is the single read-write operation: '*if there is some tuple that match $\bar{t}$ on the space, remove it*'. The listener pattern is used to "fit" the *rdp* between the occurrence of two *inp* operations. The listener pattern is not used to ensure linearizability as in previous works, but for capturing replicas' state between removals.

The third technique is the modification of the Byzantine PAXOS algorithm to allow the leader to propose the order *plus* a candidate result for an operation, allowing the system to reach an agreement even when there is no state agreement between the replicas. This is the case when the tuple space has to select a tuple to be removed that is not present in all servers. Notice that, without this modification, two agreements would have to be executed: one to decide what *inp* would be the first to remove a tuple, in case of concurrency (i.e., to order *inp* requests), and another to decide which tuple would be the result of the *inp*.

3.2. Protocols

Additional assumptions. We adopt several simplifications to improve the presentation of the protocols. First, we assume that all tuples are unique. In practice this might be implemented by appending to each tuple its writer id and a sequence number generated by the writer. Second, we assume that any message that was supposed to be signed by a server *s* and is not correctly signed is simply ignored. Third, all messages carry nonces in order to avoid replay attacks. Fourth, access control is implicitly enforced: the tuple space has some kind of access control mechanism (like an ACL) specifying what processes can insert tuples in it and each tuple has two sets of processes that can read and remove it. Fifth, the algorithms are described considering a single tuple space T, but their extension to support multiple tuple spaces is straightforward: a copy of each space is deployed in each server and all protocols are executed in the scope of one of the spaces (adding a field in each message indicating which tuple space is being accessed). Finally, we assume that the reactions of the servers to message receptions are atomic (e.g., lines 3-4 in Algorithm 1).

Protocol variables. Before we delve into the protocols, we have to introduce four variables stored in each server s: T_s, r_s, R_s and L_s. T_s is the local copy of the tuple space T in this server. The variable r_s gives the number of tuples previously removed from the tuple space replica in s. The set R_s contains the tuples already removed from T_s. We call R_s the removal set and we use it to ensure that a tuple is not removed more than once from T_s. Finally, the set L_s contains all clients registered to receive updates from this tuple space. This set is used in the *rdp* operation. The protocols use a function $send(to, msg)$ to send a message *msg* to the recipient *to*, and a function $receive(from, msg)$ to receive a message *msg* sent by *from*.

Tuple insertion. Algorithm 1 presents the *out* protocol.

Algorithm 1 *out* operation (client p and server s).

```
{CLIENT}
procedure $out(t)$
  1: $\forall s \in U$, $send(s, \langle \text{OUT}, t \rangle)$
  2: wait until $\exists Q \in \mathcal{Q} : \forall s \in Q$, $receive(s, \langle \text{ACK-OUT} \rangle)$
{SERVER}
upon $receive(p, \langle \text{OUT}, t \rangle)$
  3: if $t \notin R_s$ then $T_s \leftarrow T_s \cup \{t\}$
  4: $send(p, \langle \text{ACK-OUT} \rangle)$
```

When a process p wants to insert a tuple t in the tuple space, it sends t to all servers (line 1) and waits for acknowledgments from a quorum of servers (line 2). At the server side, if the tuple is not in the removal set (indicating that it has already been removed), it is inserted in the tuple space (line 3). An acknowledgment is returned (line 4).

With this simple algorithm a faulty client process can inserts a tuple in a subset of the servers. In that case, we say that it is an *incompletely inserted tuple*. The number of incomplete insertions made by a process can be bounded to one, as described in [3]. As can be seen in next sections, *rdp* (resp. *inp*) operations are able to read (resp. remove) such a tuple if it is inserted in $f+1$ servers.

Tuple reading. *rdp* is implemented by Algorithm 2. The protocol is more tricky than the previous one for two reasons. First, it employs the listener communication pattern to capture the replicas state between removals. Second, if a matching tuple is found, the process may have to write it back to the system to ensure that it will be read in subsequent reads, satisfying linearizability property.

When $rdp(\bar{t})$ is called, the client process p sends the template $\bar{t}$ to the servers (line 1). When a server s receives this

Algorithm 2 *rdp* operation (client p and server s).

```
{CLIENT}
procedure $rdp(\bar{t})$
 1: $\forall s \in U, send(s, \langle \text{RDP}, \bar{t}\rangle)$
 2: $\forall x \in \{1,2,...\}, \forall s \in U, Replies[x][s] \leftarrow \bot$
 3: repeat
 4:     wait until $receive(s, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s\rangle_{\sigma_s})$
 5:     $Replies[r_s][s] \leftarrow \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s\rangle_{\sigma_s}$
 6: until $\exists r \in \{1,2,...\}, \{s \in U : Replies[r][s] \neq \bot\} \in \mathcal{Q}$
 7: {From now on $r$ indicates the $r$ of the condition above}
 8: $\forall s \in U, send(s, \langle \text{RDP-COMPLETE}, \bar{t}\rangle)$
 9: if $\exists t, count\_tuple(t, r, Replies[r]) \geq q$ then
10:     return $t$
11: else if $\exists t, count\_tuple(t, r, Replies[r]) \geq f+1$ then
12:     $\forall s \in U, send(s, \langle \text{WRITEBACK}, t, Replies[r]\rangle)$
13:     wait until $\exists Q \in \mathcal{Q} : \forall s \in Q, receive(s, \langle \text{ACK-WB}\rangle)$
14:     return $t$
15: else
16:     return $\bot$
17: end if
{SERVER}
upon $receive(p, \langle \text{RDP}, \bar{t}\rangle)$
18: $L_s \leftarrow L_s \cup \{\langle p, \bar{t}\rangle\}$
19: $T_s^{\bar{t}} \leftarrow \{t \in T_s : m(t, \bar{t})\}$
20: $send(p, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s\rangle_{\sigma_s})$
upon $receive(p, \langle \text{RDP-COMPLETE}, \bar{t}\rangle)$
21: $L_s \leftarrow L_s \setminus \{\langle p, \bar{t}\rangle\}$
upon $receive(p, \langle \text{WRITEBACK}, t, proof\rangle)$
22: if $count\_tuple(t, proof) \geq f+1$ then
23:     if $t \notin R_s$ then $T_s \leftarrow T_s \cup \{t\}$
24:     $send(p, \langle \text{ACK-WB}\rangle)$
25: end if
upon removal of $t$ from $T_s$ or insertion of $t$ in $T_s$
26: for all $\langle p, \bar{t}\rangle \in L_s : m(t, \bar{t})$ do
27:     $T_s^{\bar{t}} \leftarrow \{t' \in T_s : m(t', \bar{t})\}$
28:     $send(p, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s\rangle_{\sigma_s})$
29: end for
Predicate: $count\_tuple(t, r, msgs) \triangleq$
        $|\{s \in U : msgs[s] = \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r\rangle_{\sigma_s} \wedge t \in T_s^{\bar{t}}\}|$
```

message, it registers p as a listener, and replies with all tuples in T_s that match $\bar{t}$ and the current number of tuples already removed r_s (lines 18-20). While p is registered as a listener, whenever a tuple is added or removed from the space the tuples that match $\bar{t}$ is sent to p [2] (lines 26-29).

Process p collects replies from the servers, putting them in the *Replies*$_s$ matrix, until it manages to have a set of replies from a quorum of servers reporting the state after the same number of tuple removals r (lines 2-6). After that, a RDP-COMPLETE message is sent to the servers (line 8).

The result of the operation depends on a single row r of the matrix *Replies*$_s$. This row represents a cut on the system state in which a quorum of servers processed exactly the same r removals, so, in this cut, quorum reasoning can be applied. This mechanism is fundamental to ensure that agreement algorithms and quorum-based protocols can be used together for different operations, one of the novel ideas of this paper. If there is some tuple t in $Replies_s[r]$ that was replied by all servers in a quorum, then t is the result of the operation (lines 9-10). This is possible because this quorum ensures that the tuple can be read in all subsequent reads, thus ensuring linearizability. On the contrary, if there is no tuple replied by an entire quorum, but there is still some tuple t returned by more than f servers[3] for the same value of r, then t is *write-back* in the servers (line 11-12). The purpose of this write-back operation is to ensure that if t has not been removed until r, then it will be readable by all subsequent $rdp(\bar{t})$ operations requested by any client, with $m(t, \bar{t})$ and until t is removed. Therefore, the write-back is necessary to handle incompletely inserted tuples.

Upon the reception of a write-back message $\langle \text{WRITEBACK}, t, proof\rangle$, server s verifies if the write-back is *justified*, i.e., if *proof* includes at least $f+1$ correctly signed REP-RDP messages from different servers with r and t (line 22). A write-back that is not justified is ignored by correct servers. After this verification, if t is not already in T_s and has not been removed, then s inserts t in its local tuple space (line 23). Finally, s sends a ACK-WB to the client (line 26), which waits for these replies from a quorum of servers and returns t (lines 13-14).

Tuple destructive reading. The previous protocols are implemented using only Byzantine quorum techniques. The protocol for *inp*, on the other hand, requires stronger abstractions. This is a direct consequence of the tuple space semantics that does not allow *inp* to remove the same tuple twice (once removed it is no longer available).

An approach to implement this semantics is to execute all *inp* operations in the same order in all servers. This can be made using a total order multicast protocol based on the Byzantine PAXOS algorithm (Section 2.4). A simple approach would be to use it as an unmodified building block, but this requires two executions of the protocol for each *inp* [4]. To avoid this overhead, the solution we propose is based on *modifying* this algorithm in three specific points:

1. When the leader s receives a request $inp(\bar{t})$ from client p (i.e., a message $\langle \text{INP}, p, \bar{t}\rangle$), it sends to the other servers a PRE-PREPARE message with not only the sequence number i but also $\langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t}\rangle_{\sigma_p}\rangle_{\sigma_s}$, where $t_{\bar{t}}$ is a tuple in T_s that matches $\bar{t}$. If there is no tuple that matches $\bar{t}$ in T_s, then $t_{\bar{t}} = \bot$.

2. A correct server s' accepts to remove the tuple $t_{\bar{t}}$ proposed by the leader in the PRE-PREPARE message if: *(i.)* the usual Byzantine PAXOS conditions for acceptance described in Section 2.4 are satisfied; *(ii.)* s' did not accept the removal of $t_{\bar{t}}$ previously; *(iii.)* $t_{\bar{t}}$ and $\bar{t}$

[2] In practice, only the update is sent to p.

[3] If a tuple is returned by f or less servers it can be a tuple that has not been inserted in the tuple space, created by a collusion of faulty servers.

match; and *(iv.)* $t_{\bar{t}}$ is not forged, i.e., either $t \in T_s$ or s' received $f+1$ signed messages from different servers ensuring that they have t in their local tuple spaces. This last condition ensures that a tuple t can be removed if and only if it can be read, i.e., only if at least $f+1$ servers report having it.

3. When a new leader l' is elected, each server sends its protocol state to l' (as in the original total order Byzantine PAXOS algorithm[4]) and a signed set with the tuples in its local tuple space that match $\bar{t}$. This information is used by l' to build a proof for a proposal with a tuple t (in case it gets that tuple from $f+1$ servers). If there is no tuple reported by $f+1$ servers, this set of tuples justifies a $\perp$ proposal. This condition can be seen as a write-back from the leader in order to ensure that the tuple will be available in sufficiently many replicas before its removal.

Giving these modifications on the total order protocol, an *inp* operation is executed by Algorithm 3.

Algorithm 3 *inp* operation (client p and server s).

```
{CLIENT}
procedure $inp(\bar{t})$
 1: $TO\text{-}multicast(U, \langle \text{INP}, p, \bar{t} \rangle)$
 2: wait until receive $\langle \text{REP-INP}, t_{\bar{t}} \rangle$ from $f+1$ servers in $U$
 3: return $t_{\bar{t}}$
{SERVER}
upon $paxos\_leader(s) \wedge P_s \neq \emptyset$
 4: for all $\langle \text{INP}, p, \bar{t} \rangle \in P_s$ do
 5:     $i \leftarrow i+1$
 6:     if $\exists t \in T_s : m(t, \bar{t}) \wedge \neg marked(t)$ then
 7:         $t_{\bar{t}} \leftarrow t$
 8:         $mark(i, t)$
 9:     else
10:         $t_{\bar{t}} \leftarrow \perp$
11:     end if
12:     $paxos\_propose(i, \langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle \rangle)$
13: end for
upon $paxos\_deliver(i, \langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle \rangle)$
14: $unmark(i)$
15: $P_s \leftarrow P_s \setminus \{\langle \text{INP}, p, \bar{t} \rangle\}$
16: if $t_{\bar{t}} \neq \perp$ then
17:     if $t_{\bar{t}} \in T_s$ then $T_s \leftarrow T_s \setminus \{t_{\bar{t}}\}$
18:     $R_s \leftarrow R_s \cup \{t_{\bar{t}}\}$
19:     $r_s \leftarrow r_s + 1$
20: end if
21: $send(p, \langle \text{REP-INP}, t_{\bar{t}} \rangle)$
```

For a client p, the $inp(\bar{t})$ algorithm works exactly as if the replicated tuple space was implemented using Byzantine state machine replication [6, 21]: p sends a request to all servers and waits until $f+1$ servers reply with the same response, which is the result of the operation (lines 1-3).

In the server side, the requests for executions of *inp* received are inserted in the pending set P_s. When this set is not empty, the code in lines 4-13 is executed by the leader (the predicate $paxos\_leader(s)$ is *true* iff s is the current leader). For each pending request in P_s, a sequence number is attributed (line 5). Then, the leader picks a tuple from the tuple space that matches $\bar{t}$ (lines 6-7) and marks it with its sequence number to prevent it from being removed (line 8). The procedure $mark(i,t)$ marks the tuple as the one proposed to be removed in the i-th removal, while the predicate $marked(t)$ says if t is marked for removal. If no unmarked tuple matches $\bar{t}$, $\perp$ is proposed for the Byzantine PAXOS agreement (using the aforementioned PRE-PREPARE message), i.e., is sent to the other servers (lines 10, 12). The code in lines 4-13 corresponds to the modification 1 above. Modifications 2 and 3 do not appear in the code since they are reasonably simple changes of Byzantine PAXOS.

When the servers reach agreement about the sequence number and the tuple to remove, the *paxos_deliver* predicate is set to *true* and the code in the bottom of the algorithm is executed (lines 14-23). Then, each server s unmarks any tuple that it marked for removal with the sequence number i (line 14) and removes the ordered request from P_s (line 15). After that, if the result of the operation is a valid tuple $t_{\bar{t}}$, the server verifies if it exists in the local tuple space T_s (line 17). If it does, it is removed from T_s (line 18). Finally, $t_{\bar{t}}$ is added to R_s, the removal counter r_s is incremented and the result is sent to the requesting client process (line 23).

It is worth noticing that Byzantine PAXOS usually does not employ public-key cryptography when the leader does not change. The signatures required by the protocol are made using *authenticators*, which are vectors of message authentication codes [6]. However, modification 3 requires that the signed set of tuples will be sent to a new leader when it is elected. Therefore, our *inp* protocol requires public-key cryptography, but only when the operation cannot be resolved in the first Byzantine PAXOS round.

4. Evaluation

This section presents an evaluation of the system using two distributed algorithms metrics: *message complexity* and *communication steps*. Message complexity measures the maximum amount of messages exchanged between processes, so it gives some insight about the communication system usage and the algorithm scalability. The communication steps is the number of sequential communications between processes, so it is the main factor for the time needed for a distributed algorithm execution to terminate.

In this evaluation, we compare LBTS with an implementation of a tuple space with the same semantics based on *state machine replication* [21], which we call SMR-TS. SMR is a generic solution for the implementation of fault-

[4] The objective is to ensure that a value decided by some correct server in some round will be the only possible decision in all subsequent rounds.

tolerant distributed services using replication. The idea is to make all replicas to start in the same state and deterministically execute the same operations in the same order in all replicas. The implementation considered for SMR-TS is based on the Byzantine PAXOS [6] with fast decision (two communication steps) in nice executions [17, 25]. The fast decision is also considered for the modified Byzantine PAXOS used in LBTS' *inp* protocol. The SMR-TS implements an optimistic version for read operations in which all servers return immediately the value read without executing the Byzantine PAXOS if no concurrency is perceived.

Operation	LBTS		SMR-TS	
	M.C.	C.S.	M.C.	C.S.
out	$\mathbf{O(n)}$	**2**	$\mathbf{O(n^2)}$	**4**
rdp	$\mathbf{O(n)}$	$\mathbf{2}/4$	$\mathbf{O(n)}/O(n^2)$	$\mathbf{2}/6$
inp	$\mathbf{O(n^2)}$	$\mathbf{4}/7$	$\mathbf{O(n^2)}$	**4**

Table 1. Costs in nice executions

Table 1 evaluates nice executions of the operations in terms of message complexity (M.C.) and communication steps (C.S.)[5]. The costs of LBTS' operations are presented in the second and third columns of the table. The fourth and fifth columns show the evaluation of SMR-TS. The LBTS protocol for *out* is cheaper than SMR-TS in both metrics. The protocol for *rdp* has the same costs in LBTS and SMR-TS in executions in which there is no matching tuple being written concurrently with *rdp*. The first values in the line of the table corresponding to *rdp* are about this optimistic case ($O(n)$ for message complexity, 2 for communication steps). When a read cannot be made optimistically, the operation requires 4 steps in LBTS and 6 in SMR-TS (optimistic phase plus the normal operation). Moreover LBTS' message complexity is linear, instead of $O(n^2)$ like SMR-TS. The protocol for *inp* uses a single Byzantine PAXOS execution in both approaches. However, in cases in which there are many tuples incompletely inserted (extreme contention or many faulty clients), LBTS might not decide in the first round (as discussed in [3]). In this case a new leader must be elected. We expect this situation to be rare.

The table allow us to conclude that an important advantage of LBTS when compared with SMR-TS is the fact that in SMR-TS all operations require protocols with message complexity $O(n^2)$, turning simple operations such as *rdp* and *out* as complex as *inp*. Another advantage of LBTS is that its quorum-based operations, *out* and *rdp*, always terminate in few communication steps while in SMR-TS these operation relies on Byzantine PAXOS, that we can have certainty that terminates in 4 steps only in nice executions [17].

5. Related Work

Two replication approaches can be used to build Byzantine fault-tolerant services: Byzantine quorum systems [15] and state machine replication [21, 6]. The former is a data-centric approach based on the idea of executing different operations in different intersecting sets of servers, while the latter is based on maintaining a consistent replicated state across all servers in the system. One advantage of quorum systems in comparison to the state machine approach is that they do not need that the operations are executed in the same order in the replicas, so they do not need to solve consensus. Quorum protocols usually scale much better due to the opportunity of concurrency in the execution of operations and the shifting of hard work from servers to client processes [1]. On the other hand, pure quorum protocols cannot be used to implement objects stronger than register (in asynchronous systems), on the contrary of state machine replication, which is more general [9].

To the best of our knowledge there is only one work on Byzantine quorums that has implemented objects more powerful than registers in a way that is similar to ours, the Q/U protocols [1]. That work aims to implement general services using quorum-based protocols in asynchronous Byzantine systems. Since this cannot be done ensuring wait-freedom, the approach sacrifices liveness: the operations are guaranteed to terminate only if there is no other operation executing concurrently. A tuple space build using Q/U has mainly two drawbacks, when compared with LBTS: *(i.)* it is not wait-free so, in a Byzantine environment, malicious clients could invoke operations continuously, causing a denial of service; and *(ii.)* it requires $5f+1$ servers, f more than LBTS, and it has an impact on the cost of the system due to the cost of diversity [20].

Recently, Cowling et al. proposed HQ-REPLICATION [7], an interesting replication scheme that uses quorum protocols when there are no contention in operations executions and consensus protocols to resolve contention situations. This protocol requires $n \geq 3f+1$ replicas and process reads and writes in 2 to 4 communication steps in contention-free executions. When contention is detected, the protocol uses Byzantine PAXOS to order of contending requests. This contention resolution protocol adds great latency to the protocols, reaching more than 10 communication steps even in nice executions. Comparing LBTS with a tuple space based on HQ-REPLICATION, in executions without contention, LBTS' *out* will be faster (2 steps instead of 4 of HQ), *rdp* will be equivalent (the protocols are similar) and *inp* will have the same latency in both, however, LBTS' protocol has $O(n^2)$ message complexity instead of $O(n)$ of HQ. In contending executions, LBTS is expected to outperform HQ in orders of magnitude since its protocols are little affected by these situations. On the other hand, HQ-REPLICATION requires f fewer replicas than LBTS.

There are several works that replicate tuple spaces for fault tolerance. Some of them are based in the state machine replication (e.g., [2]) while others use quorum systems (e.g., [24]). However, none of these proposals deals with Byzan-

[5] Recall from Section 2.2 that an execution is said to be *nice* if the maximum delay Δ always hold and there are no failures.

tine failures and intrusions, the main objective of LBTS.

The construction presented in this paper, LBTS, builds on a preliminary solution with several limitations, BTS [4]. LBTS goes much further in mainly three aspects: it is linearizable; it uses a confirmable protocol for operation *out*; and it implements the *inp* operation using only one Byzantine PAXOS execution, instead of two in BTS.

6. Final Remarks

In this paper we presented the design of LBTS, a Linearizable Byzantine Tuple Space. This construction provides reliability, availability and integrity for coordination between processes in open systems. The overall architecture is based on a set of servers from which less than a fourth may be faulty and on an unlimited number of client processes, from which arbitrarily many can also be faulty.

LBTS combines Byzantine quorum systems protocols with consensus-based protocols resulting in a design in which simple operations use simple quorum-based protocols while a more complicated operation, which requires servers's synchronization, uses more complex agreement-based protocols. An important contribution of this work is the assertion that *out* and *rdp* can be implemented using quorum-based protocols, while *inp* requires consensus. This design shows important performance benefits when compared with the same object implemented using state machine replication.

Acknowledgements. We thank Paulo Sousa, Piotr Zielinski and Rodrigo Rodrigues for their suggestions to improve the paper. This work was supported by LaSIGE, CNPq (Brazilian National Research Council) through process 550114/2005-0, the EU through project IST-4-027513-STP (CRUTIAL) and CAPES/GRICES (project TISD).

References

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

[2] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):287–302, Mar. 1995.

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

[4] A. N. Bessani, J. S. Fraga, and L. C. Lung. BTS: A Byzantine fault-tolerant tuple space. In *Proc. 21st ACM Symp. on Applied Computing - SAC 2006*, Apr. 2006.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, Feb. 2000.

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

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

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

[9] R. Ekwall and A. Schiper. Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, 11(5):703–711, 2005.

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

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

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

[13] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[14] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of 26th IEEE Int. Conf. on Distributed Computing Systems - ICDCS 2006*, 2006.

[15] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[16] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*, Oct. 1998.

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

[18] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the 16th Int. Symposium on Distributed Computing - DISC 2002*, Oct. 2002.

[19] A. Murphy, G. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. on Software Engineering and Methodology*, 15(3):279–328, July 2006.

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

[21] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

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

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

[24] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proc. of the 19th Symp. on Fault-Tolerant Computing - FTCS'89*, June 1989.

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