BTS: A Byzantine Fault-Tolerant Tuple Space*

Alysson Neves Bessani
DAS/UFSC
Florianópolis - SC, Brazil
neves@das.ufsc.br

Joni da Silva Fraga
DAS/UFSC
Florianópolis - SC, Brazil
fraga@das.ufsc.br

Lau Cheuk Lung
PPGIA/PUC-PR
Curitiba - PR, Brazil
lau@ppgia.pucpr.br

ABSTRACT

Generative coordination is one of the most prominent coordination models for implementing open systems due to its spatial and temporal decoupling. Recently, a coordination community effort has been trying to integrate security mechanisms to this model aiming to improve its robustness. In this context, this paper presents the BTS coordination model, which provides a Byzantine fault-tolerant tuple space. Byzantine faults are commonly used to represent both process crashes and intrusions. As far as we know, BTS is the first coordination model that supports this dependability level.

1. INTRODUCTION

Most of the modern distributed systems (ex. peer-to-peer networks, web services and grid computing) have **open systems** characteristics: an unknown number of unreliable and heterogeneous (ex. variable computation and communication capacity) participants. The coordination mechanisms used for processes interaction in most of current distributed systems, as the message passing communication paradigm, are not suited for the future open systems. Requirements of anonymity and possibility of temporary disconnections (unreliable communications) imply the need for decoupled interactions. Therefore, alternative coordination models are needed. Amongst these models, the generative coordination [11] stands out due to its flexibility and simplicity. In this model, processes interact through a shared memory space (tuple space) in which generic data structures (tuples) are inserted, read and removed through a reduced (and simple) operations set. The coordination is decoupled in time (the participants do not need to be active at the same time) and in space (they do not need to know each other) [5].

Independently of the coordination model used in open distributed systems, these interactions are subject to all sorts of failures and security attacks. These events can affect both the participants being coordinated and the coordination infrastructure, compromising the entire system. One way to improve the system reliability is to interpret accidental or malicious failures as **Byzantine** faults [15] so that fault-tolerance techniques can be used to make the coordination infrastructure resilient to both, accidental crashes and intrusions.

This paper proposes a Byzantine fault-tolerant tuple space coordination model, called **BTS (Byzantine Tuple Space)**. This proposal is based on a shared tuple space simulation over a distributed system (without shared memory) in which processes communicate by message passing. The replication technique used for implementing this simulation is Byzantine quorum systems [16, 18].

The development of this coordination infrastructure aims to provide **intrusion tolerance**[1] [10, 22] in open systems. Therefore, the environment considered in this work is very unfavorable: an arbitrary number of processes, subject to Byzantine failures and executing in a very weak environment in terms of time guarantees. The construction of a reliable tuple space in an environment with these characteristics complements recent efforts of the coordination community aiming to integrate security mechanisms into the generative coordination model [8, 19, 23, 3].

This paper is structured as follows. Section 2 presents an overview of the generative coordination model. Section 3 presents our system model, the BTS protocols and the Byzantine quorum system used to build it. Section 4 discusses the performance of BTS. Section 5 describes some related work. Finally, Section 6 ends the paper with some concluding remarks.

2. GENERATIVE COORDINATION

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

In this model, each tuple is a sequence of typed fields that may have a defined value. A tuple in which all fields have a defined value is called **entry**. A **template**, usually denoted by $\bar{t}$, is a tuple that has one or more undefined fields. The **type of a tuple** t, denoted by $type(t)$, is the sequence of types of each field of t. An entry t and a template $\bar{t}$ **match**, denoted by $m(t,\bar{t})$, if they have the same type and all defined field values of $\bar{t}$ are equal the corresponding field values of t.

There are three basic operations defined for data manipulation in a tuple space [11]: $out(t)$, which outputs the entry t in the tuple space; $in(\bar{t})$, which removes a tuple that matches $\bar{t}$ from the tuple space (destructive read); and $rd(\bar{t})$, which is similar to $in(\bar{t})$, but the tuple read is not removed from the space (non-destructive read). The *in* and *rd* operations are blocking, i.e., if there is no tuple in the space that matches the specified template, the invoking process

*This work is supported by CNPq (Brazilian National Research Council) through process 550114/2005-0.

[1]A system is called intrusion-tolerant if it can provide a continuous and safe service despite the occurrence of intrusions in a bounded number of its components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'06 April 23-27, 2006, Dijon, France
Copyright 2006 ACM 1-59593-108-2/06/0004 ...$5.00.

will block until a matching tuple becomes available. A common extension to this model (adopted in this paper) is the inclusion of non-blocking variants of these read operations, called *inp* and *rdp*, respectively. These operations work in the same way of their blocking versions but return even if there is no matching tuple for the specified template in the space (signaling failure). Notice that according to the definitions above, the tuple space works just like an associative memory: tuples are accessed through their contents and not through their address.

3. BTS PROTOCOLS

In this section we will define our system model as well as the Byzantine quorum system replication technique, and present the BTS protocols.

3.1 System Model

The adopted system model assumes a 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 fault-tolerant tuple space.

All communications between client processes and servers are held over **reliable and authenticated point-to-point channels** that deliver messages in FIFO (First-In First-Out) order[2]. Additionally, all processes are equipped with a local clock, used to compute messages timeouts. These clocks can diverge one from the others, i.e., they are not synchronized.

In terms of processes failures, we assume that an arbitrary number of clients and a bound of $f \leq \lfloor \frac{n-1}{3} \rfloor$ servers could be subject to **Byzantine failures** [15]: they could deviate arbitrarily from their specification and work in collusion to corrupt the system behavior. A process behaving like that is said to be **faulty**. A process that is not faulty is said to be **correct**. Also, we assume **fault independence**, obtained by the systematic use of hardware and software diversity (SO, VM, etc) [7].

As a tuple space cannot be implemented using only basic read/write shared memory objects (registers) [21, 13], we must assume some weak time assumptions to build this kind of object. In that way, we assume the **eventually synchronous system model** [9]: in all executions of the system, there is a bound Δ and a time GST (Global Stabilization Time), so that every message sent by a correct process to a correct process at time $t >$ GST is received at time $t + \Delta$. It is important to notice that neither Δ nor GST need to be known by the processes. An execution of a distributed algorithm is said to be **nice** if the bound Δ is always held and there are no process failures.

The intuition behind this model is that the system works asynchronously (respecting no time bounds) most of the time but eventually enters in a stable synchronous period where all tasks (message transmissions, for instance) are performed in a timely way[3]. Finally we assume also that all local computations require negligible time. Notice that this model resembles the Internet behavior.

3.2 Byzantine Quorum Systems

Quorum systems [12] are valuable alternative for implementing reliable shared memory objects in systems in which processes communicate by message passing. In this way, a quorum system for a universe of data servers is a collection of server sets, called **quorums**, that have a non-empty intersection. The principle behind their use in storage services is that if a shared variable is stored in all servers, read and write operations could be done only in a quorum of servers. The existence of intersections between the quorums enables the development of read and write protocols that maintain the integrity of the shared variable even if these operations are performed in different quorums of the system. **Byzantine quorum systems** [16] are a generalization of this technique suited for environments in which processes could fail in a Byzantine way.

Formally, a Byzantine quorum system is a set of server quorums $\mathcal{Q} \subseteq 2^U$ in which each pair of quorums from $\mathcal{Q}$ intersect in sufficiently many servers (consistency property) and there is always a quorum in which all servers are correct (availability property).

Servers simulate shared memory objects (in this paper, a tuple space) organized as a masking quorum system, which tolerates at most f faulty servers [16]. More specifically, BTS is built using an **asymmetric masking quorum system** [18], that distinguishes itself from other types of quorum systems by the use of different quorum sizes in different operations.

Due to the requirements of asymmetric quorum systems, we assume $n \geq 3f + 1$ and a system composed by two types of quorums ($\mathcal{Q} = \mathcal{Q}_r \cup \mathcal{Q}_w$): read ($Q_r \in \mathcal{Q}_r$) and write ($Q_w \in \mathcal{Q}_w$), each with $\lceil \frac{n+f+1}{2} \rceil$ and $\lceil \frac{n+f+1}{2} \rceil + f$ servers, respectively.

3.3 Protocols

This section presents the protocols for executing all tuple space operations in the BTS model. First, we will present protocols for the non-blocking variants of operations *rd* and *in* and then we will derive their blocking versions.

All algorithms defined in this section assume that each server s has a local copy of the tuple space, denoted by T_s, and a "to be removed" set, denoted by R_s. Since standard set operations are applied to manipulate these two tuple sets, we assume that identical tuples does not exist. It can be implemented, for example, by appending the id of the process that created the tuple plus a sequence number in an opaque field (will not be read) of it.

3.3.1 *out*

The invocation of the output operation *out* inserts a tuple t in space. This operation is implemented through the Algorithm 1.

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

$\{Client\}$	$\{Server\}$
procedure $out(t)$	**upon** $receive(p, \langle \text{OUT}, t \rangle)$
1: $\forall s \in Q_w, send(s, \langle \text{OUT}, t \rangle)$	2: **if** $t \notin R_s$ **then**
	3: $T_s \leftarrow T_s \cup \{t\}$
	4: **end if**
	5: $R_s \leftarrow R_s \setminus \{t\}$

Due to the generative model non-deterministic nature and the use of reliable communication channels, a client does not need to wait replies from servers when it inserts a tuple in the space. When a server receives a tuple t to be inserted, it only adds this tuple to the space if t was not already removed, and its removal from this server is pending (it belongs to R_s). This kind of control is necessary to ensure that the same tuple will not be removed twice from the tuple space. This type of write protocol, called **non-confirmable** [18], requires less message exchanges and can be used in systems in which the writer does not need to know the exact time when its write completes. We assume that an *out* operation completes when all correct servers in a write quorum receives the tuple.

A disadvantage about this implementation strategy is that it implies an **unordered** semantics for the *out* operation [4]. This semantics, the weakest for this operation, has been proved to be not sufficient to implement all programs [4].

[2]That can be easily implemented using SSL (Secure Socket Layer) technology.

[3]In practice, this stable period must be long enough for the distributed protocol terminate.

3.3.2 *rdp*

The non-destructive read operation $rdp(\bar{t})$ of a tuple t, which matches a template $\bar{t}$, is done in BTS through the protocol presented in Algorithm 2.

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

{*Client*}	{*Server*}
procedure $rdp(\bar{t})$	**upon** $receive(p, \langle \text{RD}, \bar{t} \rangle)$
1: $T[s_1, ..., s_n] \leftarrow (\perp, ..., \perp)$	11: $T_s^{\bar{t}} \leftarrow \{t \in T_s : m(t, \bar{t})\}$
2: $\forall s \in U$, $send(s, \langle \text{RD}, \bar{t} \rangle)$	12: $send(p, \langle \text{TS}, T_s^{\bar{t}} \rangle)$
3: **repeat**	
4: **wait** $receive(s, \langle \text{TS}, T_s^{\bar{t}} \rangle))$	
5: $T[s] \leftarrow T_s^{\bar{t}}$	
6: **until** $\{s \in U : T[s] \neq \perp\} \in \mathcal{Q}_r$	
7: **if** $\exists t : (\#T_{t\in} \geq f+1)$ **then**	
8: **return** t	
9: **end if**	
10: **return** $\perp$	

The protocol works basically with the client accessing a read quorum of servers[4] trying to get all tuples that match with the template $\bar{t}$ (lines 2-6). By the algorithm, each accessed server s must reply a match set $T_s^{\bar{t}}$ with all its tuples that match $\bar{t}$ (lines 11 and 12). The client collects the matching tuples from the read quorum and store them in an array T. Then it chooses a tuple t, common to the match set of at least $f+1$ servers (denoted by $\#T_{t\in} \geq f+1$ - lines 7 and 8). If there is no such tuple, the special symbol $\perp$ is returned, meaning a failure in the operation (line 10).

3.3.3 *inp*

Operation *inp* requires the most complex protocol for its implementation in BTS. This complexity results from the fact that one tuple cannot be removed from the space by two distinct invocations of *inp*. This requirement implies a natural mutual exclusion for processes trying to remove tuples from the same type.

Our proposal to implement this operation is based on two building blocks:

- **Mutual Exclusion**: The mutual exclusion problem concerns the management of a set of processes that want to have access to an indivisible resource that cannot be accessed by more than one process at a time. An algorithm that solves this problem must manage these processes ensuring that at most one of them will access the resource at a time (mutual exclusion property) and all processes will have access to the resource some time (no lockout property) [1]. This problem is usually defined in terms of a processes set trying to execute a privileged **critical section** of their code. Mutual exclusion algorithms are generally defined in terms of operations to enter a critical section referring to a resource ($enter(r)$) and leaving this section ($exit(r)$). In the BTS *inp* protocol, a mutual exclusion algorithm is used to ensure that an unique process will try to remove a tuple, i.e., the tuple destructive read in the critical section of the process.

- **Byzantine PAXOS Algorithm** The PAXOS algorithm [14] is one of the most interesting algorithms for solving the well know problem of **consensus** in distributed systems[5]. In this algorithm, we have three sets of processes (that may overlap) according to the different roles they play in the algorithm: **proposers**, who propose values, **acceptors**, who together try to establish agreement (chose) on a single proposed value, and **learners**, who must learn the chosen value [17]. The algorithm ensures that a single value proposed by a proposer will be chosen by all acceptors and learned by all learners. The BTS *inp* protocol uses a Byzantine variant of the PAXOS algorithm [6, 17] to force all correct servers to agree on a decision about removing or not a tuple for a client.

[4] A better implementation of a quorum access is to make the client send a message to a read quorum and also periodically to other servers until it receives responses from a complete read quorum.

[5] In this problem a set of processes try to reach agreement about a single value to be decided, based on their (possibly conflicting) proposed values [1].

Using these two building blocks and the read operation (defined in Section 3.3.2), the *inp* implementation can be defined in a very elegant way, as presented in Algorithm 3.

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

{*Client*}	{*Server*}
procedure $inp(\bar{t})$	**upon** $hasLock(p, type(\bar{t}))$
1: **repeat**	12: $d \leftarrow paxos(p, P, A, L, \perp)$
2: $enter(type(\bar{t}))$	13: **if** $d \neq \perp$ **then**
3: $t \leftarrow rdp(\bar{t})$	14: **if** $d \notin T_s$ **then**
4: **if** $t = \perp$ **then**	15: $R_s \leftarrow R_s \cup \{d\}$
5: $exit(type(\bar{t}))$	16: **end if**
6: **return** $\perp$	17: $T_s \leftarrow T_s \setminus \{d\}$
7: **end if**	18: **end if**
8: $d \leftarrow paxos(p, P, A, L, t)$	
9: $exit(type(\bar{t}))$	
10: **until** $d = t$	
11: **return** t	

This protocol uses the PAXOS algorithm through the invocation of the function *paxos* passing five parameters: the first process to propose a value, a set of proposers P, a set of acceptors A, a set of learners L and the proposed value. The returned value of this function is the value chosen by acceptors and learned by learners, which we call the decided value. The first four parameters of the PAXOS function invocation are the same in both client and server side invocations (lines 8 and 12): p (first proposer is the process in critical section), $P \triangleq \{p, s_1, ..., s_{f+1}\}$ (the proposers are p and $f+1$ servers, ensuring that there will always exist a correct proposer), $A \triangleq U$ (servers are the acceptors) and $L \triangleq \{p\} \cup U$ (learners are p and the servers). The proposed values by p and the servers are different, and will be explained below.

In the protocol of Algorithm 3, a process p which tries to remove a tuple t that matches a template $\bar{t}$ must first get exclusive access to the tuple type, i.e., reach its critical section (line 2). After that, p reads t (line 3) and, supposing that such tuple exists, runs the PAXOS algorithm to remove it (line 8). When PAXOS completes, p and all servers have reached a decision about t deletion or not, when p leaves its critical section (line 9). If the PAXOS decided value is t, the operation is over and t is the result of operation (line 11). Otherwise, p runs the described procedure again (loop of lines 1-10). If there is no tuple in space that matches $\bar{t}$, p exits its critical section and returns $\perp$, meaning a failure in the operation (lines 4-7).

When the server grants permission to a process p to enter its critical section (p locks a tuple type - $hasLocked(p, type(\bar{t})) = true$), it runs the PAXOS algorithm (line 12) assuming p as the initial proposer and proposing $\perp$. If the decided value is a tuple $t \neq \perp$ (line 13) then t is the tuple removed by process p. If t does not exists in the local tuple space copy, it is added to the "to be removed" set R_s (lines 14-16), ensuring that it will not be removed again. After that, t is removed from the tuple space local copy (lines 17).

For a process p to remove a tuple t matching $\bar{t}$ from the space, the decided value must be t. Opening the PAXOS black box we could see that the removal only happens when t proposed by p reaches

sufficiently many servers[6] before these servers suspects p and try to elect another proposer as the coordinator. The algorithm correctness is derived directly from this observation, the eventually synchronous system model assumed (Section 3.1), and the continuous execution of the algorithm until a process removes a tuple or realizes that there is no tuple to be removed.

As described in Algorithm 3, the *inp* protocol can still block since a process can execute line 8 and fail before executing line 9, failing to release the tuple type for other processes access. In order to solve this problem we have to remove the explicit resource release (line 9) from the client and include a local release in each server after it completes its part of the protocol (including a command $unlock(type(\bar{t}))$ after line 18). With this simple modification, the mutual exclusion algorithm ensures that a client that reaches its critical section will have an opportunity to propose a tuple for deletion to servers in a single execution of PAXOS, leaving its critical section after one value is decided.

Another modification that could be made in the algorithm, to optimize it, is the execution of the read operation (line 3) together with the mutual exclusion protocol (line 2). A client p could send the template $\bar{t}$ to be read to the servers with the mutual exclusion request and each server s could return their $T_s^{\bar{t}}$ together with the permission for p to enter its critical section (when this permission is granted). This optimization eliminates one access to the quorum system, decreasing the amount of messages exchanged and the time needed to execute *inp*.

Figure 1 shows a nice execution of the *inp* protocol. In this execution the algorithm is using a PAXOS-based mutual exclusion algorithm: each process sends a message to the servers asking for access to the critical section and the servers execute an agreement to decide the order in which each requesting process will be served. Byzantine PAXOS internal communication is explained in [17].

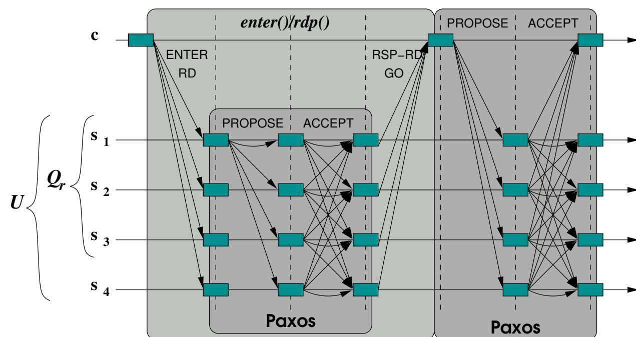

Figure 1: ***inp*** **protocol execution ($n = 4$ and $f = 1$).**

Figure 1 also shows that the *inp* protocol is non-confirmable (as *out* protocol from Section 3.3.1): a process p that removes a tuple t learns that this tuple will be removed only by itself and by no other process due to the PAXOS safety properties [17]. Moreover, p does not know when all correct servers will remove t from their local tuple spaces, since it happens only after they learn t (line 14-17).

3.3.4 Blocking Operations

The blocking operations *rd* and *in* can be implemented by a client p simply by repeating the execution of *rdp* and *inp* until p obtains the desired tuple [24]. Although this implementation strategy is not optimal, it is valid according to the inherent tuple space non-determinism [11].

[6]At least $f+1$ servers, ensuring that at least one correct server accepted the value.

4. PROTOCOLS ANALYSIS

Table 1 presents the operations protocols cost considering two metrics: message complexity (M.C.) and communication steps (C.S.). The message complexity measures the maximum amount of messages exchanged between processes. Consequently, it gives some insights about the communication system usage and the algorithm scalability. The communication steps accounts the number of sequential communications between processes, being the main factor in time needed for a distributed algorithm execution terminate.

Operation	BTS		State Machine	
	M.C.	C.S.	M.C.	C.S.
out	$O(n)$	1	$O(n^2)$	3
rdp	$O(n)$	2	$O(n)$	2
inp	$O(n^2)$	6	$O(n^2)$	4

Table 1: Tuple space protocols analysis

Values from Table 1 considers a nice execution of operations. The cost of BTS protocols are presented in the second and third columns of the table. The protocols for *out* and *rdp* operations are very simple and, consequently have a reduced cost in terms of both metrics when compared to the protocol for operation *inp*. This operation has its cost dominated by the mutual exclusion and the PAXOS algorithm, so it demands several extra communication steps and have a non-linear message complexity (see Figure 1).

For comparison purposes, the last two columns of Table 1 presents the protocols cost for the same operations (with the same semantics) when implemented using the well know **state machine replication** [20, 6] based on the same Byzantine PAXOS algorithm used in our *inp* protocol [17]. The state machine approach is the standard replication model for implementing general replicated services [17, 1]. To successfully implement this technique we must ensure **replica determinism**: starting from the same state, all correct replicas must execute the same operations sequence so that their states will be the same after executing each operation. To ensure this property, each request issued to the service must be executed in the same order in all replicas. It requires a consensus algorithm execution to establish a total order for replicated service requests. Since BTS does not satisfies linearizability [1], the same read protocol (Algorithm 2) can be used in a tuple space state machine implementation. The great disadvantage of this approach, as compared to our quorum-based BTS protocols, is the fact that the *out* operation becomes as complex as *inp*. This complexity is due to the fact that both operations updates the tuple space.

5. RELATED WORK

There are two replication models appropriate for building Byzantine fault-tolerant services: Byzantine quorum systems [16, 18] and state machine replication [20, 6]. The former is based on the concept 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. The advantage of quorum systems as compared to the state machine approach is that they do not need to solve consensus in every replicated service invocation. Consequently, there are lightweight protocols more appropriate to asynchronous systems. Our work use Byzantine quorum systems and develops specific protocols for each of the operations defined by the generative model. Only one of our protocols, for *inp* operation, requires a consensus protocol, therefore it needs more message exchanges and time assumptions (like the eventually synchronous system model assumed in this paper) to complete.

There are several works aiming to replicate tuple space for fault-tolerance. Some of them are based in the state machine approach (e.g. [2]) while others use quorum systems (e.g. [24]), however, none of these proposals regards the occurrence of Byzantine failures, the main objective of BTS.

Recently, several papers have proposed the integration of security mechanisms (like access control) in the generative model. These works are justified since this model has been increasingly used in hostile environments like the Internet. Amongst the proposals already published, some try to enforce predefined security policies according to the interacting processes expected behavior [19], others are concentrated in maintaining the confidentiality and integrity of tuples stored in space [8, 23, 3]. However, none of these works consider the availability of the tuple space, the fault-tolerance mechanisms main objective, neither apply any robust approach concerning intrusion tolerance. BTS copes with this problem through the use of Byzantine-resilient protocols for a replicated tuple space, surviving through faults, attacks and intrusions, as long as less than one third of the space replicas are comprised.

6. CONCLUDING REMARKS

In this paper we proposed BTS, a generative (tuple space) coordination model that tolerates Byzantine failures using a Byzantine quorum system and the Byzantine PAXOS consensus algorithm. As far as we know, our proposal is the first one to offer this dependability level and therefore, this is our main contribution. Moreover, the protocols presented built a shared memory object type strictly stronger (in terms of wait-free hierarchy [13]) than a register [21], usually implemented using quorum systems. This construction is possible through the use of a new approach that integrates asymmetric Byzantine quorum systems and consensus algorithms.

The approach presented in this paper complements other works that integrate security mechanisms to the generative model [8, 19, 23, 3] aiming its use in hostile environments (subject to failures and security attacks) like the Internet. Despite the fact that our protocols does not avoid poisonous or incomplete writes by faulty clients, it ensures that tuples written by correct ones will be available in the system as long as less than a third of servers are faulty.

Finally, we remark that despite the fact that the BTS protocols are defined considering only a single tuple space, they can be used to support multiple tuple spaces. To implement this, we have to put a copy of each space in each server of the system and execute the protocols in the scope of a specified space.

7. REFERENCES

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 2004.

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

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

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

[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 Transactions Computer Systems*, 20(4):398–461, 2002.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, 2003.

[8] R. De Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

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

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

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

[12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

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

[14] L. Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, 1998.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, 1982.

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

[17] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Dependable Systems and Networks, DSN 05*, 2005.

[18] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Dependable Systems and Networks, DSN 01*, 2001.

[19] N. H. Minsky, Y. M. Minsky, and V. Ungureanu. Making tuple-spaces safe for heterogeneous distributed systems. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 218–226, 2000.

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

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

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

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

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

APPENDIX

Not included in the SAC'06 published version

A. PAXOS-BASED MUTUAL EXCLUSION

It is well know that using the PAXOS algorithm one can build a straightforward implementation of a total order multicast communication primitive [1, 17, 6], a fundamental building block for implementing state machine replication [20] (see Section 4). Using this replication model we can implement a fault-tolerant mutual exclusion service in the same servers used for tuple space provision. Algorithm 4 presents the protocol to access this service.

Algorithm 4 Mutual exclusion (process p and server s).

$\{Client\}$
procedure $enter(r)$
1: $TO\text{-}multicast(U, \langle \text{ENTER}, p, r\rangle)$
2: **wait** $receive(Q_r, \langle \text{GO}, r\rangle)$
procedure $exit(r)$
3: $\forall s \in U, send(s, \langle \text{EXIT}, p, r\rangle)$
$\{Server\}$
var: $\forall r \in R, q_r \leftarrow \langle\rangle$
predicate $hasLock(p,r) \triangleq (head(q_r) = p)$
upon $TO\text{-}receive(\langle \text{ENTER}, p, r\rangle)$
4: $q_r \leftarrow append(q_r, p)$
5: **if** $head(q_r) = p$ **then**
6: $send(p, \langle GO, r\rangle)$
7: **end if**
upon $receive(p, \langle \text{EXIT}, p, r\rangle)$
8: **if** $head(q_r) = p$ **then**
9: $unlock(r)$
10: **end if**
procedure $unlock(r)$
11: $q_r \leftarrow tail(q_r)$
12: **if** $\neg empty(q_r)$ **then**
13: $send(head(q_r), \langle GO, r\rangle)$
14: **end if**

In the protocol from Algorithm 4 each server has a processes queue[7] for each resource r that it controls, denoted by q_r. The process holding the resource is the first in the queue. A client contends for mutual exclusion by sending a message to all servers (using the total order multicast primitive - line 1) and waits until it receives messages from a read quorum of servers (line 2). All correct servers, upon reception of the client request, put the client in the corresponding processes queue (line 4), and wait for resource release. The resource access is granted to processes as the resources are released by other processes (lines 5-7 and 10-12). Resource release is made through the *exit* procedure; a process sends messages to all servers informing that it exited its critical section (line 3 and 8-10).

B. BTS CORRECTNESS

In this section we will argue (informally) about BTS protocols correctness. The protocols safety is ensured by the fact that a tuple t cannot (always) be read in the system if *(i.)* its output in tuple space is not complete or if *(ii.)* its removal is complete. *(i.)* is correct because for t to be read, it must be replied by $f+1$ servers (line 7 of Algorithm 2) from a read quorum. Since a completed *out* outputs t in a read quorum of correct servers and $\forall Q_1, Q_2 \in \mathcal{Q}_r, |Q_1 \cap Q_2| \geq f+1$ (according to the consistency property [18]), it is clear that t will always be read. The correctness of *(ii.)* is stated by the fact that when a removal completes (line 14 of Algorithm 3), all correct servers remove t from their tuple space local copy or add t to the "to be removed" set. Consequently, there will be no $f+1$ servers that will reply t in a *rdp* operation, therefore it cannot be read.

The proposed protocols do not satisfy linearizability, because they allow a read that is concurrent with an update (*out* or *inp*), to return either the tuple being inserted/removed (assuming that this tuple matches the specified template) or other tuple that was already in the space before the update begin.

In terms of BTS liveness, all operations implemented by the protocols are **wait-free**, i.e., they always complete, independently of failures in other processes invoking operations in the shared tuple space [13]. The operation *out* always completes for a correct process since there is no **wait** clause in it. Operation *rdp* is also wait-free since the availability property of the quorum system being used ensures that there is always a correct read quorum available, which will reply the RD request sent by the client. The *inp* termination is ensured by the termination of each of its building blocks: the mutual exclusion must satisfy no lockout, consequently $enter$ terminates for every invoking process, *rdp* terminates (as showed above), and the PAXOS Algorithm terminate since we assume a eventually synchronous system model [14, 17].

[7] This queue is managed using standard list operators like $append$, $empty$, $tail$ and $head$.