On the Practicality to Implement Byzantine Fault Tolerant Services Based on Tuple Space

Aldelir Fernando Luiz*†, Lau Cheuk Lung‡, Luciana de Oliveira Rech‡
*College of Blumenau, Federal Institute Catarinense - Brazil
†Department of Automation and Systems, Federal University of Santa Catarina - Brazil
‡Department of Informatics and Statistics, Federal University of Santa Catarina - Brazil
aldelir.luiz@posgrad.ufsc.br, lau.lung@ufsc.br, luciana.rech@ufsc.br

***Abstract*—A major challenge of computer systems is making these more robust, reliable and secure. In recent years, it has been found that the state machine replication is one of the most common techniques for designing systems that need high reliability. This paper introduces a new architecture for replication of Byzantine fault tolerant service, based on the tuple space model. The proposed model requires only $2f + 1$ replicas for one service, and is generic enough to accommodate a different set of services at the same execution infrastructure. Use of tuple space supports communication and coordination, as well as a storage medium for sharing of data between service replicas. In addition, the paper also presents an experimental evaluation of some protocols for replication of Byzantine fault tolerant systems for practical purposes.**

I. Introduction

An interesting feature on designing distributed systems is its inherent support for redundancy, one of the fundamental principles of fault tolerance. On the other hand, implementing fault-tolerant distributed applications using redundancy techniques has been investigated for more than twenty-five years [1], [2]. Redundancy is implemented through abstractions of State Machine Replication (SMR) [1], [2], this technique is very attractive because it allows implementing services, which are capable of meet reliability, integrity and availability requirements, which are essential regarding to the trustworthiness of the computing system [3]. The SMR approach was first defined as a mechanism to provide survivability against crash faults [2], being later extended to the class of Byzantine faults [4].

On the other hand, the evolution of network and communication technologies have stimulated the emergence of new models of distributed systems. As a result, the community has been consolidating efforts to propose alternatives to **messages-passing model**, in order to simplify the programming task of reliable services for these models [5]. Moreover, the greatest difficulty in implementing reliable distributed systems lies in the relevant aspects to the coordination of the entities (e.g. communication, synchronization, etc.). Thus, recent studies have shown that the use of **coordination services** [6], [7], [8] greatly simplify one of the hardest parts of implementing distributed applications.

Before the rise of PBFT [4], proposals for Byzantine fault tolerant algorithms were dependent on very strong assumptions of complex cryptographic mechanisms, making the efficiency of these solutions to be put to the test. However, it is noteworthy that in recent years many studies have produced practical solutions for SMR replication focusing on Byzantine fault tolerance [4], [9], [10], [11], in order to make the applications resistant to malicious attacks against the system (intrusions), and thereby making the services intrusion-tolerant [12]. Some of these researches have shown, also, the feasibility of using these solutions to increase real service's reliability [4], [9], [10], [11]. Due to theoretical restrictions imposed by the Byzantine agreement problem [1], the solutions usually proposed in the literature [2], [4] involve a high cost of hardware and software. To tolerate f faulty entities, the algorithms usually require at least $3f + 1$ entities. Given this fact, new approaches for Byzantine fault tolerance were proposed aiming at reducing the number of communication steps [10], as well as the number of system entities [9], [13]. For this purpose, these solutions assume more realistic synchronism assumptions and, in some cases, use safe components. Particularly, the solution presented in [9] is quite interesting because it allows reducing the cost of Byzantine fault-tolerant replication to $2f + 1$, in the case of the service/application replicas.

This paper presents the RePEATS (Replication over Policy-Enforced Augmented Tuple Space), the proposed architecture for Byzantine fault tolerant SMR based on an abstraction of tuples, more precisely to extent the approach known in the literature as PEATS (Policy-Enforced Augmented Tuple Space) [14]. Thus, following the principle defined in [9], using the tuple space allows us to separate the entities responsible for the agreement, implemented on the tuple space, from those responsible for execution of requests sent by clients, with the advantage of having modular and much simpler replication algorithms. Our model requires only $2f + 1$ replicas for one service, and is generic enough to accommodate a set of services running distinct applications sharing the same communication and coordination support, while the particulars of services do not interfere each other. The RePEATS is, to the best of our knowledge, the first study to propose the use of higher level abstractions, in this case a tuple space, for implementing replicated services.

II. Related Works

Recent works in BFT have produced solutions based on Byzantine fault tolerant SMR, aiming to provide more secure

and reliable services. Typically, the algorithmic solutions for BFT assume that at most $f \leq \lfloor \frac{n-1}{3} \rfloor$ members of a set $|S| = n$ may arbitrarily deviate from their specifications simultaneously during a window of vulnerability of the system (e.g. a period). The PBFT protocol [4] is one of the most successful works in the field of practical Byzantine fault tolerance systems, due to the set of optimizations implemented in it. Moreover, most of the solutions for Byzantine faults replication such as [9], [10] are PBFT based on.

Due to the cost of PBFT from theoretical restrictions of Byzantine agreement [1], as well as the computational cost of its algorithms in terms of message complexity, two later studies have proposed alternatives to improve the limitations of PBFT both in terms of theoretical restrictions [9] by reducing the number of replicas, and of message's complexity under favorable conditions [10]. The first of them [9] introduced an architecture for state machine replication, where authors encourage a separation of the tasks performed by the PBFT in two distinct layers. The stimulus to separate these tasks, which are respectively "agreement and execution", occurred based on the finding that $2f + 1$ replicas are enough for masking Byzantine faults [2], in spite of the $3f + 1$ entities required to reach the Byzantine agreement [1]. Based on these assumptions, there is a possibility of building Byzantine fault tolerant services by means of two sets of servers; one for executing the Byzantine agreement protocol and, therefore, requiring $3f + 1$ replicas; and another with $2f + 1$ replicas for implementing the replicated service.

Moreover, in order to reduce the message complexity and the number of communication steps it was proposed the protocol Zyzzya [10], which introduced the concept of "speculative" execution in the context of Byzantine faults. The notion of speculation is linked to the ability to execute a request in a replicated service, without the need for an explicit agreement between the replicas. The idea behind Zyzzyva is the design of an optimistic ordering protocol (because the PBFT is rather pessimistic), since in practice it is observed that in most cases the executions are favorable (free of failures).

III. Concepts on Tuple Space

Conceptually, a tuple space can be seen as a shared memory abstraction, having the purpose of facilitating the interaction activities between distributed processes [15]. The origin of the tuple space occurred in the context of the language LINDA, from the definition of a model called generative coordination model [15]. In this model the tuple space is used as coordination and communication support to the participants of the system processes, where an interface to access the space is provided, so as to enable storage and retrieval (insertion, reading and removal) of generic data structures in the form of tuples. One tuple $t = \langle f_1, f_2, ..., f_n \rangle$ consists of the composition of a sequence of fields, where each field f_i can have representations: a defined value, a formal (variable) "?", or still a special symbol "*". A formal field is used to extract contents of individual fields from a tuple, while the special symbol is used to represent a field with no value, or a defined type. A tuple t where all fields have defined values is called entry. However, one tuple whose composition admits some formal field "?" and/or a special field "*" is called template, and is represented by $\bar{t}$. The space only allows the storage of tuples and never of templates, as the latter are used as arguments for over space reading operations. Reading and/or removal of tuples is performed based on the combination of the templates with the respective tuple space. Thus, we say that a template $\bar{t}$ matches with an entry t if they have the same number of fields and all fields with defined values of $\bar{t}$ contain the same values of the corresponding fields in t.

The manipulations of the tuple space are carried out by the invocation of three operations [15]: $out(t)$ to insert a tuple t in the tuple space; $in(\bar{t})$ to remove the tuple t that matches with the template $\bar{t}$, from tuple space; and $rd(\bar{t})$ for reading the tuple t (that matches with $\bar{t}$) without removing it from space. Reading and removal operations are non-deterministic, and therefore, if there is a set of tuples that matches with the specified template, any of them can be chosen as response to the operation. Also it should be noted that the operations *in* and *rd* are blocking, and if there is no tuple t that matches with the template $\bar{t}$ in the space, the process will remains blocked until a tuple matching with the template is inserted so that the operation is completed. However, a typical extension of the model provides non-blocking variants of reading and removal operations, usually called inp and rdp. It is worth mentioning, that the classical model of tuple-space coordination does not provide mechanisms to deal with malicious processes accessing the tuple space. This problem was solved with the introduction of the PEATS (Policy-Enforced Augmented Tuple Space) [14], which consists of a tuple space where the interactions between processes are governed by fine-granularity access policies.

IV. The Architecture RePEATS

A. An Overview of the RePEATS

An aspect of fundamental importance for the understanding of our contribution stems from the fact that all existing solutions for BFT (including those presented in section II), are based on the communication by message passing model, adopted in traditional distributed systems. However, the ability to solve fundamental problems of fault tolerance in distributed systems is inherently linked to the specification of a proper system model.

In our proposal, the PEATS provides a decoupled communication support for the interactions between the clients and the replicas of a service, in addition to a stable and secure storage environment for the information shared between replicas. In the Figure 1, clients issue their requests in the form of tuples in the PEATS (step 1) and the service replicas being accessed read these tuples from the PEATS to obtain the requests to be executed (step 2). Then, the replicated services process the requests and send the results, also in the form of tuples, to the PEATS (step 3), so that clients can get the response (step 4).

Obviously, the same assumptions and requirements applied to the SMR (replica determinism [2]), apply for the system

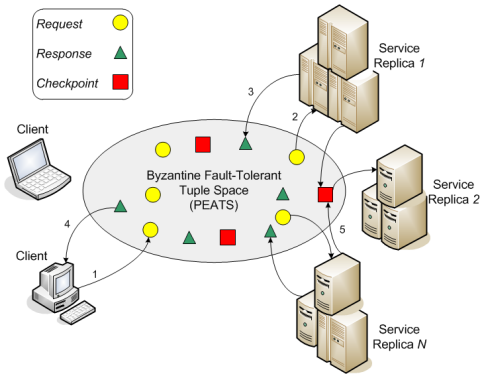


Fig. 1. Execution model of REPEATS

REPEATS, since it is a SMR realization in tuple space. In a system where the replicas implement a deterministic service, this property is implemented through the use of total order broadcast [16], ensuring that all submitted transactions are processed by all replicas (agreement) in the same order (total order). Thereafter, each replica performs the operation, updates its state (when necessary) and sends the result to the client. The client accepts the result of the operation as soon as it receives $f+1$ identical responses from different replicas, considering f the maximum number of servers which may suffer Byzantine faults.

B. System Model

We assume a system model that consists of an asynchronous computing environment provided by the tuple space, where there are two sets of processes: $C = \{c_1, c_2, ..., c_n\}$, which represents clients who interact with the service, and $S_e = \{s_{e1}, s_{e2}, ..., s_{en}\}$ representing the servers that implement the service replicas[1]. However, for the implementation of the tuple space we assume the existence of the set $S_{ts} = \{s_{ts1}, s_{ts2}, ..., s_{tsn}\}$ representing the servers that implement the reliable space.

In the case of faults, the model assumed is the Byzantine, where an arbitrary number of clients and up to $f \leq \lfloor \frac{n-1}{2} \rfloor$ servers may arbitrarily fail in their specifications, and thus stop, omit sending or delivery of messages, send incorrect answers, among others. Despite the faults, the independence of faults is assumed through the use of diversity in the various components that make up the architecture (hardware, operating systems, virtual machines, etc.) [17]. In the underlying communication system, e.g. the support system for the tuple space of no more than $f \leq \lfloor \frac{n-1}{3} \rfloor$ replicas may fail in maintaining corrected the coordination and communication environment. Thus, it is assumed that the tuple space is reliable, and therefore it is not susceptible to failures. This strong assumption is substantiated in practice by the use of a Byzantine fault tolerant tuple space, like the DEPSPACE [7], used in this work.

[1]Although the system supports multiple replicated services sharing the same infrastructure coordination, for simplicity of the algorithms, we assume only a single service replicated in this paper.

The system maintains the correction properties in the asynchronous interaction model (unknown time). However, to ensure the liveness (including the underlying communication), it is assumed an eventually synchronous system [18]. Thus, during the implementation of the computations and communications the time (which is unknown) does not increases indefinitely, and there are periods of stability that result in terminally synchronous computations and communications.

Finally, it is assumed that all communications occur through reliable point-to-point channels (even in the underlying level of communication with the tuple space abstraction) and authenticated. A pragmatic approach for these channels is the use of the TCP protocol with MACs [19] (Message Authentication Codes), using retransmission and reconnection mechanisms when appropriated. Moreover, all processes are equipped with local clocks that are not necessarily synchronized, whose sole premise is the progress.

C. Properties

Usually, the algorithms in a distributed systems are specified considering the notions of correctness (or safety) and termination (or liveness). For this work, some properties must be satisfied in order to allow the system to be successful and correct, they are:

- **Total order** (*safety*): the requests are performed with total order by all correct replicas of system;
- **Lock freedom** (*liveness*): in any system execution, if a correct process sends a request to run where there are pending requests, some request will be executed.

The first property concerns to correctness (safety), while the second is regarding to progress (liveness). Besides these properties, a service replicated using the REPEATS behaves equivalent to its implementation in a non-replicated system, satisfying the consistency model known as linearizability [20].

D. Access Control

A fundamental mechanism for the algorithms from REPEATS to tolerate malicious behavior of processes, is the access control provided by the fine-granularity policies supported by the PEATS [14]. This control takes place through the implementation of security policies associated with the tuple space. When an operation is invoked in the PEATS, the rules specified in these policies are checked based on the identification of process that invokes the operation, the operation which is being invoked (and its arguments) and the current state of the tuple space to deny or allow the operation.

E. Algorithm Ordering Requests

In this section we present the replication algorithm REPEATS. This algorithm, as well as the others that support this architecture, are executed on a PEATS, having as foundation the access control policies that prevent malicious processes of interfere with the correctness of our replication scheme. Briefly, the ordering protocol is supported by a persistent messages queue (built on the tuple space), analogously to the proposed group communication protocol for the system

SINFONIA [21]. This message queue, together with the appropriate access policy, ensures: (*i*) messages delivered in the same order to all processes (correct or faulty), (*ii*) persistence of messages while necessary.

As already discussed, the total order broadcast is a fundamental requirement for the state machine replication to be ensured that the properties concerning the determinism of the replicas. Since all communications take place via PEATS, the messages deposited in the space are available at the same time for all processes having access to the space. Thus, the total order of the requests is made from a message sequencing algorithm implemented on the tuple space. In this algorithm, the requests are inserted in the space with a sequence number only in tuples. These sequence numbers establish the order in which these requests must be executed by all replicas of a service. The message queues used in the REPEATS are essentially very similar to the characteristics of the group communication protocols [22], having its difference in how the order of messages delivery is determined. In the REPEATS this order is determined and controlled exclusively by the state of the application/replica, and not by means of views exchange as usually occurs in the group communication protocols. The persistence capacity of the messages provided by the messages queue of the REPEATS permits, above all, to implement a more efficient logging recovery mechanism. Thus, in the case of failure and subsequent recovery of a replica, it is not necessary to make an explicit agreement between the replicas (to recover and determine the execution sequence of messages that are not yet stable), since all system messages are persistent and are available in the queue.

The formalization of REPEATS ordering protocol is presented in the Algorithm 1. It is important to note that the correctness of the said protocol requires the following assumptions:

1) The clients requests are listed, where each one have a unique and growing identifier;
2) A client only sends a request after receiving response from a previous request[2];
3) For every request sent, the client associates a timer, and if it expires before the response has been obtained, the client re-sends the request;
4) Each replica has a results retention mechanism [23] that stores the response for last request (or last k requests) for each client, this response buffer is used to prevent reprocessing of requests that are sent by chance, besides allowing the status restoration of a replica, that eventually have remained behind the others.

The algorithm operation executed by the client (lines 4-8) is quite simple. When the client has to send a request to the service (command C), he tries to place a REQUEST tuple in the space with a sequence number one unit higher than its last sequence number (initially 0 - line 3). This insertion is done by invoking the operation cas, which checks if there is a tuple

[2]Note that this limitation can be relaxed to k requests if the servers have the ability to store the last k answers of each client.

Algorithm 1 Ordering Requests (client p_i and server s_i).

Shared variables:
1: $ts = \emptyset$ {tuple space}

{Client Side}

Local variables:
2: $seqno = 0$ {indicates the last position in the queue}
3: $reqid = 0$ {number of last sent request}
procedure $execute(C)$
4: $reqid \leftarrow reqid + 1$
5: **repeat**
6: $seqno \leftarrow find\_tail() + 1$
7: **until** $ts.cas(\langle REQUEST, seqno, *, *, * \rangle, \langle REQUEST, seqno, p_i, reqid, C \rangle)$
8: **return** $wait\_response(reqid)$

{Server Side} **Local variables:**
9: $seqno = 0$ {last position in the queue}
10: $data\_buffer = \emptyset$ {buffer to hold results}
11: $req\_ckpt = \perp$ {number of the greatest request in the last checkpoint}
main task
12: **loop**
13: **if** $req\_ckpt > seqno$ **then**
14: $ts.rd(\langle CKPT, ?ckptno, ?tail, ?state, ?buffer \rangle)$
15: $local\_state \leftarrow state$
16: $data\_buffer \leftarrow buffer$
17: $seqno \leftarrow tail$
18: **end if**
19: $seqno \leftarrow seqno + 1$
20: $ts.rd(\langle REQUEST, seqno, ?client_i, ?reqid, ?command \rangle)$
21: **if** $\nexists client_i \in client\_data$ **then**
22: $response \leftarrow local\_execute(command)$
23: **else**
24: $\langle reply, last\_reqid \rangle \leftarrow get\_data(data\_buffer, client_i)$
25: **case** $reqid$ **do**
26: ($last\_reqid + 1$)**:** $response \leftarrow local\_execute(command)$
27: ($last\_reqid$)**:** $response \leftarrow reply$
28: **default:** $continue$
29: **end case**
30: **end if**
31: $update\_data(data\_buffer, \langle client_i, reqid, response \rangle)$
32: $send\_response(client_i, response)$
33: **end loop**

REQUEST with the sequence number in the space in question, and if isn't, it issues the required tuple (line 7).

When it is not successful in this insertion, the client adds the sequence number and try again (lines 5-7). Upon successful insertion of the tuple, the client is blocked on line 8, where will wait for $f+1$ identical responses coming from distinct servers. The *find_tail* function (line 6) serves to prevent a client who has started the execution after having already a large number of requests in the space to have to iterate many times through the loop of lines 5-7 (increasingly invoking an operation on the tuple space).

This function implements a strategy that inspects the tuple space and tries to find a sequence number closer to that last request inserted in this space. The existence of this function does not affect the correctness of the algorithm, but greatly improves the performance of "delayed" clients. One possible strategy for the implementation of this function would be to check what is the newest checkpoint of the system and return

the number of last request in this checkpoint. Other possible strategies for implementing this function can be used by taking advantage of special functionalities which a particular implementation of the PEATS can implement.

Similarly to what happens in the clients' algorithm, the servers' algorithm (lines 9-33) is also quite simple. Initially the algorithm starts the recovery position of the requests' message queue to 0 (line 9). The first step executed in the algorithm is to check against the current state of the replica (line 13), when it is checked if the concerned replica has an out of date status in relation to other replicas, and being so, it updates its state from last valid checkpoint. This verification is important to ensure that the delayed replicas are not indefinitely blocked waiting for requests that are no longer contained in the ordered queue. Further, the replica processes the commands contained in the tuples REQUEST in ascending order firstly obtaining the tuple with sequence number higher than the last executed request (lines 19-20), and then checks if the request is the first one from the client, and being so, the replicas execute the command in the tuple/request through *local_execute* (line 22), insert the command response in the results table (line 31) and send the response to the client, indicated in the tuple REQUEST, by the function *send_response* (line 32).

For performance reasons, this function sends a message with the response directly to the client, without using the PEATS. In the case of the request not being the first one from the client, the information regarding the last processed request for this client are retrieved from the results table (line 24). Then, the replicas check if the request identifier is one unit higher than the last processed request or it's equal to the request in the table. In the first case, the replicas process normally the command from the request (line 26) and proceed to the end of the algorithm. In the latter, the answer stored in the table is returned to the client, because it's a retransmission. If the request is not in appropriate circumstances, it is discarded (line 28).

V. Implementation, Evaluation and Results

We choose the experimental way because, as we said, the RePEATS uses a shared memory abstraction to coordinate the algorithms' activities, which is different from other BFT solutions that coordinate their activities by message passing. It is noteworthy that the means to measure complexity of distributed algorithms based on message passing are different from shared memory based algorithms [24]. Thus, we did our implementations in Java. The reliable and authenticated communication channels (according to system model) were made by using socket channels from NIO API, using TCP with MACs (Message Authentication Codes).

The evaluation was performed on an environment composed by some Dell Optiplex 755 machines, every one with the same settings of hardware and software. Each machine had: 1 Intel®Core™2 Duo 2.33GHz processor, 2GB of RAM, one Ethernet Gigabit Intel 82566DM-2 interface, and SUSE Linux SLES 10 O.S. (Kernel 2.6.16.21-0.8-smp x86-64) equipped with a JVM-JIT IBM 1.6.0. The number of machines utilized for each experiment changed accordingly to the number of allowed faults (e.g. f parameter).

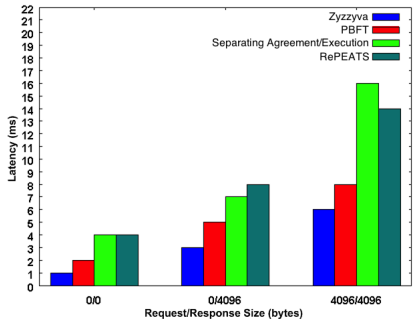
For evaluation, we adopted latency and throughput metrics because they allow a simplified verification of the system's efficiency [25]. The results were obtained from micro-benchmarks and macro-benchmarks in several load conditions, where latency were obtained from some round-trips and throughput were gotten from processing capacity of requests by time unit. We evaluated the systems from micro-benchmarks to the possibility of cost analysis (in time units) of ordering algorithms, without any application/service's influence. Moreover, it's noteworthy that for this micro-benchmark we evaluated protocols for systems using a stateless service with null operations, ranging requests and answers messages sizes in 0KB and 4KB.

A. Performance Evaluation in Normal and Faulty Scenarios

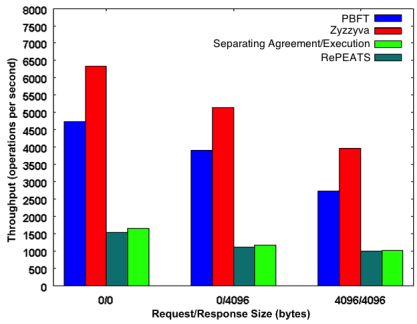
The first experiment was designed to evaluate executions of protocols in normal conditions without any faults. In both experiments from Fig. 2 we executed 10000 operations from 30 different concurrent clients disregarding 1% of the highly deviated values as observed from the executions. The results for latency measurement are reported in Fig. 2(a). They were obtained from executions with distinct load conditions, varying requests and responses size. The reported values include average time of request processing as observed by means of executions. The results (Fig. 2(a)) demonstrate that RePEATS's operations latency is the biggest of all, but still very similar to separating agreement's architecture from [9]. We must emphasize that these results were expected because RePEATS and [9] architecture systems share the same functional features. It is noteworthy that PBFT [4] and Yin's protocol [9] used in evaluation are own implementations.

The reason about the results reported for both RePEATS and [9] systems are too close to each other is because the DepSpace [7] is based on Byzantine Paxos replication algorithm, which is very similar to PBFT [4] that is employed as an agreement layer in [9]. Moreover, latency of RePEATS is worse than [9] due to additional cost from tuple space access.

An evaluation of throughput for BFT protocols is presented in Fig. 2(b). The result shows that RePEATS has smaller performance in relation to other evaluated protocols, however, being the results close to those reported to the 'separating agreement from execution' architecture. On the other hand, analyzing throughput in relation to the requests size and answers we can that RePEATS presents acceptable scalability if we consider that the decreasing number of operation/second is small when there is increase on the request and answers size. These results also show that the system has reasonable scalability when the number of request and answers is increased, because the increase in latency is less than twice. The particular case of 4096 bytes requests and answers is reasonable; since the adopted model requires one communication step further to send the ordained request to the execution servers, and that leads to a burden on the system when message's size is considerably large.

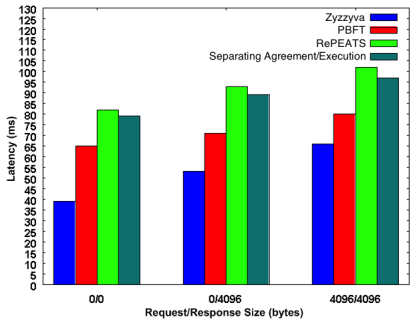


(a) Latency in normal operation.

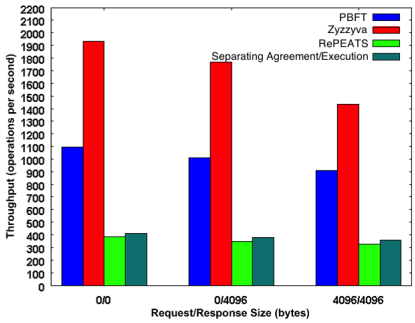


(b) Throughput in normal operation.

Fig. 2. Performance evaluation in normal case operations.



(a) Latency in faulty operations.



(b) Throughput in faulty operations.

Fig. 3. Performance evaluation in faulty operations.

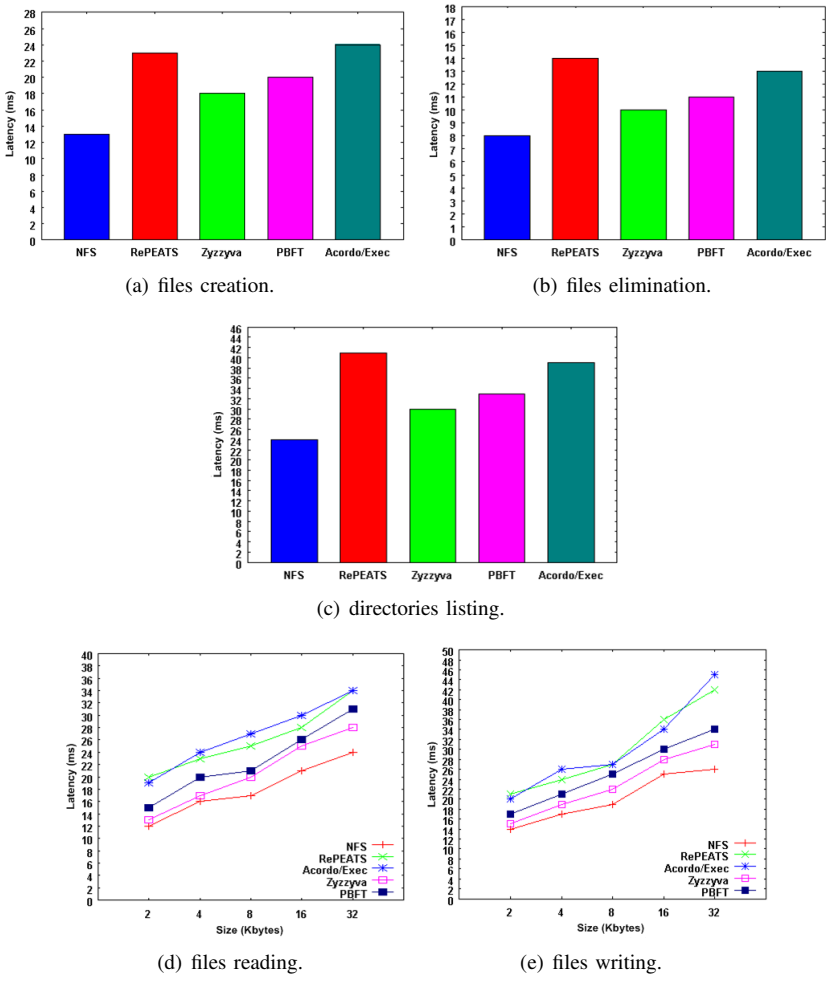
Likewise, in order to evaluate faulty behavior for BFT protocols we performed the same earlier experiments with injection of faults. Then, the faults were introduced in PBFT and Zyzzyva, also in separating agreement's architecture and in REPEATS. More precisely, the simulations injecting faults in replicas playing the leader role (because it describes the worst case!) to don't induce protocols to achieve the agreement on the first round of execution – e.g. causing a delay due to the change of the leader. For that, in case of the REPEATS and separating agreement's architecture we injected faults in the agreement layer since faults in execution layer hold almost no influence on system's performance due to masking them.

At first, we have evaluated the latency from execution of null operations on protocols. The results are similar to the earlier experiments such that both REPEATS and separating agreement's architecture presented a higher latency related to other BFT protocols. For values reported in Figure 3(a) we added the execution time from agreement protocol (PAXOS and BFT), and the time needed to send ordered messages to the execution servers. As shown in Fig. 3(a), in spite of our solution performs (e.g. it isn't the best), at this point REPEATS is slightly less than architecture of [9]. This happens due to additional access control of the tuple spaces that inhibits malicious behavior of Byzantine processes accessing the space. From results related to throughput (Fig. 3(b)) we can see that all protocols had their performance compromised on fault's situations since each protocol must execute other services and routines to restore the normal conditions, so assuring system liveness.

B. Case Study: An Evaluation on a BFT NFS Service

As quoted before, micro-benchmarks are useful to make a simple analysis on the protocol's impact over a replicated service. On the other hand, a macro-benchmark becomes essential when we want to evaluate how a real application performs. In this sense, we evaluated a replicated NFS Service implementation based on each evaluated BFT replicated state machine protocol. We did this evaluation based on the same patterns of the related works [4], [9], [10]. So, our implementation was inspired on Web NFS specifications [26], [27], in which both are extensions of NFS specifications allowing some facilities on remote objects access (files, directories and links). With the WebNFS an application got access to remote objects with no need for the operation system interaction - e.g. it's not necessary to mount the remote files system on each client's operating system's Virtual File System. The access to the objects happens through an API instantiated by client application.

The results gotten from macro-benchmark are obtained from NFS services implementation for each one of the evaluated protocols, along with a non-replicated NFS. The experiments cover the regular file operations execution to allow for latency verification for each operation. For NFS operations we analyzed: (1) files and directories creation (Figure 4(a)), (2) files and directories elimination (Figure 4(b)), (3) listing the contents of directories with one hundred recursive objects (Figure 4(c)), (4) writing on remote files varying size of data on 2k, 4k, 8k, 16k and 32k (Figure 4(d)), and (5) reading data from remote files with 2k, 4k, 8k, 16k and 32k of size



(a) files creation. (b) files elimination.

(c) directories listing.

(d) files reading. (e) files writing.

Fig. 4. Latency results for operations on a NFS service: Native NFS (*java.io* API + NFS Linux client), REPEATS, Separating Agreement, Zyzzyva e PBFT.

(Figure 4(e)). For this evaluation we ran each operation 10000 times, in which latency result took average time for operation execution, excluding 1% values with larger deviate.

From Figures 4(a) and 4(b), we observed what were already expected that is, both REPEATS and "separating agreement's" architecture presented the worst results. However, once again it is noteworthy that it happened because of the adopted replication model, which requires one more communication step. From another point of view (disregarding related works), we can verify that additional costs from the use of system REPEATS incurs between 30 to 60% compared with a non-replicated NFS service (e.g. non-tolerating Byzantine faults). This additional cost is basically due to latency for accessing the tuple space during the execution of agreement protocol (or ordering), more precisely due to the access control for accessing the space – a requirement to tolerate Byzantine faults in REPEATS. We believe that it is a moderate cost because there are some benefits considering the reliability and security covered by REPEATS. It is also noteworthy that the latency presents a slight increase when the size of request and response grows. This result is because of replies were sent directly to clients and thus, bypassing the tuple spaces (e.g. an optimization).

C. Analytical Evaluation on Byzantine Fault-Tolerant Protocols' Properties

This evaluation highlights three aspects that BFT replication protocols must stick to in order to make their algorithmic solutions more practical (of easy implementation), which are: replicas number's cost; a number of enough replicas to maintain the service, in spite of the minimum required in applications with Byzantine faults ($2f + 1$ [2]); number of services operating over one single agreement layer.

TABLE I
COMPARISON ON EVALUATED PROTOCOLS' PROPERTIES.

Properties and Characteristics	*Evaluated Protocols*			
	REPEATS	**Agreement/Execution**	**PBFT**	**Zyzzyva**
Number of replicas	$2f_e + 1 + \|S_{ts}\|$	$(3f_a + 1) + (2f_e + 1)$	$3f + 1$	$3f + 1$
Replicas of service	$2f_e + 1$	$2f_e + 1$	$3f + 1$	$3f + 1$
Services / Agreement Layer	N	$N*$	1	1

The data from Table I reflects a perspective on solutions adoption for evaluated BFTs, looking at total replication cost.

The first line in the table refers to the number of machines required in order to build a system tolerant to f Byzantine faults in each one of evaluated systems. The importance in replicas number reduction is crucial for building a service, once this implies also on system operational costs (storage, disk, etc). On this item, REPEATS stands out from all the others because it requires the fewest number of replicas on reliable service's implementation, even though S_{ts} replicas are necessary in tuple spaces that can be shared by different replicated services and other applications. A second evaluation is applied based on the number of required replicas to keep the application status, also taking into account the value established on [2]. This number indicates that both systems REPEATS and [9]'s architecture stand out for they share the same optimization, which consists on separate agreement layer from execution of services. This optimization doesn't apply to original PBFT protocols implementation neither Zyzzyva's, since on these ones same replicas execute both contract and application. Lastly, the last line in chart shows that both system REPEATS and [9]'s architecture permit the same according service use in order to implement several reliable applications, regardless of all works that does not provide that kind of optimization.

VI. FINAL REMARKS

In this work we presented an architecture's specification and implementation to support replication of those Byzantine fault tolerant systems based on tuple space. Moreover, an extensive evaluation from model and architecture proposed in relation to some protocols to fault Byzantine replication already known in literature was carried out, in order to verify the feasibility of the present proposal when planning applications tolerant to Byzantine fault. From the present work we could verify that in spite of presented performance, the extensive tuple spaces use is interesting from practical point of view, because its reduced operations set allows to specify and to implement, in fairly simple way, reliable and safe distributed system, in spite of the complexity verified to traditional message passing model.

Although the performance evaluation showed us that tuples space's use increases the replication cost, more precisely on latency terms, in relation to all evaluated works, this abstraction's use becomes attractive when we consider the increase on the number of faults supported for a service (e.g. f parameter).

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[2] 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.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Mar. 2004.

[4] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI '99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. USENIX Association, feb 1999, pp. 173–186.

[5] G. Cabri, L. Leonardi, and F. Zambonelli, "Mobile-agent coordination models for Internet applications," *Computer*, vol. 33, no. 2, pp. 82–89, Feb. 2000.

[6] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[7] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "Depspace: a byzantine fault-tolerant coordination service," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. ACM, 2008, pp. 163–176.

[8] S. F. Apache, "ZooKeeper Coordination Service," *http://hadoop.apache.org/zookeeper/*, 2009.

[9] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," in *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, oct 2003, pp. 253–267.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *SOSP '07: Proceedings of the 21st ACM SIGOPS/Symposium on Operating Systems Principles*. ACM, 2007, pp. 45–58.

[11] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *SOSP '09: Proceedings of the 22nd ACM SIGOPS/Symposium on Operating Systems Principles*. ACM, 2009, pp. 277–290.

[12] P. Veríssimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*, ser. LNCS, R. Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag, 2003, vol. 2677.

[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *SOSP '07: Proceedings of the 21st ACM SIGOPS/Symposium on Operating Systems Principles*. ACM, 2007, pp. 189–204.

[14] A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung, "Sharing memory between byzantine processes using policy-enforced tuple spaces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 419–432, March 2009.

[15] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programing Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.

[16] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[17] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 221–232, 1975.

[18] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of ACM*, vol. 35, no. 2, pp. 288–322, Apr. 1988.

[19] G. Tsudik, "Message authentication with one-way hash functions," *ACM Computer Communications Review*, vol. 22, no. 5, pp. 29–38, Oct. 1992.

[20] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programing Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[21] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP '07: Proceedings of the 21st ACM SIGOPS/Symposium on Operating Systems Principles*. ACM, 2007, pp. 159–174.

[22] K. P. Birman and T. A. Joseph, "Exploiting virtual synchrony in distributed systems," in *SOSP '87: Proceedings of the 11th ACM SIGOPS/Symposium on Operating Systems Principles*. ACM, 1987, pp. 123–138.

[23] K. P. Birman, T. A. Joseph, T. Raeuchle, and A. E. Abbadi, "Implementing fault-tolerant distributed objects," *IEEE Transactions on Software Engineering*, vol. 11, no. 6, pp. 502–508, 1985.

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

[25] R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[26] B. Callaghan, "WebNFS Server Specification (RFC 2055)," IETF Request For Comments, Oct. 1996.

[27] B. Callaghan, "WebNFS Client Specification (RFC 2054)," IETF Request For Comments, Oct. 1996.