Brief Announcement: Decoupled Quorum-based Byzantine-resilient Coordination in Open Distributed Systems*

Alysson Neves Bessani[1 2], Miguel Correia[2],
Joni da Silva Fraga[1], and Lau Cheuk Lung[3]

[1] DAS/PGEEL, Universidade Federal de Santa Catarina, Brazil
[2] LaSIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
[3] PPGIA, Pontificia Universidade Católica do Paraná, Brazil

1 Introduction

The *tuple space coordination model*, originally introduced in the LINDA programming language [2], uses a shared memory object called a *tuple space* to support coordination 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. 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.

In this paper we propose an efficient Byzantine fault-tolerant implementation of a tuple space called *LBTS (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 in a Byzantine way. Moreover, LBTS also tolerates accidental and malicious faults in an unbounded number of the clients that use its services and satisfies two important properties: linearizability and wait-freedom (with respect to client failures). In LBTS, most operations on the tuple space are implemented by pure Byzantine quorum protocols [3,4]. However, since a tuple space is a shared memory object with consensus number 2, it cannot be implemented using only quorum protocols. In this paper we identify the tuple space operations that require stronger protocols, and show how to implement them using a modified *Byzantine Paxos* consensus protocol [1]. 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.

2 LBTS Protocols

We assume an eventually synchronous system model composed by an infinite set of clients and $n \geq 4f + 1$ servers. An unbounded number of clients and at most

* Work supported by CNPq (project 506639/2004-5) and LaSIGE.

f servers can fail in a Byzantine way. The tuple space is implemented by the servers organized as a *f-masking Byzantine quorum system*, where each quorum contains $q = \lceil \frac{n+2f+1}{2} \rceil$ servers, ensuring an intersection of $2f+1$ servers between every two quorums of the system [3]. An important assumption of our protocols is that each tuple is unique. This assumption can be enforced in practice appending a nonce to each tuple.

In this paper we briefly describe the protocols that implement the three (non-blocking) operations of LBTS: *out*, for tuple insertion in the space; *rdp*, for tuple reading from the space; and *inp*, for tuple removal from the space. A fundamental property of the *inp* operation is that no two clients can remove the same tuple from the space.

Tuple Insertion (out). The tuple insertion protocol comprises a single access to a quorum giving the tuple being inserted. Each server that receives the tuple stores it in its local copy of the space if the tuple is not already stored and was not removed before. Notice that we are implementing a multi-writer storage but are not using timestamps. The protocol requires only two communication steps and has linear message complexity.

Tuple Reading (rdp). This operation is implemented by a protocol that executes in two phases. At first, the reading client accesses the servers requesting the tuples that match a given template. A server sends a response to the client together with the number of tuple removals it previously executed. The client is registered in a listener set and receives an update message every time a tuple that matches the given template is inserted in the server or some removal is executed. The client keeps collecting information from the servers until it receives matching tuples from a quorum of servers that removed the same number of tuples. If there is a tuple t that appears in at least $f + 1$ of these responses, t is the read tuple. If this tuple appears in less than q servers, the client has to write it back to the system to ensure that it will be read in subsequent reads and satisfy linearizability. Notice that the listener communication pattern is used for a different purpose than in [4]: the reader wants to "take a photo" of the system between removals in order to define the result of the read operation. This protocol requires 2 and 4 (when write-back is needed) communication steps and has linear message complexity. Its correctness relies on the fact that all removals (*inp*) are executed in all correct servers in the same total order.

Tuple Destructive Reading (inp). The approach to implement the semantics of this operation (no two clients can remove the same tuple) is to execute all *inp* operations in the same order in all servers. This can be implemented using a total order multicast protocol based on the *Byzantine Paxos* algorithm, e.g. *BFT* [1]. BFT works briefly as follows. When a client wants to multicast a message m, it sends m to all servers. When the leader server s receives m, it gives it the next *sequence number* i and sends it to all the other servers. If server s' receives $\langle m, i \rangle$ from the leader, it has previously received m from the client, and it accepted no previous message with sequence number i, then s' *accepts* m. When this happens, s' engages in two rounds of message exchange with the other servers to do agreement on the association $\langle m, i \rangle$. When agreement is reached, m is defined

as the i-th message to be delivered by correct servers. If some servers detect that the leader is faulty (e.g. because it leaves gaps in the sequence numbers), they elect another leader. LBTS' *inp* protocol is a modified version of BFT. It differs from BFT in three aspects: *(1)* when the leader s receives a $inp(\bar{t})$ request, it sends to the other servers not only the sequence number for this message but also a tuple $t_{\bar{t}}$ from its local tuple space that matches $\bar{t}$; *(2)* each server s' accepts to remove the tuple $t_{\bar{t}}$ received from the leader if the BFT conditions for acceptance are met, s' did not previously accepted the removal of $t_{\bar{t}}$, the tuple $t_{\bar{t}}$ matches the given template $\bar{t}$, and $t_{\bar{t}}$ is not forged (s' has $t_{\bar{t}}$ in its local tuple space or s' received $f+1$ signed messages from different servers ensuring that they have $t_{\bar{t}}$ in their local tuple spaces); *(3)* when a new leader l' is elected, each server sends it its protocol state 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) or $\perp$ (in case it does not). This modified version of BFT ensures total order in all *inp* executions and that the result of a *inp* is the same in all correct servers.

3 Discussion

This paper presents the first quorum-based construction for a shared memory object strictly stronger than a register. This construction is based on a combination of common quorum techniques plus three novel ones: *(i.)* instead of using timestamps, it uses a novel technique suited for collection objects, i.e., objects that store collections of elements, where the elements space is partitioned between all clients, and every element is unique; *(ii.)* it uses the listener communication pattern to capture the state of the system between executions of the read-write operations, and then apply the usual quorum-based reasoning to define the result for a read operation; and *(iii.)* it uses a modified Byzantine Paxos algorithm to do total order multicast and reach agreement about the result of an operation in a single execution. LBTS is more efficient in terms of message complexity and communication steps than a similar object would be if implemented directly on top of a BFT.

As future work, we expect to generalize the techniques used to design LBTS to define a replication algorithm that can be used to implement any shared object with consensus number greater than 1.

References

1. M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.
2. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.
3. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
4. J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the 16th Int. Symposium on Distributed Computing, DISC 2002*, volume 2508 of *LNCS*, pages 311–325, Oct. 2002.