Enhancing Efficiency of Byzantine-Tolerant Coordination Protocols via Hash Functions

Daniela Tulone

Department of Computer Science
University of Pisa
Italy

Abstract. Distributed protocols resilient to Byzantine failures are notorious to be costly from the computational and communication point of view. In this paper we discuss the role that *collision – resistant* hash functions can have in enhancing the efficiency of Byzantine – tolerant *coordination protocols*. In particular, we show two settings in which their use leads to a remarkable improvement of the system performance in case of *large data* or *large populations*. More precisely, we show how they can be applied to the implementation of atomic shared objects, and propose a technique that combines randomization and hash functions. We discuss also the *earnings* of these approaches and compute their complexity.

1 Introduction

The widespread use of the Internet and the proliferation of on-line services involving sensitive data, has lead to increasing attacks against the infrastructure. This motivates the growing interest in the design of distributed protocols resilient to *arbitrary failures*. These failures, often referred as Byzantine failures, model well both malicious behavior and software bugs, since they make the process diverge arbitrarily from the protocol specification. For instance, a Byzantine process can send arbitrary messages, remain latent for a while and then mount a *coordination attack* with other malicious nodes. It is clear that asynchronous Byzantine-tolerant protocols play a crucial role in the implementation of systems that must stay always available and correct. However, commercial applications often implement protocols that are resilient only to crash failures rather than to Byzantine failures, because they are more efficient though less robust. Clearly, coping with Byzantine failures increases the complexity of the protocol. For example, the number of nodes required to guarantee liveness and safety is larger than in case of crash failures, and this affects the communication and computational complexity. The computational cost of masking faulty responses is costly in case of *large data* or *large populations*, since it involves comparisons.

This paper proposes two settings in which collision-resistant hash functions enhance the system performance in case of large objects and large populations. It is well-known that hash functions allow to represent any stream of data by means of a fixed-size string (i.e 128 bit-string) called *message digest*. This identification can be considered unique if the function is *strongly collision-resistant* that is, if it is *computationally infeasible* to find two different objects with the same mapping. Clearly, replacing an object by its message digest enhances the performance of the system, optimizes its resources, such as memory utilization and network bandwidth, and decreases the computational cost of masking faulty values. Hash functions have been applied to a number of fields to enhance efficiency, such as network protocols, peer to peer systems [12], web caching [13]. For instance, Broder and Mitzenmacher [12] proposed an approach for obtaining good hash tables on using multiple hashes of each input key to improve the efficiency of IP address look-ups. However, our paper diverges from these works because it focuses only on collision resistant hash functions (i.e. MD5, SHA, HMAC). These functions have been applied to *secure data dispersal* by Krawczyk [11] to verify the integrity of data stored at servers that might be malicious. Castro and Liskov in [5] apply hash functions to enhance the efficiency of their atomic protocol resilient to Byzantine failures. More precisely, they use Message Authentication Codes (keyed hash functions) to authenticate messages instead of public key signatures, and in the last phase of their protocol servers return a digest reply while a *designated server* returns the full object. Notice that this approach is *optimistic* since it improves the system performance only if the designated server is correct, in case of malicious failure the protocol turns to a slower mode. Our technique improves this idea because it does not only replace large replies with their message digests, but *embeds* hash functions in the design of coordination protocols, in particular atomic protocols, thus exploiting their inherent properties. This approach leads *always* to a significant improvement of the network bandwidth consumption and CPU overhead. Hence, our contribution consists in embedding hash functions in the design of Byzantine-tolerant atomic protocols to improve their efficiency in case of large data and large populations, and in analyzing their benefits over the previous approach. Notice that the techniques we propose are *general* and their applicability goes beyond the settings presented in this paper. We are not aware of similar use of hash functions in previous Byzantine-tolerant protocols. Our study was originally motivated by the performance evaluation of the Fleet system [4], a middleware implementing a distributed data repository for persistent shared objects resilient to malicious attacks against the infrastructure. Fleet is built on top of Byzantine quorum system techniques [3] that make the system highly scalable and available, and improve the system load balancing and the access cost per operation since each operation is performed across a *subset* of servers (quorums). Fleet is targeted to highly critical applications, for instance it was used to implement a prototype of an electronic voting application [2]. While evaluating the performance of Fleet we noticed a remarkable performance degrade as the number of processes or the size of the object increased. This motivated us to look into the underlying Byzantine-tolerant protocols, and investigate ways to boost their efficiency. In this paper we show first how hash functions can enhance the efficiency of protocols implementing shared atomic objects, a fundamental building block for distributed systems. In particular, we propose an optimized version based on message digests, of the protocol proposed by Chockler et al [1]. The choice of analyzing [1] was motivated by its quorum approach that improves the system performance for large populations and makes the use of hash functions particularly suitable. In addition, we propose an approach that combines randomization and hash functions and that for its generality can be embedded in most distributed coordination protocols to enhance efficiency in case of large objects and large universe. We evaluate the complexity in both settings and discuss their performance impact.

2 Hash Functions Embedded in Atomic Protocols

2.1 An Overview

In this section we show how hash functions can be embedded in the implementation of atomic shared objects with *linearizable semantics* [10] to enhance efficiency and guarantee safety. A replicated object with linearizable semantics behaves as one single object exists, and the sequence of the operations applied on it is consistent with the real-time order of the invocations and responses.

Our system model is asynchronous and consists of a static set of n servers, and a limited but unknown set of clients dispersed in a WAN. Clients can fail by crash and a fraction of b servers can be compromised. Since it is impossible in an asynchronous system to distinguish between a crash failure and a slow process, progress should rely on $n - b$ replies. As a consequence, servers might have different views of the system.

A protocol implementing a fault-tolerant atomic object can be decomposed in two building blocks [9]: a *leader election* protocol to choose a coordinator, and a *3-phase commit* protocol run by it. Notice that the 3-phase commit protocol is necessary to guarantee data correctness since a leader failure could leave the system in an inconsistent state. The leader collects in the first phase of the commit protocol data regarding pending operations, and the current state of the object. It orders the operations to be performed based on such data (it may apply them) and then, proposes and commits them. If a leader crashes during the execution of the commit protocol leaving the system in an inconsistent state, the next leader needs to re-establish consistency among the replicas. This can be done by completing the previous run based on data transmitted to servers prior to crash. For instance, if a leader crashes after the proposed phase and before contacting $b + 1$ servers, the next coordinator is unable to recover such data, unless it is self-verifying (i.e. by means of digital signatures). In fact, usually a process retrieves the most up-to-date value by computing the most recent value returned by at least $b + 1$ servers. To overcome this problem, the coordinator sends same information in the propose and in the commit phase. Clearly, this guarantees safety but increases the system overhead in case of large object states.

Such a symmetry can be broken by transmitting in the proposed phase only the message digest of the *full data* sent in the commit phase and eventual additional information. The idea is to reestablish consistency among the replicas by enabling the coordinator to retrieve the data sent by a previous faulty coordinator or to recompute it by means of previous state and additional data. This idea will be developed in the next subsection and become part of the optimizations of protocol [1]. Due to lack of space, we refer the reader to [1] for details on the protocol by Chockler et al, and to the technical report [2] for an analysis of [1] and details and correctness proofs of this optimized version.

2.2 A Specific Scenario: Ordering Operations

The main novelty of the protocol by Chockler et al [1] over previous atomic protocols resilient to Byzantine failures such as [5][8], lies in its quorum system approach that enhances its efficiency in case of large populations. The protocol uses Byzantine quorum systems [3], a variation of quorum systems introduced by Malkhi and Reiter. A Byzantine quorum system is a collection of subsets of servers (*quorums*) such that any two subsets intersects in at least $2b+1$ servers (*consistency*), and for any b faulty servers there is a quorum set containing only correct servers (*availability*). Communications are performed via quorums: only a subset of servers is accessed each time. As a result, some servers have out-of-date object state. This increases the complexity of processing data at the leader side, and makes the use of message digest particularly suitable in this case, since it saves a number of unnecessary comparisons. The choice of analyzing this protocol rather than others is given by its high scalability to large populations (our focus), its generality (i.e. it supports also *non-deterministic operations*), and its efficiency due to quorum systems. Notice that its design improves the client response time since concurrent clients do not have to wait to become a coordinator in order to receive a return result.

```
1)  leader ← contend()
2)  if ($leader$)
3)      $\{\sigma_i^c, \sigma_i^p, ops_i\}_{i \in Q}$ ← getStatus()
4)      $(\sigma^c, \sigma^p)$ ← computeState()
5)      if crash in commit phase
6)          $\sigma \leftarrow \sigma^c$
7)      else if crash in proposed phase
8)          $\sigma \leftarrow \sigma^p$
9)      else
10)         $pendings$ ← computeOps()
11)         $\sigma$ ← apply($pendings, \sigma^c$)
12)     propose($\sigma$)
13)     commit($\sigma$)
14) fi
```

Fig. 1. Client side original version.

```
1)  $\langle leader, \{\sigma_i^c, \sigma_i^p, ops_i\}_{i \in Q} \rangle$ ← status()
2)  if ($leader$)
3)      $(\sigma^c, \sigma^p)$ ← computeState()
4)      if crash in commit phase
5)          $\sigma^c$ ← lastCommitted()
6)      else if crash in proposed phase
7)          $\sigma \leftarrow \sigma^p$
8)      else
9)          $pendings$ ← computeOps()
10)         $\sigma^p \leftarrow \langle$ hash($\sigma^c$), $pendings \rangle$
11)         $\sigma^c$ ← apply($pendings, \sigma^c$)
12)     propose($\sigma^p$)
13)     commit($\sigma^c$)
14) fi
```

Fig. 2. Client side optimized version.

An Overview of the Original Version. The protocol [1] works by applying operations to an object state to produce a *new state* and a return result. An object state σ, is an abstract data type containing the shared object, a sort of compressed history of the operations applied, and the return results for the last operations applied. The linearizability of the operations applied, is enforced by the order in which the object states are produced. The client side of the protocol consists of two concurrent threads: a non-blocking thread that simply submits an operation to quorum and waits for $b + 1$ identical responses, and a thread that runs a leader election protocol and a *3-phase commit* protocol. An high level description of the client side of this thread is sketched in Figure 1; σ_i^c(or σ_i^p) denotes the object state that server S_i received in the last commit (or last propose), and ops_i the client requests received by S_i and still pending according to its local view. The client runs a leader election protocol to access replicas for some fixed time units, line 1:1. If the majority of the correct servers in the quorum grants such permission, the client executes the 3-phase protocol. It first collects information regarding the last proposed and commit phase, line 1:3 and, based on these data detects possible inconsistency among the replicas due to client crash. In case of leader failure, it completes the previous execution lines 1:5-6, 1:7-8, otherwise applies all the operations that have been submitted but not yet applied to the last object state, thus generating a *new object state*, lines 1:10-11. Then, it proposes this new state to quorum and commits it, line 1:12-13. Notice that in this protocol the proposed object state σ^p is equal to the commit state σ^c. An analysis of this protocol sketched in the next subsection, indicates the object state and the out-of-date data sent by servers that have not been recently contacted, as the major performance bottleneck. Indeed, because of the quorum system and the asynchronous system, the list of pending operations at each server can grow unbounded since a server can receive requests without being contacted in the commit phase. Since Fleet mechanisms rely on this protocol [1], this justifies the performance degrade observed by us in Fleet in case of large objects and large populations. We refer the reader to [2] for a detailed analysis of this protocol (communication and computational complexity). Partial results on the performance evaluation of Fleet can be found in [4].

Our Optimized Version. We propose an optimized version of the protocol whose efficiency is obtained by 1) introducing hash functions, 2) encapsulating the leader election protocol in the commit protocol and 3) reducing out-of-date data. An high level description of this optimization can be found in Figure 2; the variations consist in lines 2:1, 2:5, 2:10 and 2:12-13. Notice that *getStatus*() is piggybacked to the leader election protocol, line 2:1. The use of message digests lets us break the symmetry between the committed and the proposed state: in fact, the proposed state σ^p consists only of the message digest of the *previous committed* state and a signature of the operations applied in that run, line 2:10. In this version each time a coordinator applies operations, it generates a new σ^p and σ^c, lines 2:10-11. In case the coordinator crashes right after the proposed phase, the next leader is able to compute σ^p because the proposed phase was completed, and can compute the correspondent σ^c based on σ^p, line 2:5. Notice that for each σ^p there is one and only one committed state that immediately precedes it, and in case of inconsistencies the coordinator is able to retrieve a correct copy of it. It verifies the correctness of the previous committed state by means of its message digest contained in σ^p. Therefore, procedure *lastCommitted*() at line

2:5, looks for the most recent $b+1$ identical commit states, and if it cannot find them, computes the committed state correspondent to σ^p by: 1) retrieving the previous committed state (last state that was fully committed), and 2) applying to it the operations contained in σ^p. Details on the correctness proof can be found in [2].

Communication Complexity. The use of message digests reduces the communication complexity at least by a factor of $2(q-b)(1+|\sigma|-|ops|)$ with q size of the quorum and *ops* signature of operations applied by the coordinator. Notice that since malicious servers can transmit arbitrary data (though in this way they are easily detected), we consider only data transmitted by correct servers. The number of correct servers in a quorum is at least $q-b$. In addition, the number of operations applied depends on the degree of concurrency of the system; in absence of concurrent client requests it is equal to 1. To have a better feeling of the performance improvement, let us consider a Threshold quorum system [3] with $n = 101$, $b = 20$ and quorum size $q = 81$. If the shared object is a 100 Mbytes file and the compressed history of the operations up to that time is equal to 1Mbytes, and the size of the request is equal to 100 Kbytes, then our proposal reduces the data transmitted by at least 95 Gbytes. Notice that the communication complexity is also improved by saving one phase of the protocol and removing operations that have been already applied and that are stored at out-of-date servers. That is, it reduces the size of ops_i.

Computational Complexity. The workload of the protocol lies on the client side, in particular on *computeStatus()* at lines 1:4 and 2:3 and on *computeOps()*. Since their costs depend on the number of malicious and out-of-date servers contacted and on the data corruptness, we compute their computational complexity in the best and worst case for both versions of the protocol. Here σ represent the full object state. The computational cost in [1] is given by

$$\Omega\Big(q|\sigma| + \sum_{i\in Q} ops_i + b\sum_{o\in\mathcal{O}} |o|\Big) \qquad O\Big(b\; q|\sigma| + \sum_{i\in Q} ops_i + b\sum_{o\in\mathcal{O}\cup\mathcal{P}} |o|\Big)$$

with $\mathcal{O}$ set of pending operations and $\mathcal{P}$ set of out-of-date operations. In the optimized version the complexity is

$$\Omega\Big(b|\sigma| + \sum_{i\in Q} ops_i + b\sum_{o\in\mathcal{O}} |o|\Big) \qquad O\Big(b^2|\sigma| + \sum_{i\in Q} ops_i + b\sum_{o\in\mathcal{O}} |o|\Big)$$

Notice that since in the optimized version the proposed state is very small in most cases, the computational cost is reduced almost by half. Clearly, this approach is not convenient in case the parameters of the operations are greater than the object itself, but this case is uncommon. Evaluation data performed by us on Fleet, running this optimized version of the protocol, showed an improvement over the previous protocol [1] by a factor 10 for large objects (i.e $\geq$ 314 Kbytes) and large universe (i.e. 80 servers).

2.3 Performance Impact

It is worth noticing that the computation of hash functions such as SHA, HMAC, MD5, is very fast and that in our proposal the hash function is computed only in two cases: when an object is generated or updated, and when a coordinator crashes during the commit phase leaving the replicas in an inconsistent state. In addition, from a system point of view, each time that an object is transmitted it is serialized and then deserialized by the receiver to detect the most up-to-date object. An analysis of Fleet performance running protocol [1] with different object sizes and on a Java profiler [14] to find performance bottlenecks, identified the serialization and deserialization of the objects transmitted and received as main *hot spots* with respect to memory and CPU usage and time. Notice that the use of message digests let us bypass this performance bottleneck. This explains also the remarkable performance improvement obtained in Fleet.

3 A Randomized Approach

In this section we propose an *optimistic approach* that combines randomization and hash functions and that has broad applicability. Optimistic protocols run very fast if no corruptions occurs but may fall back to a slower mode if necessary. Our idea is based on the following intuition: it is sufficient for a process to receive *one correct full* return value, provided a guarantee from other servers of its correctness (i.e its message digest). Clearly, this approach does not guarantee a correct return value since it is hard to distinguish a correct process from a malicious one. Randomization overcomes this problem: in case of very large objects each correct server replies by sending the full data and its message digest with probability p, and its message digest with probability $1-p$. Therefore, if the client contacts n servers the expected number of full correct replies is equal to $(n-b)\,p$. For $p=\frac{1}{2}$ this approach reduces by half the amount of data transmitted and for $n=3b+1$ the expected number of correct full copies is greater than b. To retrieve the full correct data, the process performs these two steps:

1. it computes the message digest of the data, denoted by msg, by using a majority voting;
2. it verifies the correctness of one of the full data received (randomly chosen) by computing its hash function and comparing it with msg. It returns as soon as it finds a correct full response.

It might happen that no correct server has sent a full response. In this case, the client falls back to a slower mode and requests to servers the full response. Notice that the system could adopt a dynamic strategy and tune p dynamically depending on the object size and system resources.

If the process receives at least one full correct response (common case if p is tuned appropriately), the communication complexity is improved at least by a factor of $(1-p)(n-b)|data|$. This optimization is more evident for large populations and of course for large objects. The analysis of the computational complexity is less straightforward. In fact, it is very efficient to compute *msg* in step 1, but verifying the correctness of the full data in step 2 is a bit more expensive than a simple data comparison. Therefore, it is significant to evaluate the number of hash computations necessary to find a correct full reply, in order to assess the performance improvement over previous solutions. In the worst case the process needs to compute $b + 1$ hash functions, this occurs when the coordinator picks first b corrupted data sent by malicious servers but with the correct message digest. Since the process chooses the data to verify uniformly at random, this bound is very pessimistic and is not representative for the actual computational cost of step 2. Therefore, we compute the *expected number* of hash computations performed by the process before finding a correct response. This gives us a measure of the computational cost in the average case. The following lemma computes an upper bound (not tight) of the expected number of hash computations in a more general case, when not all replicas are up-to-date. We refer to [2] for the proof.

Lemma 1. *If g is the number of correct up-to-date servers that are contacted, then the expected number of computations of the hash function in the worst case is less than $2 + \frac{b}{g}$.*

Notice that the expected number of g is equal to $p\ m$ with m number of the up-to-date correct copies returned by the servers contacted. Therefore, if $p = \frac{1}{2}$ then two computations of the hash function are enough on the average to detect a correct up-to-date reply.

4 Conclusions

In this paper we proposed two settings in which the use of hash functions leads to a remarkable improvement of system performance in a replicated system resilient to malicious attacks, and in case of large data and large populations. We have proposed two techniques, a deterministic and a randomized one, and applied them to Byzantine-tolerant atomic protocols. These techniques are general and can be applied to other settings requiring coordination among replicas, and for which both efficiency and safety are crucial.

Acknowledgments

The author would like to thank Mike Reiter for helpful discussions on protocol [1] and for his suggestion to explore a randomized approach.

References

1. G. Chockler, D. Malkhi, M. Reiter *Back-off protocols for distributed mutual exclusion and ordering.* In Proc. of the 21st International Conference on Distributed Systems, pp. 11-20, April 2001.

2. D. Tulone *On the Efficiency of Ordering Operations with Arbitrary Failures.* CNR ISTI Technical Report, April 2003.
3. D. Malkhi, M. Reiter, and A. Wool *The load and availability of Byzantine Quorum Systems.* SIAM Journal on Computing 29(6):1889-1906 (2000).
4. D. Malkhi, M.K. Reiter, D. Tulone, E. Ziskind. *Persistent objects in the Fleet system.* In Proc. of the 2nd Darpa Information Survivability Conference and Exposition (DISCEX II), June 2001.
5. M. Castro, B. Liskov. *Practical Byzantine Fault Tolerance and Proactive Recovery.* ACM Transactions on Computer Systems (TOCS), 20(4), pp. 398-461, November 2002.
6. M.K. Reiter. *Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart.* In Proc. of the 2nd ACM Conference on Computer and Communication Security, pp. 68-80, Novemebr 1994.
7. K.P. Kihlstrom, L.E. Moser, P.M. Melliar-Smith. *The SecureRing Protocols for Securing Group Communication.* In Proc. of the IEEE 31st Hawaii International Conference on System Sciences, (3):317-326, January 1998.
8. G. Chockler, D. Malkhi. *Active disk paxos with infinitely many processes.* In Proc. of the 21st ACM symposium on Principles of Distributed Computing (PODC 2002), pp. 78–87, July 2002.
9. L. Lamport *The part-time parliament.* ACM Transactions on Computer Systems (TOCS), 16(2), pp. 133-169, May 1998.
10. M.P. Herley, J.M. Wing *Linearizability: A correctness condition for concurrent objects.* ACM Transactions on Programming Languages and Systems 12(3):463-492, July 1990.
11. H. Krawczyk *Distributed fingerprints and secure information dispersal.* In Proc. of the 12th annual ACM symposium on Principles of distributed computing (PODC 1993), September 1993.
12. A. Broder, M. Mitzenmacher *Using Multiple Hash Functions to Improve IP Lookups.* In Proc. of IEEE INFOCOM 2001, pp. 1454-1463, April 2001.
13. David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, Daniel Lewin *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web.* Proc. of the 29th annual ACM symposium on Theory of computing (STOC 1997), May 1997.
14. Jprofiler, http://www.codework.com/jprofiler/product.htm.