BFT for Three Decades, Yet Not Enough!

Technical Report

IRIT/RR2012-7-FR

Ali Shoker and Jean-Paul Bahsoun
University of Toulouse III, IRIT Lab.,
Toulouse, France
firstName.lastName@irit.fr

November, 2009

Abstract
Distributed systems are established to maintain safety and liveness while attending good performance. Nowadays, Byzantine Failures are considered the most critical threat for system's safety. Various BFT protocols were found through the history; however, none has been adopted yet. The reason perhaps originates from the fact that Byzantine Fault Tolerance issue is hard by nature; add to this the complications underlying BFT protocols implementation.

This paper represents a general overview on Byzantine Fault Tolerance, its evolution, and difficulties disturbing its realization. First, we introduce the subject by defining replication, its importance and some problems; then we describe in brief the basic BFT protocols reporting some comparisons. Then we expose some problems that BFT protocols implementations are facing and we give a solution proposed by Guerraoui in [8]; finally, we summarize our paper and conclude.

1 Introduction

Software technologies must always be developed to satisfy market desires; one of the main technologies used in history is the *server-client* paradigm; usually a server delivers services requested by clients. This approach faced a lot of problems where systems are not able to afford the increasing demands of e-commerce and e-Banking; the fact is that servers were getting jammed with client's requests which rendered a faulty and very slow service.

Then, *replication* appeared as a solution where data (or services) are saved with multiple copies (or fragments) on different servers, this is managed by the *replication protocols*. Many replication protocols are then established to maintain availability and fault tolerance. These protocols differ in their characteristics; (a) they handle benign faults or *Byzantine* faults, (b) they are *active or passive*, (c) or they are *pessimistic or optimistic*... A lot of replication protocols addressed *benign* faults (these are mostly known by *fail-stop* faults). *Transactional* replication protocols handled database replication in [1], while *ROWA* [2] and other *tree-based* replication protocols [3, 4, 5, 6] recently achieved a good level of availability, fail-stop fault tolerance, and load. On the other hand, *Lamport* handled benign faults by consensus based protocols like *Paxos* [7]. Paxos proved its robustness and performance; *Google* developed *Chubby* system based on Paxos and used it in some of its applications; however, it is not being used widely in practice.

The increasing number of the *Web* Services, the diversity of programming languages; and software technology complexity in the last decade, together with hardware failures and human attacks, pushed

the *Byzantine Fault Tolerance* issue to the apex of worries. This forced research community to focus on the subject, probing for new protocols that can resolve data inconsistency while maintaining good performance despite the presence of *arbitrary* faulty servers and clients.

2 Byzantine Fault Tolerance

Byzantine faults are the collection of arbitrary faults where the system does not necessarily stop (in contrast with *fail-stop* faults). Distributed services may have different behaviors; it can lie, collude, delay responses, respond erroneously, and even does not respond at all; in short, these are the software that does not follow their specifications as designed. *Lamport* in his *"The Byzantine Generals Problem"* [9] from where Byzantine faults take their name was the first to address the *BFT* problem, and proved that the minimum number of servers consensus needed to solve the Byzantine Failures is *3f+1*. After this *Schneider* -theoretically- established the first Byzantine fault tolerant protocol based on *State Machine* replication [10], this *SMR* approach was then adopted by all research community intending to ensure one-copy semantics in a Byzantine faulty environments.

About ten years later, *M. Castro* and *B. Liskov* established the *Practical BFT protocol* [11] that represents the base of BFT protocols and probably the most famous one. They used consensus to achieve agreement of servers in order to tolerate any Byzantine client and 1/3 of the servers. The protocol used three phase message communication delays: *Pre-prepare* where the primary proposes a value, *prepare* where other servers agree on the value, and *commit*. *PBFT* protocol [11] relied on cryptography techniques in authentication and signatures; which imposed a bad impact on its performance.

Q/U protocol [12] appeared then as a quorum based protocol, it improved the performance of distributed systems by designating a set of servers to form the preferred quorum which reply to client's requests optimistically using only one phase of message communication; nevertheless *5f+1* servers are needed to tolerate f faulty servers. The protocol significantly improved the latency (when there is no contention) and exhibited fault scalability on the system, in other words, it imposed slow performance degradation as compared to agreement based protocols like PBFT [11].

Then the *Hybrid Quorum* protocol [13] (quorum-agreement based) was developed to benefit from the deficiencies of its ancestors in order to enhance the performance; it used the Q/U approach in the phases where there is no contention, and used PBFT approach as a recovery protocol in contention periods. The protocol required only *3f+1* servers to maintain fault tolerance when f servers can be Byzantine. In spite of the good performance attained by HQ in the presence of contention and the fault scalability it achieves, some shortcomings appear in contrast to its predecessors i.e. Q/U achieves lower latency than HQ whenever *5f+1* servers usage is possible and there is no contention, and PBFT performs better while using batching techniques of multiple sizes (since HQ could not use batching).

R. Kotla and *L. Alvisi* then came up with *Zyzzyva* [14] as a solution to the above problems. Zyzzyva operates in two phases to achieve consensus, the first is *speculative* (optimistic) where all servers execute clients requests before making sure of servers agreement, this phase is executed in two phase message delays whenever there are no server failures. The other phase is launched to recover from failures in the first phase if some servers are Byzantine. Although this protocol shares similar techniques with PBFT [11] (consensus, 3f+1 servers, views, batches) however it has proven the best performance known among other protocols; Zyzzyva attained this by (1) following the *speculative* optimistic agreement approach, (2) communicating through only two round-trips message in cases deprived from failures, and (3) sometimes using *MAC* authentications and message digests instead of cryptographic signatures that usually impose a large CPU overhead in signatures construction and validation.

Even though Zyzzyva [14] is considered the state of the art of BFT protocols, it has not been yet adopted to be used; perhaps practice community is not yet convinced with what Zyzzyva (as well as its predecessors) provided as a solution for the Byzantine fail-

ures problem. One can understand these worries after admitting that (1) risks should be always avoided by practice community before moving to new technologies whenever dealing with critical services, and (2) BFT protocols are really complicated and hard to implement; we demonstrate this in detail in what follows. Afterwards we study *Guerraoui's Aliph* approach proposed to simplify BFT implementations and always achieving good performance and robustness.

3 BFT Protocols Realization Obstructions

Despite the enormous growth of Web users and Web services providers, and on the other hand the increasing software complications, and human attacks, BFT protocols are seldom adopted as replication fault tolerance protocols; of course this refers to many reasons; some are related to the roots of BFT issue, and others to the BFT protocol techniques used that consequently imposed many implementation difficulties and obstacles.

Working on BFT for three decades, and counting a big number of researchers in the last decade without delivering a "convincing" solution to practice community indicates the difficulty of BFT by nature. One might suffer to construct a protocol that operates under some pre-defined conditions, optimizing message communication, authentication, agreement, and recovery. Also proving the correctness of the new protocol might require a PhD thesis [8].

Nevertheless BFT protocols are written in few number of pseudo code pages, their implementations require thousands of lines of code. *Guerraoui* states in [8] "All protocol implementations we looked at involve around 20.000 lines of (non-trivial) C++ code, e.g., PBFT and Zyzzyva". Many reasons underlie these complications. First, some optimizations should be used adapting to network architecture and hardware used; essage collisions frequency, network delays, and storage devices speed and robustness, are all examples. Add to this the different software behaviors and bugs; operating systems process scheduling, their communication protocols TCP/IP, UDP ... All these differences have to be handled while implementation. We recall that the diversity of these hardware and software comes from the fact that BFT are built on *commodity* systems to reduce the cost required by large severs and infrastructures.

Again, various implementation complications are related to the protocol itself. Most of the protocols optimized the *trivial* case (where there are no server failures); however, it lunches recovery phase whenever failures occur; these recovery phases are usually more complicated and require a greater number of message communications, cryptographic authentications,clients-server communication, and clients history recovery and validation. Also the concepts of views in BFT protocols imposed some difficulties especially on *view-change* after detecting current primary server failure. Add to this using requests batching to improve performance. Zyzzyva protocol [14], the state of the art of BFT protocols, is a good example of the above. Other implementation optimizations were also appended to Zyzzyva appeared in [14], we mention here: *Separating Agreement from Execution, Read-Only Optimization, Single Execution Response, and Preferred Quorums.*

BFT protocol's testing is not easy too. First, the notion of arbitrary-failures is not obvious and hard to be simulated. Second, the multiple conditions under which a protocol operates drives the tester to predict a huge number of scenarios and use-cases. Add to this that testing follow-up under these BFT protocol's numerous techniques and phases turns to be a nightmare. In brief, no testing would be enough, complete, or lead to satisfaction in such kind of BFT protocols.

4 *Aliph*, might be the promising future

The difficulties - shown above - underling BFT protocols realization promoted researchers to follow other approaches that are easy to implement while ensuring system's robustness and at the same time maintaining a good performance. *R. Geurraoui*

proposed in [8] Aliph as a new BFT approach (the term *Aliph* was induced by the author in the new version of the paper [8]). We describe briefly how Aliph works, what its properties are, and then we show its importance.

Aliph is a general abstraction where various BFT protocols form a composition of instances of this abstraction, each instance developed and analyzed independently. Aliph can be viewed as a generic super protocol managing sub-protocols in a modular fashion; a module (i.e. instance or sub-protocol) is composed of a single BFT protocol; each of these instances operates under specific constraints (*non-triviality* conditions) and provides the desired performance under these circumstances; whenever an instance aborts (by failing or forced to stop) it constructs a request *abort history* corresponding to that request. Upon this failure, Aliph launches another predefined BFT instance called *Backup*, including the *unforgeable abort history* collected (and signed) by the previous failed *abstract* (i.e. instance) as an INIT request to Backup to initialize its history. Backup usually executes a certain number of client requests according to a *switching policy* SP and then tries to resume the previous Abstract to benefit from its performance. Usually abstracts with week non-triviality conditions -called *aAbstract*- are most likely to be executed (incase non-triviality conditions are satisfied) since it performs better than Backups (that has strong non-triviality conditions). Of course, a Backup has to collect request histories before exiting, in order to construct an INIT request again to be delivered to the designated aAbstract (See Figure 1 and Figure 2 below).

As stated above, Aliph's main component is Abstract (i.e. either aAbstract or Backup); the most these Abstracts are various the most that Aliph can be feasible and effective. Thus, it is very important to establish new instances with different non-triviality conditions including synchronization (eventually), contention, no server failures, connection delays, and others. The author in [8] constructed two Abstract instances Quorum and Chain:

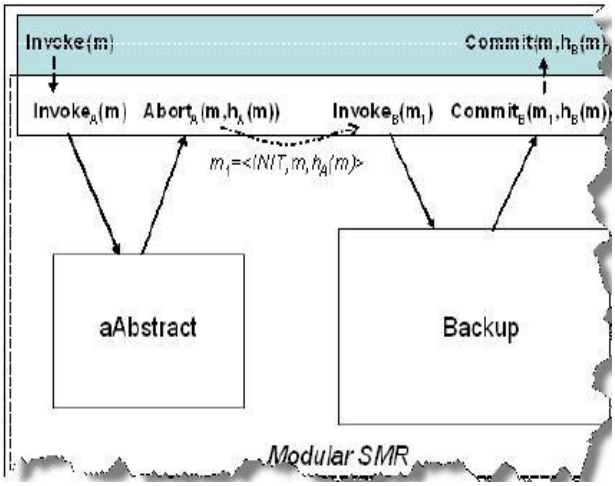

Figure 1: If aAbstract aborts, clients use the abort history as init history to switch to Backup. Backup is a powerful that guarantees to commit a certain number of requests.

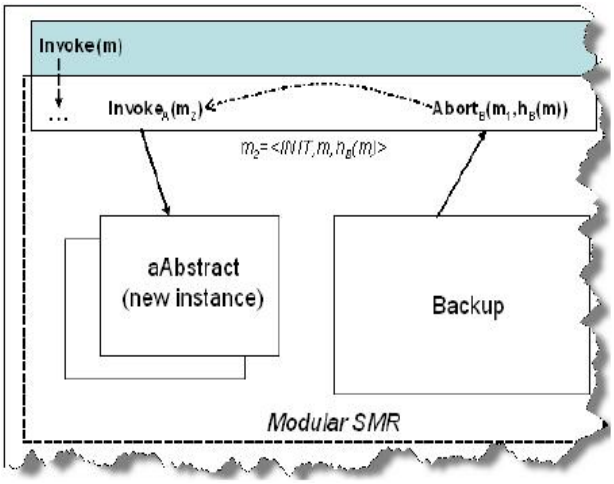

Figure 2: After a certain number of requests is delivered within Backup, a client may be switched-back to try again (some) aAbstract.

Quorum has the following non-triviality property: If a correct client c invokes request m, and (a) there are no server failures, (b) the system is synchronous and (c) there is no contention, then client c commits m. Under these conditions Quorum has the lowest latency among all BFT "known" protocols.

Chain has the following non-triviality property: If a correct client c invokes a request m, and (a) there are no server failures and (b) the set of server is synchronous, then client c commits m. Under these conditions Chain has the highest peak throughput among existing BFT protocols.

Here is an example on Aliph. We may implement aAbstract using Quorum or Chain and we implement Backup using PBFT; the protocol runs the aAbstract instances (Quorum or Chain) if non-triviality properties are satisfied, else Backup (PBFT) is launched after initializing its history using the unforgeable abort history request generated by the previous Abstract; PBFT operates for a predefined number of runs K, and then aAbstract is launched again to benefit from its performance (latency or throughput).

Now we show the properties and importance of the proposed approach. Aliph reconstructed the design of BFT protocols in order to simplify its implementation. This could be easily noticed since no complicated phases are needed anymore like recovery phases or view change. Simply, whenever a BFT instance fails, Aliph initiates a Backup BFT instance using the preceding abort-history. For instance, *Zyzzyva-like* Abstract was also established in [8]; this abstract mimics Zyzzyva's behavior in synchronous and failure-free executions (i.e. the first phase); it also proves that rebuilding Abstracts from existing BFT protocols is possible and easy. It is important to reveal that Zyzzyva-like Abstract required only 5000 lines of C++ code (while Zyzzyva requires about 20000 lines). The other Abstracts Quorum and Chain required 4000 and 5000 lines of C++ code respectively.

Any change to a protocol, although algorithmically intuitive, is extremely painful. In some cases, the changes of the protocol needed to optimize for the "normal" case have strong impacts on the part of the protocol used in other cases (e.g., view-change in Zyzzyva) [8]. Approving that No protocol fits all; no existing protocol could adapt to many situations, conditions, or periods. However, Aliph could accommodate to new circumstances by the variety of Abstract components it has (Quorum, Chain, Zyzzyva-like) and many others that can be appended. Of course this is allowed by the modular nature of Aliph, and the dynamic switching from one abstract to another. Hence, this modularity gives Aliph superiority on other protocols by (1) Implementation simplicity, (2) adapting to new situations, and (3) extendibility.

Finally, we note that Abstract instances in Aliph were constructed in an optimal way benefiting from different message communication fashions, using message digests to reduce payload, and lightweight authentication vectors to diminish CPU overheads.

5 Conclusion

After explaining briefly BFT replication, visiting the basic protocols found in history, and comparing their features and conditions, we came up with the conclusion that BFT protocols face many implementation obstacles. We have shown then how Aliph simplifies this task, and we exposed its superiority on its ancestors.

Aliph might need more enhancements to reach maturity. Several directions can be interesting to explore with Abstract in mind. It would be interesting to devise Abstract implementations for other meaningful definitions of the non-triviality property. In particular, it might be interesting to develop Abstract implementations that perform well even under failures, by assuming for instance 5f + 1 replicas. There is also a room for optimizing switching mechanism between Abstract instances; it could for instance be improved by implementing inter-replicas communications, rather than having all communications going through the client. Finally, we believe that an interesting research challenge is to define and evaluate heuristics for dynamically switching from an Abstract implementation to another one in order to improve performance [8].

References

[1] J. Gray , P. Helland, P. ONeil, and D. Shasha; *The Dangers of Replication and a Solution*; January 1996

[2] P. Bernstein and N. Goodman; *An algorithm for concurrency control and recovery in replicated distributed databases.* ACM Transactions on Distributed Systems, 9 (4), 1984.

[3]. D. Agrawal and A. E. Abbadi. *The tree quorum protocol: An efficient approach for managing replicated data.* Proceedings of the 16th VLDB Conference, pages 243254., 1990.
[4] S. Choi, H. Youn, and J. Choi. *Symmetric tree replication protocol for efficient distributed storage system.* ICCS, pages 474484, 2003.
[5] A. Kumar. *Hierarchical quorum consensus: A new algorithm for managing replicated data.* IEEE Transactions on Computers, pages 9961004, 1991.
[6] J-P. Bahsoun, R. Basmadjian, and R. Guerraoui; *An Arbitrary Tree-Structured Replica Control Protocol*;The 28th International Conference on Distributed Computing Systems 2008
[7] L. Lamport; *The Part-Time Parliament*;ACM Transactions on Computer Systems; 133-169 (May 1998).
[8] R. Guerraoui, V. Quema, and M. Vukolic ;*The Next 700 BFT Protocols*; Technical Report LPD-REPORT-2008-008
[9] L. Lamport, R. Shostak, and M. Pease; *Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982.
[10] F. B. Schneider; *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*; Department of Computer Science, Cornell University, Ithaca, New York ; 1990
[11] M. Castro and B. Liskov. *Practical Byzantine fault tolerance.* In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, Feb. 1999.
[12] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. *Fault-scalable Byzantine fault tolerant services.* In Proceedings of the 20th ACM symposium on Operating systems principles, pages 5974, Oct. 2005.
[13] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. *HQ replication: A hybrid quorum protocol for Byzantine fault tolerance.* In Proceedings of the 7th Symposium on Operating Systems Design and Implementations, Nov. 2006.
[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. *Zyzzyva: speculative Byzantine fault tolerance.* In Proceedings of 21st ACM symposium on Operating systems principles, pages 4558, NewYork, NY, USA, 2007.