Modern Large-Scale Data Management Systems after 40 Years of Consensus

Mohammad Javad Amiri, Divyakant Agrawal, Amr El Abbadi
University of California Santa Barbara
Santa Barbara, California
{amiri, agrawal, amr}@cs.ucsb.edu

***Abstract*—Modern large-scale data management systems utilize consensus protocols to provide fault tolerance. Consensus protocols are extensively used in the distributed database infrastructure of large enterprises such as Google, Amazon, and Facebook as well as permissioned blockchain systems like IBM's Hyperledger Fabric. In the last four decades, numerous consensus protocols have been proposed to cover a broad spectrum of distributed database systems. On one hand, distributed networks might be synchronous, partially synchronous, or asynchronous, and on the other hand, infrastructures might consist of crash-only nodes, Byzantine nodes or both. In addition, a consensus protocol might follow a pessimistic or optimistic strategy to process transactions. Furthermore, while traditional consensus protocols assume a priori known set of nodes, in permissionless blockchains, nodes are assumed to be unknown. Finally, consensus protocols have explored a variety of performance trade-offs between the number of phases/messages (latency), the number of required nodes, message complexity, and the activity level of participants. In this tutorial, we discuss consensus protocols that are used in modern large-scale data management systems, classify them into different categories based on their assumptions on network synchrony, failure model of nodes, etc., and elaborate on their main advantages and limitations.**

***Index Terms*—Fault Tolerance, Consensus, Data Management**

I. Introduction

On April 1980, Pease, Shostak, and Lamport addressed the problem of consensus in the presence of faults for the first time in the domain of distributed systems [52]. In this fundamental problem, a set of distributed nodes need to reach agreement on a single value [40]. Modern large-scale data management systems such as cloud and blockchain rely on consensus protocols to provide robustness and performance. In cloud data management systems, such as Google's Spanner [21], Amazon's Dynamo [24], and Facebook's Tao [14], consensus protocols are extensively used to enhance fault tolerance. Consensus is also the core component of the most recent large-scale data management system, *Blockchain*. In particular, permissioned blockchain systems, such as IBM Hyperledger [7], Quorum [17], Fast Fabric [29], SharPer [5], ResilientDB [33], and Caper [3], use consensus protocols to establish agreement on the order of transaction blocks among a set of known, identified nodes that might not fully trust each other. In the last four decades, numerous consensus protocols have been designed to satisfy the two main requirements of large-scale data management systems, robustness and performance, using State Machine Replication (SMR) [39] techniques. Robustness is the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures, while performance deals with the response time of requests (latency) and the number of processed requests per time unit (throughput) [8].

In this tutorial, we study consensus protocols in the domain of large-scale data management systems based on five aspects: (1) synchrony mode, (2) failure model, (3) processing strategy, (4) participants type, and (5) performance metrics.

Synchronous distributed systems assume known bounds on message delays and process speeds [46]. In synchronous systems, all communication proceeds in rounds. In one round, a process may send all the messages it requires, while receiving all messages from other processes. In this manner, no message from one round may influence any messages sent within the same round. On the other hand, in *asynchronous* distributed systems, there are no bounds on the amount of time a node might take to complete its work and then respond with a message [46]. In such systems, there is no global clock nor consistent clock rate, each node processes independently of others, and coordination is achieved via events such as message arrival. As shown by Fischer et al. [28], in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, most fault-tolerant protocols satisfy safety in an asynchronous network that can drop, delay, corrupt, duplicate, or reorder messages, however, consider a synchrony assumption to satisfy liveness. Finally, *partially synchronous* systems take the position between asynchronous systems where delays can be arbitrarily large and synchronous systems where there is a bound on message transmission and processing delays. A partially synchronous model assumes that among the nodes, there is a subset that can communicate in a timely manner and only a limited number of nodes are perceived as arbitrarily slow, due to either message transmission or processing delays [30]. This assumption indeed is reasonable in data centers which are more predictable and controllable than an open Internet environment.

Each node in a distributed system follows either the crash or malicious failure model. In the crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart, however, they may not collude, lie, or otherwise attempt to subvert the protocol. Whereas, in the malicious failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior. Crash fault-tolerant protocols, e.g., Paxos

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00172

[41], guarantee safety in an asynchronous network using $2f+1$ nodes to overcome the simultaneous crash failure of any f nodes while in Byzantine fault-tolerant protocols, e.g., PBFT [16], $3f+1$ nodes are needed to provide the safety property in the presence of f malicious nodes [12]. A hybrid failure model admits both crash and malicious failures. Indeed, in a hybrid network, some nodes might crash whereas some others behave maliciously. Hybrid fault-tolerant protocols, e.g. UpRight [19] and SeeMoRe [6], assume a known bound on the maximum number of crash and malicious failures.

Consensus protocols might *optimistically* assume that the nodes are well-behaved. As a result, nodes speculatively execute requests without running an agreement protocol to definitively establish the order. Such an assumption while reducing the processing time, might result in diverge states of correct nodes that need to be detected and resolved. Pessimistic consensus protocols, on the other hand, are robust and designed to tolerate the maximum number of possible concurrent failures f (where f is defined based on the failure model of nodes).

Traditional consensus protocols assume that the participants are known and identified and make an assumption on the maximum number of failures, f, in the system. In permissionless blockchain systems, e.g., Bitcoin [50], however, the set of participants is assumed to be unknown. As a result, none of the existing protocols can be used, thus, mining has been proposed to solve the consensus problem.

Finally, consensus protocols explore a spectrum of performance trade-offs between the number of required participants, number of phases/messages (latency), and message complexity. On the required number of participants, while it is known that in the presence of crash-only (malicious) nodes, $2f+1$ ($3f+1$) is needed to overcome the simultaneous failure of any f nodes, some approaches assume nodes are well-behaved (either optimistically or using techniques like trusted hardware) and reduce the number of required nodes. On the other hand and to decrease the number of required communication phases, increasing the number of required nodes is proposed for both crash-only [42] and malicious nodes [47]. Finally, to decrease the message complexity of Byzantine fault-tolerant protocols, increasing the number of communication phases and using advanced encryption techniques have been presented [63].

In this tutorial, our goal is to present to the database community an in-depth understanding of state-of-the-art solutions to design efficient consensus protocols that can be used by large-scale data management systems. We progress towards this goal by starting from a detailed description of techniques underlying the design of existing consensus protocols.

II. Tutorial Outline

Many practical large-scale data management systems such as ISIS [10], Eternal [49], Google's Spanner [21], Amazon's Dynamo [24], and Facebook's Tao [14], use consensus protocols to provide fault tolerance. Consensus algorithms are a form of State Machine Replication [39]. SMR regulates the deterministic execution of client requests on multiple copies of a server, called replicas, such that every non-faulty replica must execute every request in the same order [55] [39]. The SMR algorithm has to satisfy *safety* and *liveness* properties. Safety means all correct nodes receive the same requests in the same order whereas liveness means all correct client requests are eventually ordered.

Several approaches [55] [41] [51] generalize SMR to support crash failures among which Paxos [41] is the most well-known asynchronous protocol. Paxos guarantees safety in an asynchronous network using $2f+1$ nodes despite the simultaneous crash failure of any f nodes. In Paxos, clients send signed requests to *the primary* (a pre-elected node that initiates consensus) and the primary multicasts an accept message including the transaction to every node within the system. Upon receiving a valid accept message from the primary, a node sends an accepted message to the primary. The primary waits for f accepted messages from different nodes (plus itself becomes $f+1$), multicasts a commit message to every node, and sends a reply to the client.

Many protocols have been proposed to either reduce the number of phases, e.g., Multi-Paxos which assumes the leader is relatively stable or Fast Paxos [42] and Brasileiro et al. [13] which add f more nodes, or reduce the number of nodes, e.g., Cheap Paxos [43] which tolerates f failures with $f+1$ active and f passive nodes. Finally, Raft [51] is a leader based crash fault-tolerant protocol that was meant to be more understandable than Paxos.

Byzantine fault tolerance refers to nodes that behave arbitrarily after the seminal work by Lamport, et al. [44]. Early Byzantine fault-tolerant protocols (SecureRing [36] and Rampart [54]) were synchronous where a round based algorithm is developed to exclude faulty nodes from the group. Such systems are vulnerable to denial-of-service attack where an attacker may compromise the safety of service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the group.

Practical Byzantine fault tolerance (PBFT) protocol [16] is one of the first and the most known state machine replication protocol to deal with malicious failures in an asynchronous network. PBFT requires $3f+1$ nodes to guarantee safety in the presence of at most f *malicious* nodes. PBFT consists of *agreement* and *view change* routines where the agreement routine orders requests for execution by the nodes, and the view change routine coordinates the election of a new primary when the current primary is faulty. The nodes move through a succession of configurations called *views* [26] [27] where in each view, one node, which initiates the protocol, is *the primary* and the others are *backups*.

Although practical, the cost of implementing PBFT is quite high, requiring at least $3f+1$ nodes, 3 communication phases, and a quadratic number of messages in terms of the number of nodes. Thus, numerous approaches have been proposed to explore a spectrum of trade-offs between the number of phases/messages (latency), number of nodes, the activity level of participants (nodes and clients), and types of failures.

FaB [47] and Bosco [58] reduce the communication phases by adding more nodes. Speculative protocols, e.g., Zyzzyva

[37], HQ [23], and Q/U [1], also reduce the communication by executing requests without running any agreement between nodes and optimistically rely on clients to detect inconsistencies between nodes. To reduce the number of nodes, some approaches rely on a trusted component (a counter in A2M-PBFT-EA [18] or a whole operating-system instance [22]) that prevents a faulty node from sending conflicting (i.e., asymmetric) messages to different nodes without being detected. SBFT [31] and Hotstuff [63] attain linear communication overhead by increasing the number of communication phases and using advanced encryption techniques, e.g., signature aggregation [11]. Finally, MultiBFT [32] uses multiple parallel primary nodes to parallelize transaction processing.

Optimistic approaches reduce the required number of nodes during the normal-case operation by either utilizing the Cheap Paxos [43] solution and keeping f nodes in a passive mode (REPBFT [25]), or by separating agreement from execution [62]. In ZZ [61] both passive nodes and separating agreement from execution are employed. Note that all these approaches need $3f + 1$ nodes upon occurrence of failures. REMINBFT [25] and CheapBFT [34] use a trusted component to reduce the network size to $2f + 1$ and then utilize an optimistic approach by keeping f of those nodes passive during the normal-case operation. In contrast to optimistic approaches, robust protocols (Prime [2], Aardvark [20], Spinning [60], RBFT [9]) consider the system to be under attack by a very strong adversary and try to enhance the performance of the protocol during periods of failures.

Consensus with multiple failure modes were initially addressed in synchronous protocols [59] [48] [35] [57]. Recent protocols such as VFT [53], XFT [45], and SBFT [30] have focused on partial synchrony, a technique that defines a threshold on the number of slow (partitioned) processes. VFT is similar to PBFT regarding the number of phases and massage exchanges, however, it optimistically assumes that an adversary cannot fully control the malicious nodes and as a result, reduces the phases of communication and message exchanges. SBFT also reduces the number of message exchanges by assuming the adversary controls only crash failures. Scrooge [56], as an asynchronous hybrid protocol, uses a speculative technique to reduce the latency. UpRight [19] also utilizes the agreement routines of PBFT [16], Aardvark [20], and Zyzzyva [37] and, similar to [62], separates agreement from execution. SeeMoRe [6] is an asynchronous hybrid protocol that takes advantage of being aware of where the crash or malicious faults may occur and either reduces the number of communication phases and message exchanges by placing the primary in the crash-only private cloud, or decreases the number of required nodes by placing the primary in the untrusted public cloud.

Since permissioned blockchain systems consist of a set of known, identified nodes that might not fully trust each other, traditional Byzantine consensus protocols can be used to order the transaction blocks [15]. In Tendermint [38], only a subset of nodes, called validators, participates in a PBFT-like consensus protocol. Validators are users with accounts that have coins locked in a bond deposit and have voting power equal to the amount of the bonded coins. Quorum [17] uses a Raft-like [51] protocol to order transactions. In Hyperledger Fabric [7] and ParBlockchain [4] fault-tolerant protocols are pluggable and depending on the failure model of nodes a crash or a Byzantine fault-tolerant protocol can be used.

While traditional consensus protocols assume a priori known set of participants, in permissionless blockchain system the set of participants is assumed to be unknown. A permissionless setting allows participants to freely join and leave the system without maintaining any global knowledge of the number of participants. Since the participants are unknown, none of the existing protocols can be used to establish consensus on the order of transactions. To solve this problem, Bitcoin introduces *mining* where nodes need to solve a computationally challenging *Proof of Work (PoW)* puzzle before they can add any block of transactions to the replicated blockchain. Since the PoW puzzle is computationally hard, very few miners can successfully solve the puzzle, and hence a successful miner can add a block to the blockchain and be guaranteed, with very high probability, to be unique.

III. Tutorial Information

This is a **three hours** tutorial targeting researchers, designers, and practitioners interested in consensus and its applications in large-scale data management systems. The **target audience** with basic background about distributed systems should benefit the most from this tutorial. For the general audience and newcomers, the tutorial explains the design space of consensus in large-scale data management systems.

IV. BIOGRAPHICAL SKETCHES

Mohammad Javad Amiri is a PhD student at the University of California at Santa Barbara. His research mostly lies at the intersection of Data Management and Distributed Systems. The focus of his current research is on managing data in distributed infrastructures such as cloud and blockchain.

Divyakant Agrawal is a Professor of Computer Science at the University of California at Santa Barbara. His current interests are in the area of scalable data management and data analysis in cloud computing environments, security and privacy of data in the cloud, scalable analytics over big data, and Blockchain. Prof. Agrawal is an ACM Distinguished Scientist (2010), an ACM Fellow (2012), an IEEE Fellow (2012), and an AAAS Fellow (2016).

Amr El Abbadi is a Professor of Computer Science at the University of California, Santa Barbara. Prof. El Abbadi is an ACM Fellow, AAAS Fellow, and IEEE Fellow. He was Chair of the Computer Science Department at UCSB from 2007 to 2011. He has served as a journal editor for several database journals and has been Program Chair for multiple database and distributed systems conferences. Prof. El Abbadi was also the co-recipient of the Test of Time Award at EDBT/ICDT 2015. He has published over 400 articles in databases and distributed systems and has supervised over 35 PhD students.

Acknowledgement

This work is funded by NSF grants CNS-1703560 and CNS-1815733.

REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. *OSR*, 39(5):59–74, 2005.

[2] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE TDSC*, 8(4):564–577, 2011.

[3] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Caper: a cross-application permissioned blockchain. *VLDB*, 12(11):1385–1398, 2019.

[4] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *ICDCS*, pages 1337–1347. IEEE, 2019.

[5] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Sharper: Sharding permissioned blockchains over network clusters. *arXiv preprint arXiv:1910.00765*, 2019.

[6] M. J. Amiri, S. Maiyya, D. Agrawal, and A. El Abbadi. Seemore: A fault-tolerant protocol for hybrid cloud environments. *ICDE*, 2020.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, page 30. ACM, 2018.

[8] P. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *TOCS*, 32(4):12, 2015.

[9] P. Aublin, S. B. Mokhtar, and V. Quéma. Rbft: Redundant byzantine fault tolerance. In *ICDCS*, pages 297–306. IEEE, 2013.

[10] K. P. Birman, T. A. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering*, (6):502–508, 1985.

[11] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17(4):297–319, 2004.

[12] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[13] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *PaCT*, pages 42–50. Springer, 2001.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.

[15] C. Cachin. Architecture of the hyperledger blockchain fabric. In *DCCL Workshop*, volume 310, 2016.

[16] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[17] J. M. Chase. Quorum white paper, 2016.

[18] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *OSR*, volume 41-6, pages 189–204. ACM, 2007.

[19] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, et al. Upright cluster services. In *SOSP*, pages 277–290. ACM, 2009.

[20] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[21] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3):8, 2013.

[22] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *SRDS*, pages 174–183. IEEE, 2004.

[23] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.

[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, et al. Dynamo: amazon's highly available key-value store. In *OSR*, volume 41, pages 205–220. ACM, 2007.

[25] T. Distler, C. Cachin, and R. Kapitza. Resource-efficient byzantine fault tolerance. *Trans. on Computers*, 65(9):2807–2819, 2016.

[26] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *ACM SIGACT-SIGMOD symp. on Principles of database systems*, pages 215–229. ACM, 1985.

[27] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *SIGMOD*, pages 240–251. ACM, 1985.

[28] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[29] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*, pages 455–463. IEEE, 2019.

[30] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *DSN*, pages 568–580. IEEE, 2019.

[31] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks*, pages 568–580. IEEE/IFIP, 2019.

[32] S. Gupta, J. Hellings, and M. Sadoghi. Scaling blockchain databases through parallel resilient consensus paradigm. *arXiv preprint arXiv:1911.00837*, 2019.

[33] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *VLDB*, 13(6):868–883, 2020.

[34] R. Kapitza, J. Behl, C. Cachin, et al. Cheapbft: resource-efficient byzantine fault tolerance. In *EuroSys*, pages 295–308. ACM, 2012.

[35] R. M. Kieckhafer and M. H. Azadmanesh. Reaching approximate agreement with mixed-mode faults. *Trans. on Parallel and Distributed Systems*, 5(1):53–63, 1994.

[36] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii Int. Conf. on System Sciences*, volume 3, pages 317–326. IEEE, 1998.

[37] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *OSR*, 41(6):45–58, 2007.

[38] J. Kwon. Tendermint: Consensus without mining. *Draft v. 0.6*, 2014.

[39] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[40] L. Lamport. The part-time parliament. *Trans. on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[41] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[43] L. Lamport and M. Massa. Cheap paxos. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 307–314. IEEE, 2004.

[44] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[45] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. Xft: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[46] N. A. Lynch. *Distributed algorithms*. Elsevier, 1996.

[47] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215, 2006.

[48] F. J. Meyer and D. K. Pradhan. Consensus with dual failure modes. *Trans. on Parallel & Distributed Systems*, (2):214–222, 1991.

[49] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, and V. Kalogeraki. The eternal system: An architecture for enterprise applications. In *EDOC*, pages 214–222. IEEE, 1999.

[50] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[51] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[52] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[53] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *EuroSys*, page 8. ACM, 2015.

[54] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Sys.*, pages 99–110. Springer, 1995.

[55] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR*, 22(4):299–319, 1990.

[56] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *DSN*, pages 353–362. IEEE, 2010.

[57] H.-S. Siu, Y.-H. Chin, and W.-P. Yang. A note on consensus on dual failure modes. *IEEE TPDS*, 7(3):225–230, 1996.

[58] Y. J. Song and R. van Renesse. Bosco: One-step byzantine asynchronous consensus. In *DISC*, pages 438–450. Springer, 2008.

[59] P. Thambidurai, Y. Park, et al. Interactive consistency with multiple failure modes. In *SRDS*, pages 93–100. IEEE, 1988.

[60] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*, pages 135–144. IEEE, 2009.

[61] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. Zz and the art of practical bft execution. In *EuroSys*, pages 123–138. ACM, 2011.

[62] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.

[63] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Symp. on Principles of Distributed Computing*, pages 347–356. ACM, 2019.