A COMPARISON STUDY OF BYZANTINE FAULT TOLERANCE PROTOCOLS

BY

DANYANG ZHUO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Advisor:

Professor Nitin H. Vaidya

ABSTRACT

The Byzantine fault model is a worst-case assumption for faulty nodes in distributed systems because it assumes faulty nodes to behave arbitrarily. Previously, Lamport et al. proposed a replicated state way to solve the Byzantine broadcast problem. But the original protocol is very expensive due to large communication overhead. This thesis does a comparison study between practical Byzantine fault tolerance (PBFT) and network coding based algorithm (NCBA) which can solve Byzantine broadcast problem with lower cost. I implemented a generalized testing framework as well as the Digest protocol and the NCBA protocol. By the experiments I conducted, NCBA has comparable performance compared to Digest in fault-free cases.

To my parents, for their love and support

ACKNOWLEDGMENTS

I sincerely thank my advisor, Professor Nitin H. Vaidya, for his careful guidance of my research in University of Illinois, Urbana Champaign. He taught me how to do research and granted me freedom to explore new ideas. He helped me generate and polish ideas, exposed me to open problems and guided me to look at related papers. I am really fortunate to have received a lot of great insights from him. Also, I want to thank all the people I am working with, in particular Lewis Tseng, Philbert Lin, Larry Kai and Adrian Djokic.

Finally, I thank my parents for their constant support and encouragement for my life goal and my education.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis explores the cost of Byzantine fault tolerance (BFT) protocols. Byzantine fault tolerance assumes arbitrary behavior of the server and thus is a stronger fault model than the crash failure tolerance model. Lamport et al. [1] proposed the replicated state machine approach to address the Byzantine broadcast problem. Unfortunately, the original state machine approach has a high communication overhead and thus it is not practical to implement the original algorithm in real world distributed systems.

Later, all kinds of practical BFT protocols were designed to make Byzantine fault tolerance practical, such as practical Byzantine fault tolerance (PBFT) [2, 3], Q/U protocol [4], hybrid-quorum [5], Zyzzva [6], Aardvark [7], network coding based algorithm (NCBA) [8] and ZZ [9].

In my thesis, I did a performance comparison between the PBFT protocol [2, 3] and the NCBA protocol [8]. My results show that NCBA and PBFT are comparable in fault-free cases.

The problem I target on is Byzantine broadcast problem. A single source wants to broadcast a value to n-1 peers and meet the following requirements:

- Agreement
  Correct peers will agree on a value.
- Validity
  If the source is correct, correct peers will agree on the value the source broadcasts.
- Liveness
  The system is continuously making progress and will terminate.

Also, I assume a synchronous communication network for the experiment.

In Chapter 2, I describe previous work on Byzantine fault tolerance protocols. In Chapter 3, I present a system design to test existing Byzantine fault tolerance protocols. In Chapter 4, I analyze the results of the performances data. I state my conclusion in Chapter 5 and future work in Chapter 6.

CHAPTER 2

LITERATURE REVIEW

2.1 Byzantine Broadcast Problem

Lamport et al. proposed the Byzantine general problem [1] that n generals propose their 1-bit proposals, "attack" or "retreat", and all correct generals need to agree on a single decision, "attack" or "retreat", by voting. However, a Byzantine-faulty general can vote different values to other generals which makes correct generals collect inconsistent votes. For example, a faulty general can vote "attack" to one general and vote "retreat" to another general. In order to ensure that correct generals collect identical votes from all the generals, a sub-routine called the Byzantine broadcast is needed to make sure correct generals receive identical vote from a source even if the source is Byzantine-faulty. The problem setting is the following: one source needs to broadcast an L-bit value to n-1 other peers. If the source is fault-free, all correct peers will know the L-bit value which the correct general broadcasts. If the source is Byzantine-faulty, then all the correct peers need to agree on some value.

Lamport et al. then proved that it is impossible to solve Byzantine broadcast problem for a system that greater than or equal to one third of its nodes are compromised. For example, for a system of 1 source and 2 peers, a correct peer will not be able to differentiate whether the source is faulty or the other peer is faulty. Then they also proposed a recursive algorithm to solve this problem when the number of fault-free nodes is more than two thirds of the number of nodes in the system. However, the recursive algorithm needs $O(2^n)$ bits communication cost for agreeing on a single bit.

2.2 Practical Byzantine Fault Tolerance (PBFT)

The major improvement in PBFT [2, 3] in respect to the classical algorithm [1] is that PBFT compares the hash value of the original L-bit value from each peer rather than using the original L-bit value itself. In this way PBFT separates the Byzantine detection from actually agreeing upon the value itself and lower the communication cost to $O(n)$ bits to agree on single bit value in fault-free cases. Another benefit of using PBFT is that PBFT can work with an asynchronous network, where Lamport et al.'s classical solution only works for a synchronous network. However, because PBFT depends on the collision resistant hash functions, PBFT is not an always-correct protocol due to hash collisions. More specifically, PBFT's correctness depends on the probability of hash collisions and whether the hackers can find a hash collision. Better hash functions lead to higher computational cost in PBFT and thus slow PBFT down to some extent. Implementation details will be discussed in Chapter 3.

2.3 Network Coding Based Algorithm (NCBA)

Liang et al. [8] proposed NCBA protocol which uses network coding to protect the L-bit value's content. The source protects the value by Reed-Solomon coding and creates different coded segments of the original L-bit value. The source then sends different segments of the original L-bit value to different peers. The peers then exchange what they get from the source and determine whether the original L-bit value can be recovered or not. This will introduce slightly more communication cost than PBFT but avoids using hash functions. I will discuss my implementation and Benjamin's implementation [8] in Chapter 3.

CHAPTER 3

TESTING FRAMEWORK

3.1 Overview

Figure 3.1 is the software architecture of the testing framework. The testing framework separates the communication layer, group membership from the actual Byzantine broadcast protocols running on the framework. The first motivation is that extending the functionality of the testing system is easier so that future users can focus on writing clean protocol codes on the framework without worrying about the detailed work about communication and multi-threading. The second motivation is that measurements on all the protocols are fair to each protocol because each experiment runs similar sets of codes except for the protocol parts. Because of the above two reasons, the framework has two parts. BFTNode is installed on each machine, regardless of source or peers, to ensure packets send and receive, send retry, task scheduling, task management. BFTProtocol is an abstract class that follows the usual multi-phase BFT broadcast algorithms, where each node starts at some original state, some communications by packets delivering take place, and then each correct peer sends a commit packet to the source. When the source collects enough commits from the peers, the source knows that the protocol ends. Any Byzantine broadcast protocols need to extend this abstract class for the system to run it.

3.2 Terminology

- value
  The original L-bit value is the value that all the correct peers need to

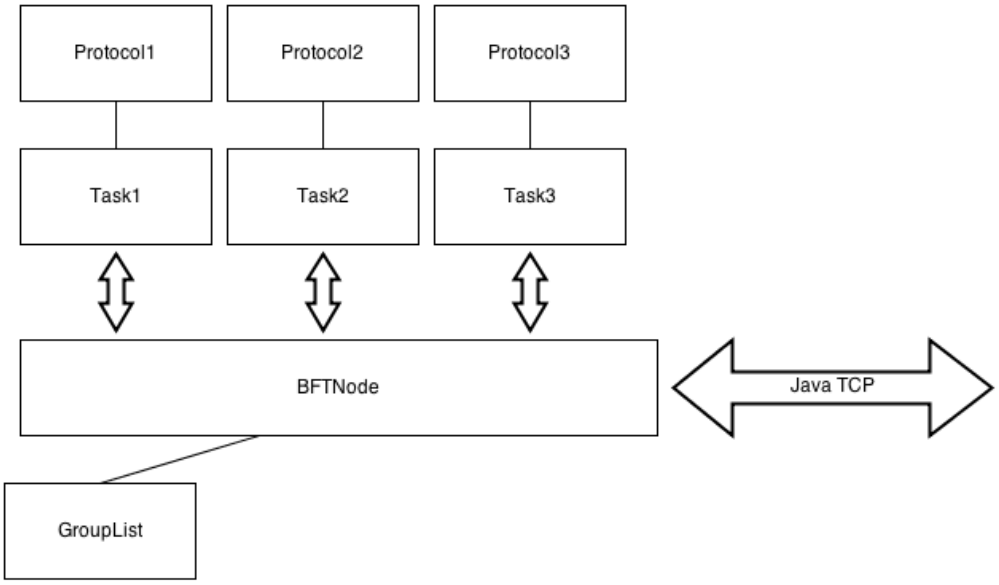

Figure 3.1: System Architecture

agree on after the protocol termination.

- packet
  The communication in the system is through sending and receiving packets.

- value id
  The value id is the id for the L-bit value. It can be understood as the id of each Byzantine broadcast.

- member id
  The member id is distinct for each node in the system.

- task
  A task is a class to handle all incoming packets for a single value id.

3.3 BFTNode

3.3.1 Membership

BFTNode holds a mapping between ids to other machines' IP address and port numbers. So the communication is transparent to the protocols. All the protocols know about the other machines only by their member id number. The source will have the id of 0 and other machines will have ids starting from 1. Each machine will have a distinct id. This mapping is retrieved from a text file called "GroupList" in the file system, which is replicated in all machines.

3.3.2 Scheduling and Synchronization

Every value the source wants the peers to Byzantine agree on has a distinct value id number. Here I assume that each L-bit value is independent of each other and there is no causal relationship between different values, which means value with value id = 2 can be agreed successfully before value with value id = 1 is successfully agreed. This gives the system the ability to agree in parallel on values that are not related to each other which will boost the performance of the Byzantine fault tolerance protocol. A detailed discussion of performance boosting is in Chapter 4.

Whenever a peer receives a new packet with a value id it has not seen before from the source or other peers, the peer generates a receiveTask locally to handle all the following packets for agreeing on the L-bit value corresponding to the unique value id. To ensure a task only process one packet at a time, incoming packets for a single value id need to be serialized. A lock is created for each task to serialize incoming packets for each unique value id. Whenever the node receives a packet from other machines and the node has the task to handle the packet, the node will generate a new thread to try to grab the lock for that task. If the lock is successfully obtained, the node will forward the packet to the task to handle it. If the lock cannot be acquired, the thread will wait until the lock is freed.

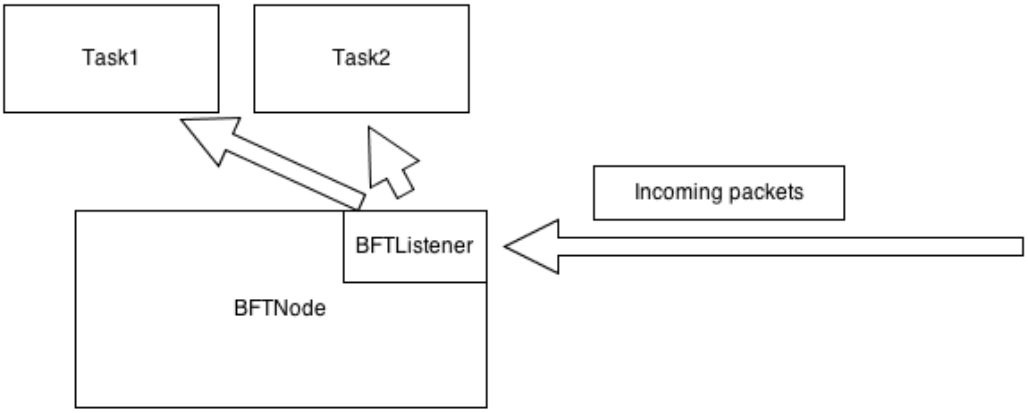

Figure 3.2: Handle incoming packets

In this way, a single task always handle one packet at a time and packets with different value id numbers can still execute simultaneously.

3.3.3 Communication

Figure 3.2 demonstrate how a BFTNode handles incoming packets. The framework uses Java's TCP implementation to communicate. Each node has a BFTListener which is a simple thread waiting for incoming packets. BFTNode sends a packet by the target machine's member id. A packet should always have a value id number indicating which value this packet belongs to. Also a packet has a field indicating the choice of BFTProtocol it wants to use and the member ids for the sender and the receiver. When a BFTNode receives a packet, BFTNode forwards the packet to the task that can handle this value id.

3.3.4 Measurement

Each protocol running on the testing framework needs to specify the termination of the protocol and thus the task will be removed from the node.

When the source starts to Byzantine broadcast a value, he needs to create a new value id number and a sendTask to handle incoming commit packets and change of states for that value id. The sendTask will notify the BFTNode whether they have completed after finishing processing each packet forwarded to the sendTask. If all the sendTasks complete, the timespan for completing all the tasks will be computed and a new set of experiments take place afterwards. Usually, in a measurement we needs a lot of similar experiments to demonstrate protocol's performance.

3.4 BFTProtocol

3.4.1 Overview

I implement the fault-free cases for Lamport et al.'s classical algorithm [1], the PBFT protocol [2, 3] and the NCBA protocol [8]. I build an abstract class to generalize those multi-phase Byzantine agreement protocols. Each protocol needs to extend BFTsendTask class and BFTreceiveTask class to add the BFT functionality. In this section, I will discuss the implementations of those protocols.

I assume the size of each value to be agreed on is L bits. The total number of nodes is n and the maximum number of Byzantine faults is f.

3.4.2 Classical Algorithm

I implement Lamport et al.'s classical algorithm to do the Byzantine broadcast for n=4, f=1 in a synchronous network. It has three phases. In the first phase, the source broadcasts the L-bit value to all the peers. In the second phase when peers receive the value sent by the source, the peers do an all-to-all communication to exchange what they receive from the source. In the third phase, when they receive all the values from other peers, they agree on the value by choosing the value appear the most frequently and send the commit packet to the source. When the source receive f+1 commits from the

peers, the protocol ends.

In the Lamport et al.'s algorithm, the first state broadcast needs $3L$. The value forwarding needs $3 \times 2L = 6L$. So in all, the communication complexity is $3L + 6L = 9L$.

3.4.3 Digest

Digest is a simple version of the PBFT protocol [2, 3] which I implemented according to Liang et al.'s paper [8] for n=4, f=1. I use SHA-1 as the hash function to generate the hash for the value. The protocol has 4 phases. In the first phase, the source broadcasts the value to all peers. In the second phase, when a peer receives the value sent by the source, the peer forwards a prepare packet to other peers. The prepare packet has a random byte array generated by this peer, and a hash value for the original L-bit value and the random byte array combined. In this way, the peer only sends the random byte array and the hash value but not the whole L-bit value to another peer. In the third phase, when a peer receives a forwarded prepare packet from another peer, it needs to verify the hash value is indeed the hash value of the original L-bit value and the random byte array he received from the source. In the commit phase, the peers Byzantine broadcast 1-bit flag indicating whether they detect a Byzantine fault in the system using the Lamport et al.'s classical Byzantine broadcast algorithm. If no node complains, the peer sends the commit packet to the source. When the source receive f+1 commits from the peers, the protocol ends.

In the Digest protocol, the first stage broadcast needs $3L$. That is the dominant communication complexity if we consider the hash value's length is far smaller than L bits.

3.4.4 NCBA

The NCBA protocol [8] is very similar to the Digest protocol discussed above except for the packet sent by the source and the way peers compare each other's forwarded packets. I implement the NCBA protocol for n=4, f=1. Because f = 1, the Reed-Solomon code to ensure detecting Byzantine faults can be simplied to a parity check [8]. Again, the NCBA protocol has 4 phases. In the first phase, the source encodes the original L-bit value to 4 segments, where 3 of the 4 segments are enough to decode the original L-bit value. Here I just separate the original value into 3 segments where each segment has $L/3$ bits and then compute the parity of the 3 segments to generate the 4th segments which is also $L/3$ bits long. It is obvious that any 3 segments in the 4 segments are enough to generate the whole L-bit value by computing the parity. Then the source sends 1 segment of the original value and the parity segment to each of the 3 peers in the system. In the second phase, peers exchange the segments they uniquely received from the source. In the third phase, after the peers receive all the segments from other peers, the peers will be able to tell whether the value is indeed correct by checking if the parity of the L-bit value still holds. The peers then Byzantine broadcast 1-bit flag indicating whether they detect a Byzantine fault in the system using the Lamport et al.'s classical Byzantine broadcast algorithm. If no node complains, the peer sends the commit packet to the source. When the source receive f+1 commits from the peers, the protocol ends.

In the NCBA protocol, the first stage broadcast needs $3 \times 2/3L = 2L$ because every peer needs two thirds of the value. In the second stage, where all-to-all communication happens, it needs $3 \times 2 \times 1/3L = 2L$ because every peer only forwards one third of the value to other two peers. So the overall communication complexity is $2L + 2L = 4L$.

3.5 Comparison with Previous Implementations

The major difference between the implementation in [8] and my work is that my testing framework supports multiple Byzantine broadcasts in parallel. The motivation behind this modification is that, for example, usually in a

distributed file system, files are not causally related to each other and there is no need to wait for the previous broadcast to terminate before starting the next broadcast. In my design, each value id is the identifier for the value and packets with different value ids can proceed in parallel without blocking each other.

CHAPTER 4

EVALUATION RESULTS

4.1 Environment

All the codes are first compiled into Java virtual machine binary code which is highly optimized. All the nodes start their Java virtual machines to run the framework and there is no disk I/O involved in the system. All the results are averages of 100 test runs.

In the experiment I conduct, I use 1 source and 3 peers. The source needs to Byzantine broadcast to all the peers so that each correct source and peer will have the exact the same value. The source is a Sumsung laptop running Windows 8 on a 1.7 GHz Intel Core i5 and 8 GiB of RAM. The 3 peers are Dell Inspiron 1545 laptops running Ubuntu 9.04 Jaunty, Gnome 2.26.1, and Linux Kernel 2.6.28-11-generic on 2.9 GiB of RAM and a 2.0 GHz Intel Core 2 Duo Processor T6400. The four machines are connected by a Netgear GS108 gigabit switch.

4.2 Data

Peers agree on a single L-bit value sequentially, where L = 300 Bytes, 3 KB, 30 KB and 300 KB. The time span for agreeing on each value is measured is in milliseconds.

L	300 Bytes	3 KB	30KB	300KB
Classic	11.09	12.17	19.69	157.69
Digest	17.81	19.63	21.22	106.39
NCBA	17.03	16.72	19.53	85.32

Peers agree on two L-bit values in parallel, where L = 300 Bytes, 3 KB, 30 KB and 300 KB. The time span for agreeing on two values is measured is in milliseconds.

L	300 Bytes	3 KB	30KB	300KB
Classic	14.07	15.47	32.34	317.23
Digest	28.36	25.86	31.87	182.52
NCBA	25.96	25.01	28.51	153.86

Savings by agreeing on 2 values in parallel can be computed by

$$1 - \frac{\text{the latency of agreeing on two L-bit values in parallel}}{2 \times \text{the latency of agreeing on a single L-bit value}}. \tag{4.1}$$

For example, for L = 30 KB and using NCBA, the savings of Byzantine agreeing on two values in parallel compared to agreeing on one value sequentially is

$$1 - \frac{28.51}{2 \times 19.53} = 27\%. \tag{4.2}$$

L	300 Bytes	3 KB	30KB	300KB
Classic	36.6%	36.5%	17.9%	-0.59%
Digest	20.4%	34.2%	24.9%	14.2%
NCBA	23.8%	25.3%	27.0%	9.8%

4.3 Analysis

I measured the delay from when the source issues the Byzantine broadcast to the time when the source receives enough commits packets from the peers. The delay roughly depends on three different aspects of the protocol:

- The number of phases and average RTT (round trip time)
- Communication complexity
- Computation complexity

When the value size is small (less than or equal to 3KB), the dominant factor in the delay is the number of phases of the algorithm. The classical

algorithm outperforms both Digest and NCBA because it is only a 3-phase protocol when f = 1 whereas both Digest and NCBA are 4-phase protocols in fault-free cases. Agreeing on multiple values in parallel is very useful (around 30% more efficient) because most of the delay is caused by propagation delay or in other words RTT. And parallelism boosts the performance of Byzantine broadcast by using the idle time span when the machine is waiting for the incoming packets to execute the Byzantine protocol for values of other value ids.

When the value size is relatively large (greater than or equal to 300KB), the dominant factors are communication complexity and computation complexity. The classic algorithm's performance is thus far behind both the Digest protocol and the NCBA protocol. In my experience, NCBA is faster than Digest. This is due to Reed-Solomon code can be quickly computed for f =1 but one-way hash functions such as SHA-1 is still computational intensive. Agreeing on multiple values in parallel is thus not very useful as the value size grows larger because most of the delay is due to heavy computation or network complexities. If the machine's CPU is heavily utilized or the network bandwidth is heavily utilized, agreeing in parallel does not help compared to agreeing on values sequentially.

CHAPTER 5

CONCLUSION

As can be seen from the results, NCBA's performance is comparable to Digest and sometimes even a little better than Digest. By not using the collision resistant hash function, NCBA saves a lot of computation cost on the servers but introduces a slightly larger communication cost. Also, NCBA is an always-correct protocol and does not use any cryptographic assumptions on the Byzantine-faulty nodes, whereas PBFT relies on collision resistant hash function to work correctly.

CHAPTER 6

FUTURE WORK

We can see two trends of distributed system researches in this area. Crash fault tolerance (CFT) systems are already been built with great performance, such as BigTable [10], Cassandra [11], Dynamo [12]. The other trend is that Byzantine fault tolerance (BFT) protocols becomes quick [3, 5, 8], fault-scalable [4, 6], costless [9, 13] and robust [7].

It is interesting to ask whether it is possible to build Byzantine fault tolerance as an add-on feature on current CFT systems and at the same time minimize the additional cost for systems to switch from CFT to BFT. Upright [14] is designed to replicate a service to make it support Byzantine fault tolerance.

In replicated systems, to ensure linearizability [15], often the system needs each replica to process all the incoming requests in the same order. This may require the system to use Byzantine fault tolerance protocol on the critical path to make sure all the correct replicas process incoming requests in exactly the same order. Unfortunately, this will incur an extra overhead because BFT protocol's overhead is larger than CFT's.

To lower the overhead of replicated systems in Byzantine fault model, we may want to let replicas temporarily process incoming requests with inconsistent ordering, and try to fix the ordering later. This can be done in read/write data objects [16, 17] by keeping enough versions of the data. We have some preliminary thoughts on how to solve this problem in general cases.

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, July 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[2] M. Oom Temudo De Castro, "Practical byzantine fault tolerance," Ph.D. dissertation, 2000, aAI0803037.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095817 pp. 59–74.

[5] 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 implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298473 pp. 177–190.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294267 pp. 45–58.

[7] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558988 pp. 153–168.

[8] G. Liang, B. Sommer, and N. Vaidya, "Experimental performance comparison of byzantine fault-tolerant protocols for data centers," in *INFOCOM, 2012 Proceedings IEEE*, 2012, pp. 1422–1430.

[9] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "Zz and the art of practical bft execution," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966457 pp. 123–138.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[11] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281 pp. 205–220.

[13] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945470 pp. 253–267.

[14] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629602 pp. 277–290.

[15] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, July 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[16] M. Abd-El-Malek, G. R. Ganger, M. K. Reiter, J. J. Wylie, and G. R. Goodson, "Lazy verification in fault-tolerant distributed storage systems," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/RELDIS.2005.20 pp. 179–190.

[17] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient byzantine-tolerant erasure-coded storage," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1009382.1009729 pp. 135–.