An Optimized Multi-Paxos Consensus Protocol for Practical Cloud Storage Applications

Wenmin Lin[1,2,3](✉), Xuan Sheng[1,2], and Lianyong Qi[4]

[1] Department of Computer Science, Hangzhou Dianzi University, Hangzhou, China
linwenmin@hdu.edu.cn, upxuans@163.com

[2] State Key Laboratory of Complex System Modeling and Simulation, Hangzhou, China

[3] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

[4] School of Information Science and Engineering, Qufu Normal University, Jining, China
lianyongqi@gmail.com

Abstract. For cloud storage applications running typical Multi-Paxos protocol, the processing of a client command normally consists of two steps, i.e., commit and apply. Commit step is to guarantee a client command achieves identical sequence number among all storage replicas; apply step is to execute a committed client commands one by one in sequence and return back the execution result to client. In practice, committed client commands are not necessarily be applied after all its previous commands get applied. In view of this observation, an optimization for Multi-Paxos protocol is proposed to improve system performance for cloud storage applications in this paper. Compared with typical Multi-Paxos protocol, we allow out-of-order applying of committed client commands. And a committed client command can be applied as long as it has no dependency on its previous commands or all dependencies are resolved. Comparison between two protocols is implemented and analyzed to prove the feasibility of our proposal.

Keywords: Optimized Multi-Paxos · Consensus protocol · Out-of-order apply

1 Introduction

Nowadays, increasing amount of applications are deployed in cloud, due to the convenience of "pay as you go" manner of using IT infrastructure. Among those applications, cloud storage application is one of the most popular one. Cloud storage applications enable users to store data of their applications on cloud, instead of building their own storage infrastructures [1, 2]. As a typical distributed computing application, cloud storage systems take advantage of replica technique to achieve fault tolerance and high availability, by storing user's data on multiple disks over the network, so as to make sure the data won't be lost as long as majority disks working probably [3].

As a distributed computing application, a cloud storage system can be treated as a set of distributed servers belonging to one cluster. The servers work as a whole to

https://doi.org/10.1007/978-3-030-37337-5_47

process client commands (i.e., write or read operations to store data and read stored data) [4]. Each sever can be described as a deterministic state machine that performs client commands in sequence. The state machine has a current state, and it performs a step by taking as input a client command and producing an output and a new state. The core implementation of a cloud storage system is the consensus module to guarantee all servers execute the same sequence of client commands [5]. As a result, every cloud storage server can be modeled as a replicated state machine as shown in Fig. 1.

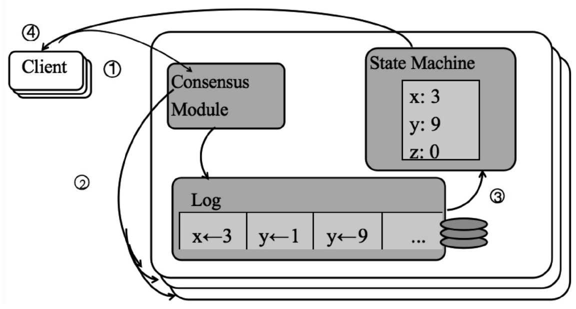

Fig. 1. Replicated state machine architecture [5].

Replicated state machines are typically implemented using a replicated log [4, 5]. Each server stores a log containing a series of client commands, from which its state machine executes in sequence. Each log contains the same commands in the same sequence, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and produces the same sequence of outputs. Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same sequence, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

There have been numerous researches on the consensus algorithm for replicated state machines. Among which Paxos is the dominated one over last decades: most implementations of consensus are based on Paxos or influenced by it. Representative algorithms include Multi-Paxos [4], E-Paxos [6], as well as Raft [5]. The difference between Paxos and its variants Raft is: Raft is strongly based on leadership mechanism, all client commands are handled by leader replica and other replicas work as followers; while for Multi-Paxos, leader is not necessarily required, but it always employs a distinguished leader to guarantee liveness of the algorithm; Moreover, E-paxos is totally leaderless to guarantee client latency for handling client commands in wide area environment. In this paper, we mainly focus on the optimization of Multi-Paxos regarding its performance optimization.

For cloud storage applications running typical Multi-Paxos protocol, the processing of a client command normally consists of two steps, commit and apply. Commit step is to guarantee a client command achieves identical sequence number among all storage replicas; apply step is to execute committed client commands one by one and return back the execution result to client. In practice, client commands could be distributed to all replica servers concurrently. Therefore, commands could be committed in an out-of-order manner due to factors such as network delay. In original design, commands should be applied strictly in sequence after they are committed [1, 7, 8]. That means, any committed command C_i and C_j, ($Seq(C_i) < Seq(C_j)$), C_j must be applied after C_i is applied even C_j is committed before C_i. In practice, this is not necessary if C_j has no dependency on C_i. By allowing out-of-order apply, we can reduce client latency by improving the system's I/O throughput of each storage replica node.

In view of this observation, we propose an optimized Multi-Paxos protocol in this paper, where client commands could be applied in out-of-order manner after they get committed. The reminder of this paper is organized as follows: Sect. 2 discusses related work on consensus algorithm for cloud storage applications. Section 3 highlights the problem of typical Multi-Paxos protocol. The details of the optimization of Multi-Paxos protocol is presented in Sect. 4. Section 5 evaluates the performance of optimized Multi-Paxos protocol and typical Multi-Paxos protocol in terms of commit throughput. And Sect. 6 concludes the paper.

2 Related Work

As a distributed computing system, the core component of cloud storage applications is the consensus module, which is to guarantee each replica server executes client commands in the same sequence [10–12]. There have been numerous researches on consensus algorithms of distributed systems over last decades [13, 14], from which Paxos is the dominated one. Most implementations of consensus are based on Paxos or influenced by it. Among those consensus algorithms, they can be categorized as follows: (1) Lamport's Paxos [4, 8, 9], and its variants such as Multi-Paxos, Elaborations Paxos (E-Paxos) [6]; (2) Raft protocol [5], which is based on strong leadership mechanism.

Paxos protocol is a two-phase protocol, which contains prepare phase and accept phase [15, 16]. For a given command C_i, prepare phase is to make sure majority replicas agree to append C_i as the i-th command in its local log; and accept phase is to double confirm C_i has been appended as the i-th command in majority replica servers. Since majority replicas (more than half members) reach consistency regarding the sequence of C_i, all replicas will finally learn such information according to pigeonhole principle. As a result, Paxos protocol could guarantee that each replica sever will reach consistency regarding the sequence of each client command. A single Paxos instance is to determine the sequence of a single client command. Multi-Paxos is a variant of Paxos protocol, which enables handling multiple client commands concurrently with multiple Paxos instances. Moreover, in Multi-Paxos protocol, a leader could be elected to nominate sequence for each client command. As a result, the prepare phase could be omitted to improve system latency, so as to guarantee the liveness of the protocol.

E-Paxos is also a variant of Paxos protocol, which is to optimize system performance of client latency in wide-area applications, especially when there are nodes fail during consensus process. RAFT is a strong leader-based consensus protocol, where all determinations regarding the sequence of each command is made by the leader, and all other replica servers works as follower of the leader replica.

The main difference between those consensus algorithms is the leadership mechanism: (1) Multi-Paxos does not necessarily requires a leader; and when there's no leader in a replica group, Multi-Paxos degrades to the basic two-phase protocol. (2) E-Paxos is totally leaderless, which is designed to reduce client latency in wide-area scenario. (3) Moreover, Raft uses a strong leadership mechanism compared with other two consensus algorithms. Without a leader, the system running Raft protocol will become unavailable.

In this paper, we focus on optimization for Multi-Paxos protocol. As mentioned in aforementioned section, a cloud storage application running Multi-Paxos protocol normally consists of commit phase and apply phase. Commit phase is to guarantee a client command achieves identical sequence number among all storage replicas; apply phase is to execute a committed client commands one by one and return back the execution result to client. In practice, committed client commands are not necessarily be applied after all its previous commands get applied. In view of this observation, an optimization for Multi-Paxos protocol to improve system performance of cloud storage applications is proposed in this paper. By enabling out-of-order apply client commands, we can improve system's throughput, so as to reduce client latency to read committed commands.

3 Preliminary Knowledge

3.1 How Multi-Paxos Protocol Works

A typical Multi-Paxos protocol is similar to two-phase commit protocol (i.e., the two phases are prepare phase and accept phase, respectively). When a replica R_i within a replica group receives a client command C_k, the two-phase Multi-Paxos protocol works as follows:

Prepare Phase: R_i first record C_k as the k-th client command in its local log, then broadcast *prepare_*C_k requests with proposal number R_i-k within the replica group. On receiving the *prepare_*C_k request for each replica R_j, it will send back *prepare_*C_k*_OK* response to R_i after checking it's ok to log C_k as the k-th log entry locally. If R_i receives *prepare_*C_k*_OK* response from majority replicas, it will enter Accept phase to make C_k as the k-th log entry in majority replicas.

Accept Phase: R_i initiates *Accept_*C_k requests and broadcast it within the replica group. On receiving *Accept_*C_k requests for each replica R_j, it will record C_k as the k-th log entry, and send back *Accept_*C_k*_OK* response after it checks there's no proposal number larger than R_i-k for the k-th log entry. Similarly, when R_i receives *Accept_*C_k*_OK* responses from majority replicas, it will mark C_k as committed; and broadcast *Commit_C*$_k$ requests. Once R_j receives *Commit_C*$_k$ request, it will mark C_k as committed if it has recorded C_k as the k-th log entry as well.

After a command get committed, it can be applied to state machine as long as it has no dependency on other commands, or all its dependency are resolved probably. Therefore, a response will be send back to client to indicate the success of executing C_k by the cloud storage application.

An issue with Multi-Paxos protocol is when there are multiple replicas raising prepare requests simultaneously, it is with great possibility that none replica's proposal will be accepted. For example, when R_i raises (*Prepare_C*$_k$, R_i-k), there is another replica R_j raises (*Prepare_C*$_k$', R_j-k') with R_j-k' > R_i-k simultaneously. R_j will win majority votes in prepare phase since R_j-k' is the largest proposal number. Before R_j's accept requests reaches majority replicas, R_i will initiate a new prepare request with a new proposal number R_i-m > R_j-k' to win majority votes in prepare phase. As a result, R_j's accept request will be ignored by majority replicas since R_i-m is the largest proposal. This scenario is called "mutual-tread" and will cause live lock of the protocol [4]. For the cloud storage applications, the live lock issue means for a same log entry (e.g., the k-th log entry), more than one replica issues *Prepare_C*$_k$ requests within the replica group and none replica wins the right to write to the k-th log entry. To address this problem, a distinguished replica could be elected as leader to determine the sequence for each client command. Therefore, the two-phase protocol is reduced to one-phase protocol by omitting the prepare phase, so as to improve system's performance and avoid the liveness issue. The difference between the two-phase Multi-paxos protocol and the one-phase Multi-paxos protocol is depicted in Fig. 2.

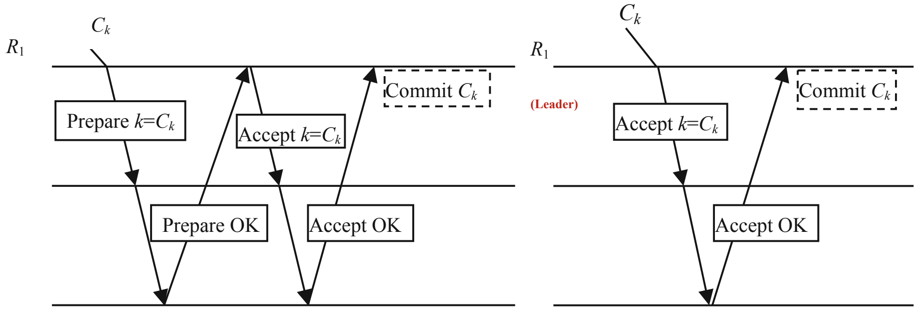

Fig. 2. Two-phase Multi-Paxos protocol vs. One-phase Multi-Paxos protocol

Let's take the scenario in Fig. 3 to describe how Multi-Paxos protocol works. A client sends three commands C_1, C_2 and C_3 at the same time to the leader replica R_1, i.e., {C_1: "$x = v_1$", C_2: "$y = v_2$", C_3 = "$x* = v_3$"}. C_1 and C_3 are updating the same key x; while C_2 is updating key y. Then R_1 will log C_1, C_2, and C_3 in sequence at its local log firstly, then broadcast *Accept_C*$_k$ messages regarding each command to each follower in the replica group. On receiving the *Accept_C*$_1$ request from R_1, R_2 and R_3 will record the C_1 in its local log; then send back *Accept_C*$_1$*_OK* message to R_1. Once R_1 receives Accept OK messages from at least follower replica, it will mark C_1 as committed, and broadcast *Commit_*C_1 request to all followers. And on receiving a *Commit_*C_1 message, a follower replica will mark C_1 as committed if it has already record C_1 in its local log. Once C_1 is committed, it can be applied to the state machine and sends back to client that C_1 has already been recorded correctly. For C_2 and C_3, the workflow is similar to C_1.

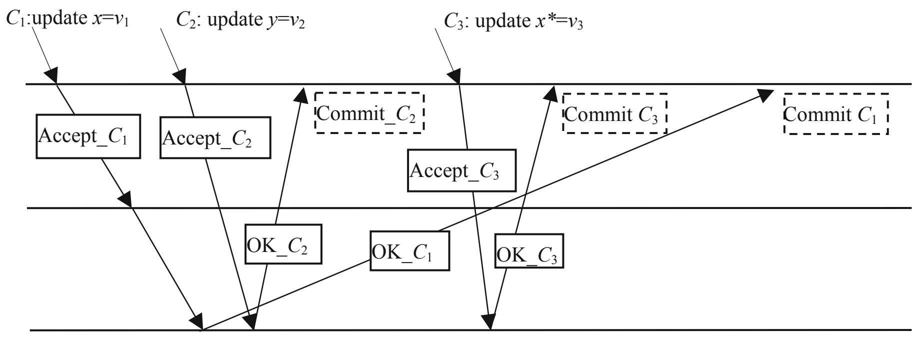

Fig. 3. A Multi-Paxos workflow example

3.2 Problem Statement

In the example depicted in Fig. 3, C_1, C_2 and C_3 may get committed in out-of-order manner, since the network delay may results the "*Accept_*C_2*_OK*" and "*Accept_*C_3*_OK*" messages arrives R_1 before "*Accept_*C_1*_OK*" message. But they must be applied in sequence due to protocol's design, since there may be some dependency among those 3 commands. For example, in our case, C_3 can only be applied after C_1 has been applied. However, it is not necessary for C_2 to get applied after C_1 has been applied in our case, since C_2 has no dependency on C_1. As a result, we can do some optimization to allow out-of-order apply of committed commands in Multi-Paxos protocol.

In view of those observations, we propose an optimization for Multi-Paxos protocol by allowing out-of-order applying of committed commands. By doing this, we could improve system throughput, and reduce client latency on reading the committed commands as well.

4 Our Solution: An Optimized Multi-paxos Protocol

Motivated by the problem discussed in Sect. 3, the optimized Multi-Paxos protocol is discussed in details in this section. In our proposal, we introduce a concept named dependency window for each client command. And a committed command could be applied as long as it's dependency window is empty or all the dependent commands are getting applied.

Definition 1 (dependency window). For a command C_k, its dependency window Dep_k is a data structure recording the m former keys close to command C_k in the whole log entry, i.e., $Dep_k = \{x_{k-m}, x_{k-(m-1)}, \ldots x_{k-1}\}$. For each x_i in Dep_k, if C_k depends on its corresponding command C_i, C_k can only be applied after C_i get applied.

Compared with original design of Multi-paxos protocol, dependency window is added for each client command, to help a replica judge whether a client command can be applied or not. When a client command C_k arrives the leader replica R_i, an Accept message for C_k will be broadcast to all followers, with the dependency window information (i.e., Dep_k) attached. With Dep_k, a follower replica R_j can extract the dependent keys on which C_k relies on. And R_j will intuitively check whether C_k can be applied once it get committed.

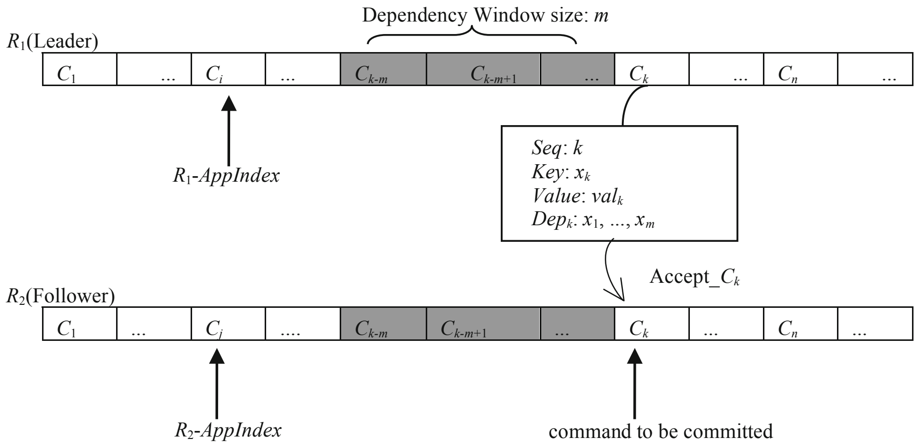
Dependency Window size: m
R_1(Leader)
C_1 ... C_i ... C_{k-m} C_{k-m+1} ... C_k ... C_n ...
R_1-*AppIndex*
Seq: k
Key: x_k
Value: val_k
Dep_k: $x_1, \ldots, x_m$
Accept_C_k
R_2(Follower)
C_1 ... C_j .... C_{k-m} C_{k-m+1} ... C_k ... C_n ...
R_2-*AppIndex*
command to be committed

Fig. 4. The workflow of optimized Multi-Paxos protocol to commit a command

Step 1: Commit client commands

Figure 4 demonstrates the workflow to commit a client command C_k in our proposed optimized Multi-Paxos protocol. For a given client command C_k sent to Leader R_i, the commit protocol works as follows: (1) R_i records C_k in its local log as the k-th command in sequence; (2) R_i broadcasts "*Accept_C_k*" message to all follower replicas in the cluster; (3) On receiving a "*Accept_C_k*" message, a follower replica will first log C_k locally and sends back R_i the "*Accept_C_k_OK*" message; (4) Once R_i receives "*Accept_C_k_OK*" message from the majority in the cluster, it will identify C_k as committed; then broadcasts "*Commit_C_k_OK*" message to all followers in the cluster; (5) On receiving a "*Commit_C_k_OK*" message, a follower replica will mark C_k as committed if it has C_k in local log; otherwise it will mark C_k as committed after "*Accept_C_k_OK*" message arrives.

Step 2: Apply committed commands

Definition 7 (Apply index R_i-*AppIndex*). R_i-*AppIndex* is the index of a replica R_i evolves in a Multi-Paxos protocol. It indicates that commands C_1, C_2, … $C_{Ri\text{-}AppIndex}$ are already applied from local log to the state machine.

Concretely, for a command C_k to be applied, 2 conditions should be satisfied. (1) C_k must be committed: for leader, it means it receives "*Accept_*C_k*_OK*" message from the majority of peers in the system. For a follower replica, it must receive both "*Accept_*C_k" and "*Commit_*C_k*_OK*" from leader; (2) C_k's key has no dependency on the command keys between R_i-*AppIndex* and the command C_k itself.

Taking Fig. 3 for example, the optimized multi-paxos protocol works as follows:

(a) Commit Step

(1) On receiving client request C_1, C_2, C_3, R_1 records C_1, C_2, C_3 in sequence to local log;
(2) R_1 broadcast *Accept_*C_1 = {1, x, 3, θ}, *Accept_*C_2 = {2, y, 4, θ} and *Accept_*C_3 = {3, x, 5, {x}} messages to R_2 and R_3;
(3) Since *Accept_*C_2 and *Accept_*C_3 arrives R_3 before *Accept_*C_1 message, *Accept_*C_2*_OK* and *Accept_*C_3*_OK* message sends back to R_1, C_2 and C_3 are logged in R_3 in sequence as well;
(4) C_2 and C_3 are committed by R_1 on receiving R_3's response at t_1;
(5) R_1 sends "*Commit_*C_2*_OK*" and "*Commit_*C_3*_OK*" message to R_2 and R_3;
(6) R_3 mark C_2 and C_3 as committed, since they have already been recorded in local log;
(7) Similarly, C_1 will be committed by R_1 after receiving R_2's response time at $t_2(t_2 > t_1)$ and then committed by R_2 as well.

(b) Apply Step

(1) for R_1, since C_2 and C_3 are committed at t_1, apply process is triggered;
(2) for C_2, since its updated *key* = y, has no conflict with any command between R_1-*AppIndex* = 0 and C_1, it get applied immediately;
(3) while for C_3, since its updated *key* = x, is interfering with C_1 which also updates key x, so it won't be applied immediately at t_1. And it will only be applied after C_1 get applied;
(4) After t_2, C_1 get committed, and R_1-*AppIndex* is updated to 2 as well;
(5) C_3 get applied since it has no dependency on commands between C_2 and C_3.

5 Evaluation

We evaluated the optimized Multi-Paxos protocol against typical Multi-Paxos protocol, using three replicas for each replicated state machine. The protocols are implemented with Golang and running on Mac OS 10.13.16.

According to [3], in practice, dependency among commands is rare cases for cloud storage applications with statistics around 1%. We did this comparison just to highlight that dependency among commands does have impact on distributed system's performance. In our scenario, we assume there is 20% probability a command depends on another command.

The size of client commands sent by each client is 100, 000; and for each command, the key is a random integer with 64bits; while the value is a random string with fixed size (i.e., 4 KB). The size of dependency window for each Accept message is set to 4. We compare the difference of throughput between the two protocols.

As shown in Fig. 5, with the increase of number of clients, the throughput is increasing linearly. And the optimized Multi-Paxos protocol increases faster than the typical one. This is because it's unnecessary for each replica to wait until all previous commands get applied before applying a committed command. Moreover, from Fig. 5, we can find that when the number of clients reaches 6, the throughput does not change anymore in optimized Multi-Paxos protocol. The reason for this is when client number reaches 5, the storage I/O is fully utilized to process client commands.

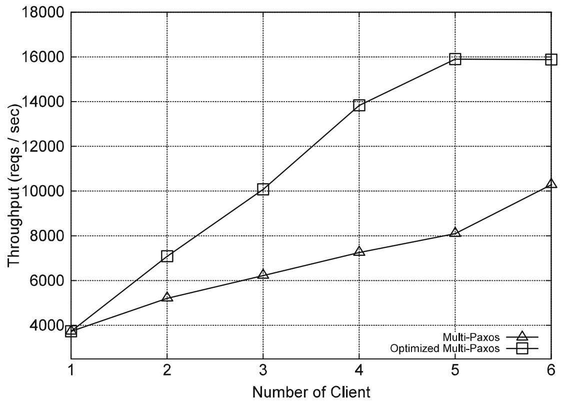

Fig. 5. Throughput performance comparison with no dependency among commands

6 Conclusion

In this paper, we proposed an optimization for Multi-Paxos protocol by allowing out-of-order applying client commands. Compared with original design of typical Multi-Paxos protocol, a committed client command could be applied to state machine, as long as it has no dependency on its previous commands or all dependencies are resolved. By doing this, we could improve the system throughput of cloud storage applications. Finally, comparison between two protocols is analyzed to prove the feasibility of our proposal.

Acknowledgments. This paper is supported by The National Key Research and Development Program of China (No. 2017YFB1400601), National Natural Science Foundation of China (No. 61872119), Natural Science Foundation of Zhejiang Province (No. LY12F02003).

References

1. Wenying, Z., et al.: Research on cloud storage architecture and key technologies. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, pp. 1044–1048. ACM, Korea (2009)
2. Arokia, R., Shanmugapriyaa, S.: Evolution of cloud storage as cloud computing infrastructure service. IOSR J. Comput. Eng. **1**(1), 38–45 (2012)
3. Ousterhout, J., Agrawal, P., Erickson, D., et al.: The case for RAM Cloud. Commun. ACM **54**, 121–130 (2011)
4. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)
5. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of ATC 2014, Usenix Annual Technical Conference, pp. 1–18 (2014)
6. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. In: SOSP, pp. 358–372 (2013)
7. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
8. David, M.: Paxos Made Simple. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf
9. Tushar, C., Robert, G., Joshua, R.: Paxos made live - an engineering perspective. In: ACM PODC, pp. 1–16 (2007)
10. Lamport, L.: Fast Paxos. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf
11. Jun, R., Eugene, J.S., Sandeep, T.: Using Paxos to build a scalable, consistent, and highly available datastore. Proc. VLDB Endow. **4**(4), 243–254 (2011)
12. Ailidani, A., Aleksey, C., Murat, D.: Consensus in the cloud: Paxos systems demystified. In: 2016 25th International Conference on Computer Communication and Networks, pp. 1–10 (2016)
13. Parisa, J.M., et al.: The performance of Paxos in the cloud. In: Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, pp. 41–50 (2014)
14. Jonathan, K., Yair, A.: Paxos for system builders: an overview. In: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, pp. 1–5 (2008)
15. Wang, C., Jiang, J., Chen, X., Yi, N., Cui, H.: Apus: fast and scalable Paxos on RDMA. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 94–107 (2017)
16. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1), 63–73 (2010)
17. GoLang. https://github.com/golang/go