Migratable Paxos

Low Latency and High Throughput Consensus Under Geographically Shifting Workloads

Yanzhao Wang, Huiqi Hu(✉), Weining Qian, and Aoying Zhou

School of Data Science and Engineering,
East China Normal University, Shanghai, China
51175100009@stu.ecnu.edu.cn, {hqhu,wnqian}@dase.ecnu.edu.cn,
ayzhou@sei.ecnu.edu.cn

Abstract. Global web services or storage systems have to respond to changes in clients' access characteristics for lower latency and higher throughput. As access locality is very common, deploying more servers in datacenters close to clients or moving related data between datacenters is the common practice to respond to those changes. Now Paxos-based protocols are widely used to tolerate failures, but the Reconfiguration process (i.e., the process to changing working servers) of existing Paxos-based protocols is slow and costly, thus has a great impact on performance. In this paper, we propose *Migratable Paxos* (MPaxos), a new Paxos-based consensus protocol that minimizes the duration of reconfiguration and accelerates the migration of data between datacenters, without losing low commit latency and high throughput.

Keywords: Global web services · Paxos consensus protocol · Shifting workloads · Migration

1 Introduction

The access hot spots in global web services usually come from the same geographical region (e.g., a country, a state, or a time zone). And such hot spots may move frequently, as a function of local time. For example, a global electronic trading system will witness the moving of hot spots between countries because of different stock opening times. If such web services use servers in fixed locations, the latency will increase significantly as the hot spots migrate to a location far away from the working servers. Different applications have different tolerances for delays, but high latency generally leads to a decline in revenue.

Now Paxos-based protocols [1,2] are widely used in web services [3,9] and database systems [10] to tolerate server failures and provide high availability. But existing Paxos protocols are not adaptive for the frequent changing workloads, two problems lie ahead of these protocols:

1. Agreements are reached by a majority. Such that the total number of nodes (denote N) is related to tolerated failures (denote F), which satisfies $N = 2F + 1$. Then larger N results in a larger F. This property limits the number of nodes that can be deployed in a Paxos instance and makes it impossible to place a server near every possible client.

© Springer Nature Switzerland AG 2020
Y. Nah et al. (Eds.): DASFAA 2020, LNCS 12112, pp. 296–304, 2020.
https://doi.org/10.1007/978-3-030-59410-7_20

2. In data migration, usually only a specific part of data needs to be migrated. But existing reconfiguration mechanisms transferring the entire state machine, which adds extra cost.

In this paper, we proposed MPaxos to solve these problems. We proposed the *Working Cluster allocation* mechanism. A Working Cluster is the set of nodes explicitly specified for one object, a command can and only can be committed in a majority of the related Working Cluster. In this way, we decoupled quorum size $(F+1)$ from the number of nodes in cluster (N) by introducing W (the number of nodes in a Working Cluster), where $W <= N$ and $W = 2F+1$, thus solved problem 1. Also, the Working Clusters of different objects work independently, so the migration of one object will not affect the read and write of other objects, thus alleviate problem 2. Moreover, MPaxos provides a scheduling framework for automating Working Cluster selection and migration to maintain low latency.

We briefly introduce the relevant background of Paxos and other related works in Sect. 2. Then the design and implementation of MPaxos are present in Sect. 3. The scheduling framework is introduced in Sect. 4. Section 5 shows the performance evaluation. The paper concludes in Sect. 6 with a summary.

2 Related Work

Leader-based Paxos protocols use a master to determine the execution order of all the command, while the Leaderless Paxos protocols usually don't care the order of irrelevant commands and only establish order constraints between commands for the same object. *Egalitarian Paxos* [4] is an efficient leaderless Paxos protocol and it uses a totally decentralized approach to commit commands and handle interferes: in the process of choosing a command in a log entry, each participant attaches ordering constraints to that command, and the agreement is achieved when a majority agree with that constraints, thus irrelevant commands can be committed by different replicas. Therefore, EPaxos is born with high throughput.

The commit protocol of EPaxos is divided into 3 phases. When there are no interferes between commands from different command leaders, these commands can be committed on the Fast-Path (involves only Phase 1 and 3); otherwise interfered commands will be committed on the Slow-Path (involves Phase 1, 2, and 3). On the Slow-Path, each command proposed by command leader requires replies from a majority of replicas, while in the fast path, $F + \lfloor \frac{F+1}{2} \rfloor$ replies is needed to guarantee the correctness, where $F = \lfloor \frac{N}{2} \rfloor$ and N is the number of replicas.

There are also numerous works proposed for the migration of services and workload burst handling. Some early works use the idea of VM live migration, in which the VMs containing the services are migrated between datacenters. This method has been widely used in practice [7,8]. These works adopt the shared disk technology for faster migration but face long latency in accessing a shared disk image. Then another VM based protocol Supercloud [6] proposed a Data Propagation method that implements the storage layer much like a state machine, modified blocks are transferred between datacenters to maintain a consistent storage view.

3 The Design of MPaxos

MPaxos is proposed to achieve lower latency and higher throughput than existing Paxos based protocols in wide-area deployment. High throughput is reached by implementing a decentralized command committing method similar to EPaxos, as the read and write requests can be distributed across different nodes. But a Fast-Path in EPaxos consisting of roughly $\frac{3}{4}$ of nodes makes the commit latency even higher. So MPaxos made the following changes to EPaxos: 1. Introduce the concept of *Working Cluster*. 2. Introduce the concept of *Reorganization* that changes the Working Cluster. 3. Let each object has its own Working Cluster.

Below we describe the components in MPaxos in more detail.

3.1 Working Cluster

The ***Working Cluster*** is a subset of the overall cluster. A replica inside the Working Cluster is called ***Working Replica***. The committing of the commands need to be performed by a Working Replica. The replica outside of the Working Cluster on receiving a command from the client should redirect it to a Working Replica.

Through the Working Cluster mechanism, we can reduce the Fast-quorum size to $F_W + \lfloor \frac{F_W+1}{2} \rfloor$, slow-quorum size to $F_W + 1$ (where $F_W = \lfloor W/2 \rfloor$, so F_W is the actual number of tolerated failures under Working Cluster with size W). Thus we can deploy more idle machines in the cluster, while keeping a small quorum size.

3.2 Reconfiguration and Reorganization Algorithm

The process of changing the Working Cluster (i.e. ***Reorganization***) is essentially a reconfigure process for the Working Cluster, except that the migration does not involve the startup and shut down of replicas. The process of reconfiguration can be divided into three steps: 1. Stop the old state machine, 2. Transfer the state, 3. Start the new state machine. Below we present a detailed description of steps 1 and 2. As the third step is simple and trivial, we don't discuss it here.

The general way to stop the old state machine is to submit a stop command (it is usually done by committing a RECONFIG command), and there can be no other valid commands in old state machines after the stop command (only NOP commands are permitted) [5]. Due to the multileader style of MPaxos, it is possible to have multiple RECONFIG commands committed at the same time, and their contents may be different. One solution is to modify the commit protocol to refusing the old RECONFIG and using another round of communication to confirm this RECONFIG. But this could cause a livelock (different replicas alternately send new RECONFIG and no RECONFIG command can be confirmed successfully). Thus we chose another way: allow multiple potentially different reconfig commands to be committed, but only the earlier one will take effect. To do this, two settings need to be introduced:

Definition 1. *RECONFIG commands conflict with each other.*

Definition 2. *RECONFIG commands conflict with read/write commands.*

With Definition 1, the concurrent RECONFIG commands will establish an execution order. Definition 2 establishes an execution order between the read/write commands and the RECONFIG commands. The read/write commands have to be set to NOP when there is a RECONFIG command in its dependencies, this guarantees no valid command after RECONFIG.

We abstract the reconfiguration process of MPaxos into three states: NORMAL, RECONFIGURING, and TRANSFERING. RECONFIGURING implies that some replica has sent a RECONFIG command, and the command is not yet committed; TRANSFERRING state means that the RECONFIG command has been committed and the transfer of states is in progress. To know when transfer finishes, *TRANSFER-FINISH* command is defined and commit it in a majority of the new config. A receiver confirms this log after its transfer process is completed. Upon this command is committed successfully, the transfer state ends. The replica cannot submit normal commands in the RECONFIGURING and TRANSFERING states, while the RECONFIG command can be submitted at any time.

The reorganization process inherits the 3 steps and the 3 replica state from reconfiguration, and introduce 1 extra log type *REORGANIZE*. But reorganization conceptually just alter the roles some set of replicas plays. Hence it has less impact on the performance. Figure 1 shows the pseudocode of the protocol for choosing commands in MPaxos, and Fig. 2 shows the Execution logic of the REORGANIZE command.

4 Scheduling Framework

In this chapter, we present the scheduling framework of MPaxos which is responsible for making migration decisions. Suppose there are N replicas deployed in MPaxos: $\{d_1, d_2, ..., d_n\}$. A Working Cluster placement plan for some object θ with k nodes is denoted as $P = \{p_1, p_2, ...p_k\}$, where d_{p_i} is a replica in the Working Cluster. Periodically, MPaxos measures end-to-end latency between different replicas and stores the results in matrix L, where $L(i, j)$ is the round-trip-time (RTT) from d_i to d_j. The workload statistics is denoted as $S = \{(r_1, w_1), (r_2, w_2), ..., (r_n, w_n)\}$, (r_i, w_i) is the read and write workload on replica d_i.

To evaluate a placement plan, we provide a function $f(P, S, L)$ that evaluates a placement plan under a certain workload:

$$f(P, S, L) = -\frac{\sum_{i=0}^{n}(\alpha \cdot C(P, i) \cdot r_i + \beta \cdot C(P, i) \cdot w_i)}{n}$$

where $C(P, i)$ is replica d_i's commit latency under the placement plan P. α and β are weights indicated the importance of read and write latency.

Phase 1: Establish ordering constraints

Receiving NORMAL Command

Replica L on receiving $Request(\gamma)$ of type NORMAL from a client becomes the designated leader for command γ

1: **if** $L \notin W_{\gamma.key}$ **then**
2: send $Request(\gamma)$ to $R \in W_{\gamma.key}$
3: **else**
4: **if** $state[\gamma.key]$!= NORMAL **then**
5: push(γ) to waitQueue($\gamma.key$)
6: terminate
7: **else**
8: increment instance number $i_L \leftarrow i_L + 1$ $\{Interf_{L,\gamma}$ is the set of instances $Q.j$ such that the command recorded in $cmds_L[Q][j]$ interferes with $\gamma\}$
9: $seq_\gamma \leftarrow 1 + max(\{cmds_L[Q][j].seq | Q.j \in Interf_{L,\gamma} \cup 0\})$
10: $deps_\gamma \leftarrow Interf_{L,\gamma}$
11: $ver_\gamma \leftarrow Ver_{cur}[\gamma.key]$
12: $cmds_L[L][i_L] \leftarrow (\gamma, seq_\gamma, pre - accepted)$
13: send $PreAccept(ver_\gamma, \gamma, seq_\gamma, deps_\gamma, L.i_L)$ to all other replicas in Q_F, where Q_F is a fast quorum of $W_{\gamma.key}$ that includes L
14: **end if**
15: **end if**

Any replica R, on receiving $PreAccept(ver_\gamma, \gamma, seq_\gamma, deps_\gamma, L.i)$ of type NORMAL:

16: **if** $ver_\gamma < Ver_{cur}[\gamma.key]$(Working cluster has moved) **then**
17: reply $QuorumChanged(\gamma, W_{\gamma.key})$ to L, then L will update its working cluster and redo $PreAccept$
18: **else**
19: update $seq_\gamma \leftarrow max(\{seq_\gamma\} \cup \{1 + cmds_R[Q][j].seq | Q.j \in Interf_{R,\gamma}\})$
20: update $deps_\gamma \leftarrow deps_\gamma \cup Interf_{R,\gamma}$
21: $cmds_R[L][i] \leftarrow (\gamma, seq_\gamma, pre - accepted)$
22: reply $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to L
23: **end if**

Replica L (command leader for γ), on receiving at least F_W PreAcceptOK responses($F_W = \lfloor W/2 \rfloor$):

24: $cmd \leftarrow \gamma$
25: $canGoFast \leftarrow True$
26: **if** $state[\gamma.key]$!= NORMAL **then**
27: $cmd \leftarrow NOP$
28: $canGoFast \leftarrow False$
29: **else if** received a $PreAcceptOK$ with a REORGANIZE command not yet known to L **then**
30: $state[\gamma.key] \leftarrow$ RECONFIGURING
31: $cmd \leftarrow NOP$
32: $canGoFast \leftarrow False$
33: **else**
34: **if** $canGoFast$ and received $PreAcceptOK$'s from all replicas in a fast quorum($F_W + \left\lfloor \frac{F_W+1}{2} \right\rfloor$) except L, with seq_γ and $deps_\gamma$ the same in all replies **then**
35: run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
36: **else**
37: update $deps_\gamma \leftarrow Union(deps_\gamma$ from all replies)
38: update $seq_\gamma \leftarrow max(\{seq_\gamma$ from all replies$\})$
39: run Paxos-Accept phase for $(cmd, seq_\gamma, deps_\gamma)$ at $L.i$
40: **end if**
41: **end if**

Receiving REORGANIZE Command

Replica L on receiving $Request(\gamma)$ of type REORGANIZE from a client becomes the designated leader for command γ

1: **if** $L \notin W_{\gamma.key}$ **then**
2: send $Request(\gamma)$ to $R \in W_{\gamma.key}$
3: **else**
4: increment instance number $i_L \leftarrow i_L + 1$ $\{Interf_{L,\gamma}$ is the set of instances $Q.j$ such that the command recorded in $cmds_L[Q][j]$ interferes with $\gamma\}$
5: $seq_\gamma \leftarrow 1 + max(\{cmds_L[Q][j].seq | Q.j \in Interf_{L,\gamma} \cup 0\})$
6: $deps_\gamma \leftarrow Interf_{L,\gamma}$
7: $ver_\gamma \leftarrow Ver_{new}[\gamma.key]$
8: $cmds_L[L][i_L] \leftarrow (\gamma, seq_\gamma, pre - accepted)$
9: $state[\gamma.key] \leftarrow$ RECONFIGURING
10: send $PreAccept(ver_\gamma, \gamma, seq_\gamma, deps_\gamma, L.i_L)$ to all other replicas in Q_F, where Q_F is a fast quorum of $W_{\gamma.key}$ that includes L
11: **end if**

Any replica R, on receiving $PreAccept(ver_\gamma, \gamma, seq_\gamma, deps_\gamma, L.i)$ of type REORGANIZE:

12: **if** $state[\gamma.key]$ == NORMAL or $ver_\gamma > Ver_{new}[\gamma.key]$(receive new REORGANIZE command) **then**
13: $state[\gamma.key] \leftarrow$ RECONFIGURING
14: update $seq_\gamma \leftarrow max(\{seq_\gamma\} \cup \{1 + cmds_R[Q][j].seq | Q.j \in Interf_{R,\gamma}\})$
15: update $deps_\gamma \leftarrow deps_\gamma \cup Interf_{R,\gamma}$
16: $cmds_R[L][i] \leftarrow (\gamma, seq_\gamma, pre - accepted)$
17: reply $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to L
18: **else**
19: terminate
20: **end if**

Replica L (command leader for γ), on receiving at least F_W PreAcceptOK responses($F_W = \lfloor W/2 \rfloor$):

21: **if** received $PreAcceptOK$'s from all replicas in a fast quorum, with seq_γ and $deps_\gamma$ the same in all replies **then**
22: run Commit phase for $(\gamma, seq_\gamma, deps\gamma)$ at $L.i$
23: **else**
24: update $deps_\gamma \leftarrow$ Union($deps_\gamma$ from all replies)
25: update $seq_\gamma \leftarrow max(\{seq_\gamma\}$ of all replies)
26: run Paxos-Accept phase for $(\gamma, seq_\gamma, deps\gamma)$ at $L.i$
27: **end if**

Slow-Path Paxos-Accept phase

Fast-Path Commit phase

Fig. 1. The basic migratable Paxos protocol for choosing commands

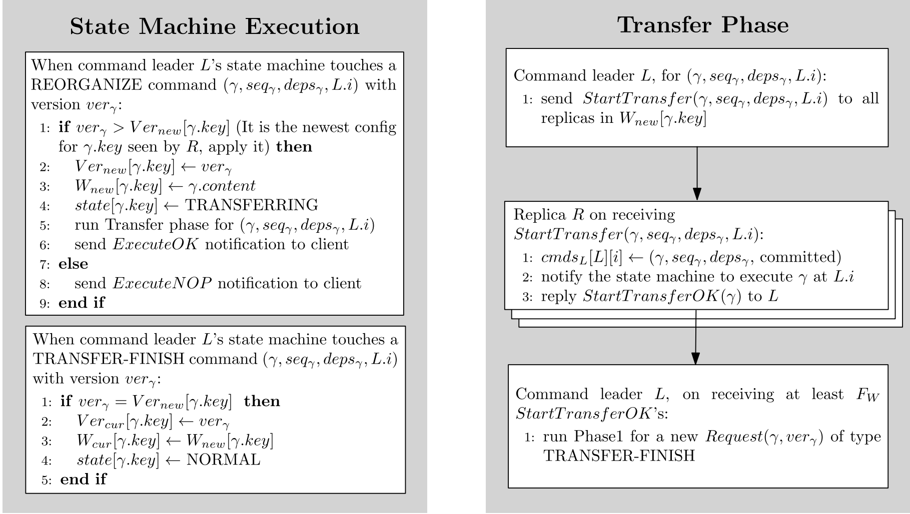

Fig. 2. The execution and transfer phase of migratable paxos

The evaluation of $C(P, i)$ is split into 2 steps:

1. Send a command from d_i to the closest replica in P (denoted p_L), the latency is:

$$C_1 = \begin{cases} L(i, p_L), & \text{if } i \text{ not in } P \\ 0, & \text{if } i \text{ in } P \end{cases}$$

2. Commit latency. The command could be committed in the Fast-Path or the Slow-Path, we specify a parameter e to represent the possibility of going through Fast-Path, then the latency is:

$$C_2 = e \cdot Fast(P, p_L) + (1 - e) \cdot Slow(P, p_L)$$

The Fast-Path only involves one round-trip between a Fast-quorum of working replicas. As a Fast-quorum contains $F_W + \lfloor \frac{F_W+1}{2} \rfloor$ replicas ($F_W = \lfloor W/2 \rfloor$, with W indicates the cardinality of P), the network latency of commit in Fast-Path is roughly the same as the third quartile of latencies between p_l and other replicas in P, that is:

$$Fast(P, pL) = 3rd\ Quartile(\{L(p_l, p_i) \mid p_i\ in\ P\})$$

The Slow-Path involves two round-trips between a Slow-Quorum (i.e. a majority). So the network latency is 2-times the median of the latencies from p_l to replicas in P.

$$Slow(pl) = 2 \cdot Median(\{L(p_l, p_i) \mid p_i\ in\ P\})$$

5 Evaluation

We implement MPaxos on Paxi, a framework that implements EPaxos and other Paxos protocols, to evaluate and compare their performance. The implementation is in Go, version 1.11.2, and we release it as an open-source project on GitHub at https://github.com/dante159753/MPaxos.

We evaluated MPaxos on Amazon EC2, using small instances (two 64-bit virtual cores with 2 GB of memory) for both state machine replicas and clients, running Ubuntu Linux 18.04.2.

5.1 Workloads

We specify two types of workloads: (1) hot spots are static and from one or two continents; (2) a workload with a request peak at 9:00 am local time, and the relationship between the number of requests and local time is subject to normal distribution.

Our tests also capture conflicts, an important workload characteristic – conflict is a situation when potentially interfering commands reach replicas in different orders. Conflicts affect EPaxos and MPaxos. As write requests usually occupy no more than 2% of all requests, we believe that 0% and 2% command interference rates are the most realistic [4].

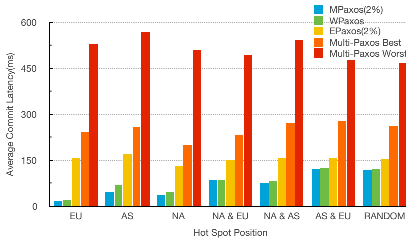

Fig. 3. Commit latency under static workloads

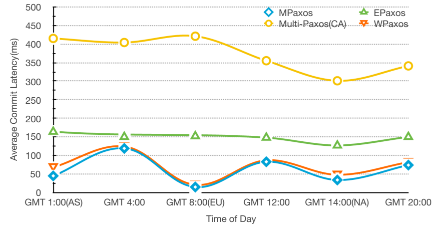

Fig. 4. Latency under regularly shifting workloads

5.2 Latency in Wide Area

We evaluate MPaxos with nine replicas (tolerating one failure, so the minimal size of Working Cluster is 3). The replicas are located in Amazon EC2 datacenters in California (CA), Virginia (VA) , Oregon (OR) , Japan (JP), Korea (KR), Singapore (SG), London (LON), Paris (PAR), and Sweden (SE). We set $F_W = 1$ for MPaxos and $F = 0, f = 1$ for WPaxos [11].

Figure 3 shows the average client latency for MPaxos, EPaxos, Multi-Paxos and WPaxos under static workloads, where WPaxos is a recent leader-based

Paxos protocol optimized for migration scenario. The X-Axis indicates the positions of clients. With nine replicas, protocols with static clusters – such as EPaxos and Multi-Paxos with static clusters – produce high latency, while protocols with migratable clusters – such as MPaxos and WPaxos – achieve lower latency, and MPaxos outperforms WPaxos because of the leaderless command committing fashion. And the last group of the test (RANDOM), in which requests come from a random client of the world, shows that MPaxos also works well under irregular workloads.

Figure 4 shows the average client latency for these protocols under the shifting workload. MPaxos also achieves the lowest commit latency by timely responding to the shifting workload. WPaxos performs very close to MPaxos. Nevertheless, as shown in Fig. 5, MPaxos outperforms WPaxos in migration cost. Although WPaxos only need 1 round of communication during migration instead of 2 round in MPaxos, it suffers from a larger Phase-1 quorum size (at least 6 replicas), where MPaxos only need to communicate with 2 replicas twice.

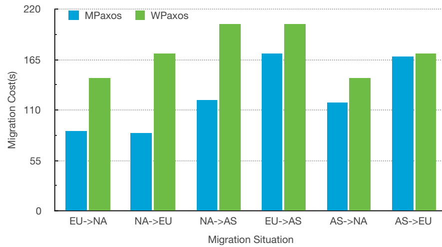

Fig. 5. Migration cost between different continents

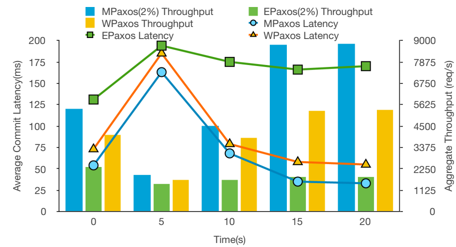

Fig. 6. Latency and workload when an emergency occurred in the 3rd second.

To evaluate how MPaxos and other protocols perform under emergency, we initialize a workload by deploy 300 clients in NA, then simulate the emergency by shutting down all clients in NA and starting 300 new clients in AS in the 3rd second. Figure 6 shows how latency and throughput change. It shows that MPaxos consistently retain the lowest latency and highest throughput by timely responding to the changes in client characteristics. The latency of WPaxos is roughly as good as MPaxos, but the single-leader-per-object style limits the throughput.

6 Conclusion

In this paper, we propose MPaxos, a Paxos-based protocol for shifting workloads. We show that designing specifically for the client characteristics yields significant performance rewards. MPaxos includes two main proposals: (1) use Working Clusters to make Replication quorums small and close to users, (2) Reorganization that enables Working Clusters timely responding to workload

shifting with low migration cost. These proposals improve performance significantly as we show on a real deployment across 9 datacenters.

Acknowledgement. This work is supported by National Key R&D Program of China (2018YFB 1003404), National Science Foundation of China under grant number 61672232 and Youth Program of National Science Foundation of China under grant number 61702189.

References

1. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. (TOCS) **16**(2), 133–169 (1998)
2. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)
3. Hunt, P., Konar, M., Junqueira, F.P., et al.: ZooKeeper: Wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference, vol. 8(9) (2010)
4. Moraru, I., Andersen, D.G., Kaminsky, M.: Egalitarian paxos. In: ACM Symposium on Operating Systems Principles (2012)
5. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1), 63–73 (2010)
6. Shen, Z., Jia, Q., Sela, G.E., et al.: Follow the sun through the clouds: application migration for geographically shifting workloads. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 141–154. ACM (2016)
7. Bryant, R., Tumanov, A., Irzak, O., et al.: Kaleidoscope: cloud micro-elasticity via VM state coloring. In: Proceedings of the Sixth Conference on Computer Systems, pp. 273–286. ACM (2011)
8. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 51–60. ACM (2009)
9. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 335–350 (2006)
10. Guo, J., Chu, J., Cai, P., Zhou, M., Zhou, A.: Low-overhead Paxos replication. Data Sci. Eng. **2**(2), 169–177 (2017). https://doi.org/10.1007/s41019-017-0039-z
11. Ailijiang, A., Charapko, A., Demirbas, M., et al.: WPaxos: wide area network flexible consensus. IEEE Trans. Parallel Distrib. Syst. **31**(1), 211–223 (2019)