GIRAFFE: A Scalable Distributed Coordination Service for Large-scale Systems

Xuanhua Shi*, Haohong Lin*, Hai Jin*, Bing Bing Zhou†, Zuoning Yin‡, Sheng Di§, Song Wu*
*Services Computing Technology and System Lab, School of Computer
Huazhong University of Science and Technology
†Centre for Distributed and High Performance Computing
School of Information Technologies, University of Sydney
‡GraphSQL Inc.
§INRIA and Argonne National Laboratory

***Abstract*—The scale of cloud services keeps increasing over time, significantly introducing huge challenges in system manageability and reliability. Designing coordination services in cloud is the right track to solve the above problems. However, existing coordination services (e.g., Chubby and ZooKeeper) only perform well in read-intensive scenario and small ensemble scales. To this end, we propose *Giraffe*, a scalable distributed coordination service. There are three important contributions in our design. (1) Giraffe organizes coordination servers using interior-node-disjoint trees for better scalability. (2) Giraffe employs a novel Paxos protocol for strong consistency and fault-tolerance. (3) Giraffe supports hierarchical data organization and in-memory storage for high throughput and low latency. We evaluate Giraffe on a high performance computing test-bed. The experimental results show that Giraffe gains much better write performance than ZooKeeper when server ensemble is large. Giraffe is nearly 300% faster than ZooKeeper on update operations when ensemble size is 50 servers. Experiments also show that Giraffe reacts and recovers more quickly than ZooKeeper against node failures.**

Keywords*—Coordination Service; Distributed Cloud System; Scalability; Fault Tolerance; Consistency

I. Introduction

With immensely increasing scale of cloud services, it is common that a cloud service is built on tens of thousands of machines within one data center or across multiple data centers, or even fully distributed commodity machines (e.g., Bitdew [1], SlapOS [1]), which introduces huge challenges to manageability and reliability in the cloud. On the one hand, it is non-trivial to enable such a large or global scale system to update configuration or architecture, due to the inevitable heavy burden of changing system components and probable inconsistent membership of services introduced. On the other hand, the bottleneck issue will raise up significantly, resulting in the frequent inconsistent states inside the whole system and thus complicated recovery problem from system fail-over. For example, jobtracker of Hadoop [2] suffers serious bottleneck problem when managing thousands of slave machines in a Hadoop cluster.

[1]https://www.slapos.org

Recently, coordination service has become the key technology to solve the above problems. It is often used to achieve better scalability, manageability, and reliability in different cloud systems, like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and other large-scale distributed systems. A whole cloud system usually contains multiple loosely-coupled self-organizing subsystems, and it is supposed to possess the following five important features.

- *Ease-of-use management*: One can efficiently query the basic and common meta services like message queue service in the system. For instance, many NoSQL databases and transactional systems [3], [4] use coordination service components for metadata management, name service and load balancing. OpenStack adopts message queue service to smoothen the management which is also transparent to users.
- *Strong consistency*: Essential information (such as membership and leader election) is dynamically shared/transferred across all different subsystems/components, to guarantee the consistent system states. The latest release of Hadoop, YARN [5], for example, migrates the keepalive mechanism with most of Remote Procedure Calls (RPCs) into ZooKeeper to disperse the burden of jobtracker.
- *High robustness*: The data under the management of the coordination service should always be kept available even in case of component failures in the system.
- *High scalability*: A coordination service should be able to scale gracefully with the increase of the ensemble size.
- *High Read/Write performance*: A coordination service should provide high performance in both read-intensive and write-intensive scenarios.

Basically, most of the existing designs of coordination service (such as [6], [7], [8], [9], [10], [11], [12]) have extensively explored the first two features, but not for the last three. Existing coordinating protocols, such as Paxos, Fast Paxos, and Zab [13] are not, in general, the best consensus protocol for the wide-area or dynamic environments. For example, the Zab in Zookeeper is leader-centric, and the leader does most of the work. When one leader crashes, the system will not be available until a new leader is elected, and the recovery time is highly subject to system scale. The recovery time of zookeeper with the scale of 9 is three times of the scale of 3. In general, the operations of coordination service can be simplified into two types: *read operation* (getting information from coordination service component) and *write operation* (writing or publishing information to the coordination service component). The existing systems cannot deal with write-intensive scenarios very well. For example, Chubby [10] provides a lock semantic and preserves both strong consistency and high availability for applications in a loosely coupled distributed system. ZooKeeper [11] provides a wait-free coordination service interface which achieves better read performance than Chubby, but with a relaxed strong consistency model. Both of them adopt a primary-backup model, which is subject to a central-controlled consensus protocol. A primary process (a.k.a., leader) collects all of information/operations and sends them back to replicas in a particular order. This will definitely cause a serious bottleneck in a large scale write-intensive system. As we evaluated, the write performance of Zookeeper with scale size = 21 (i.e., using 21 slave nodes) will be degraded down to 50% in comparison to the scale size = 3. That is, Chubby and ZooKeeper face serious scalability issues in the write-intensive scenarios.

To address the issues listed above, we propose a novel coordination service called *Giraffe*[2]. There are three key contributions in our design.

- Giraffe organizes memberships using interior-node-disjoint forest for better scalability. In the forest, every tree has the same set of tree nodes, but the interior nodes of one tree must be the leaves of others. Giraffe nodes are partitioned into groups with different unique identifications (denoted as *colors* in the papers) and built into trees with the interior-node-disjoint property. Such a hierarchical topology can disseminate information reliably and dynamically achieve load balancing.
- A scalable consensus protocol based on a novel *Group Paxos* algorithm is implemented in Giraffe for strong consistency and fault-tolerance. Similar to Chubby and ZooKeeper, read operations are processed locally in Giraffe, while the update operations in Giraffe will be dynamically synchronized among replicas. Giraffe guarantees the order of transactions in each group as well as the global order in the whole system. To guarantee the global order, our designed *Group Paxos* algorithm adopts multiple coordinators to run the proposal phase in different groups and then determines the vote globally. This can effectively guarantee the total ordering property of broadcast transactions and also achieve better load balancing than the primary-backup model. It also reduces overheads of coordinators and performs correctly even in case of node insertion/departure.
- Hierarchical data organization and in-memory storage are improved in Giraffe for high throughput and low latency. Giraffe provides both blocking and non-blocking primitives to the cloud applications. Different from the server-client service in Chubby and ZooKeeper, Giraffe provides advanced service models for applications in different scenarios. Giraffe extends service interface to both local and remote application interfaces to provide coordination services for servers and applications. As mentioned in ZooKeeper, when using blocking primitives, slow or faulty clients may have negative impact on the performance of healthy and fast clients. Therefore, blocking primitives (e.g., lock) are enabled when Giraffe is implemented as an inner-component of an application.

We evaluate Giraffe and ZooKeeper on the High Performance Computing Cluster (HPCC) [3] platform. Experiments show that Giraffe significantly outperforms ZooKeeper in update operations, for a large scale and dynamic distributed computing environment. In absolute terms, it is nearly 300% faster than ZooKeeper in write operations when the scale size is 50. Giraffe also has more stable update throughput than ZooKeeper when the ensemble size increases, and also recovers quickly from node failure.

The rest of this paper is organized as follows: Section II presents the system overview of Giraffe. Section III describes the design and implementation. Section IV presents the experimental results. Related work is discussed in Section V and we conclude the paper in Section VI .

II. System Overview

Figure 1 shows the system architecture of Giraffe.

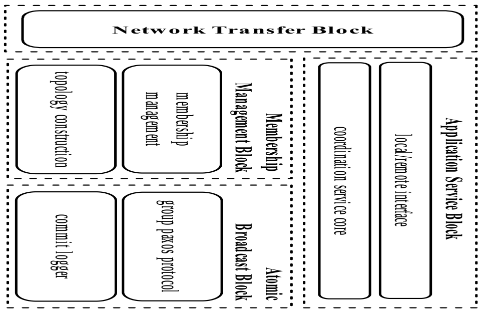

Fig. 1: System architecture

There are four building blocks in the system:

- **Membership Management** provides the construction and management of hierarchical Giraffe server topology for dynamic membership. In this part, we take into account reliability and scalability issue. To minimize the data loss when some Giraffe servers fail in data dissemination, we employ a hierarchical topology - interior-node-disjoint trees. In the topology, there are several trees and each node is a Giraffe server existing in all the trees. So there must be several data broadcast chains through which messages can be delivered from roots to one Giraffe server. So one Giraffe server can get the broadcast data even some servers are failed in the broadcast chains. Unlike other forest topologies, we exploit a near-balance constructing mechanism. In this

[2]The source code is available at https://github.com/haohonglin/Giraffe

[3]http://grid.hust.edu.cn/hpcc

mechanism, we minimize the change of membership in the trees to reach the expected balance.

- **Atomic Broadcast** exploits a novel Paxos protocol for strong consistency and high durability to preserve the total order of update events among replicas. We propose a novel Paxos variant, called *Group Paxos*, which is based on the interior-node-disjointed trees. *Group Paxos* addresses two issues: *quorum size*[4] and *coordinator failure* discussed in [7], [14], [15]. In *Group Paxos*, we divide acceptors[5] into several groups, and use multiple group coordinators to make votes in parallel in different groups and globally. Upon the *Group Paxos*, we implement *atomic broadcast mechanism* to commit transaction updates in the sequential order among all servers.
- **Application Service** provides both coordination service semantics and application interface. Giraffe provides for applications a simple file system interface, which is handled by the *local/remote interface* component. Based on the low-level file interface, the *local/remote interface* can build high-level semantics regarding queue and lock. Through the *local/remote interface*, applications store information (e.g., metadata and configuration) to coordination service core of Giraffe. Moreover, applications can get data from Giraffe and observe the data modifications by the call-back mechanism, so that applications can implement many distributed algorithms without blocking the application. In the *coordination service core*, the data of applications is organized in a hierarchical key-value structure in memory, where the key is equal to a file path and the value is the corresponding file content. Once applications create or modify data, Giraffe synchronizes the updates throughout the Giraffe cluster using a novel consensus protocol of *atomic broadcast block*.
- **Network Transfer Block** provides a high performance transformation protocol and a simple development interface for high-level network applications.

III. Design and Implementation

In this section, we first discuss the *membership management protocol* to achieve a scalable ensemble for coordination service. And then we discuss the *consensus protocol* which guarantees the strong consistency among the Giraffe ensemble.

A. Membership Management Protocol

A dynamic membership management protocol is implemented to build the interior-node-disjoint trees for reliable data transmission and load balancing. Furthermore, the membership management provides topology that *group paxos* requires and improves the efficiency of commitments of *group paxos*. Different from other load balance algorithms, such as DHT and round-robin, interior-node-disjoint trees is able to deliver data reliably to all the nodes even some are in failure. Also the hierarchical topology is more helpful to run *group paxos* and to relieve the overhead of the central node, e.g., leader.

[4]slow vote performance of paxos with big quorum size.

[5]*acceptors* mean the processes of making votes for messages

Two issues are addressed in the membership management protocol: 1) membership registration and notification mechanism which guarantees the consistent sequence of membership updates inside Giraffe cluster and even distribution of nodes in trees; 2) interior-node-disjoint trees construction which is a solution for forest deployment and reliable delivery network.

Membership Registration and Notification. The Giraffe nodes are constructed into interior-node-disjointed trees. Each tree has a unique identification, called *color*. Each Giraffe node has two properties: 1) *IP address* for exchanging membership information with others and 2) *node color* for denoting which tree the node belongs to. In Giraffe cluster, the central manager, called *Rooter*, determines the number of the trees by configuration, promotes membership version if the membership changes, generates the color information for new nodes and broadcasts membership updates. Initially the first Giraffe node in the cluster is configured as *Rooter*.

Giraffe uses version membership and central management of *Rooter* to register new member and update membership. The view of each membership is associated with a version number. When a membership change occurs because of the registration or departure of some node, *Rooter* increments the membership version and disseminates the update as well as the new version to other nodes. *Rooter* holds the whole cluster membership view, so it is easy to manage color distribution. When a new node arrives, it firstly retrieves the membership view from an existing node, and then requests color information from *Rooter* by sending a message. This message includes its IP and recent membership version that it has fetched from the other node. *Rooter* compares its current version with that in the request message and then sends membership update and color information back to the new node. As the registration is managed centrally, *Rooter* holds the latest membership view in the cluster, and so the new node is able to build the latest view by incrementing the version obtained from *Rooter*.

Node's failure or departure is detected by periodic heartbeats between its parents and its children. In the interior-node-disjoint trees, a node is either an interior or leaf member in every tree, so it may have several parents and children. For failure detection a node is assigned a lease, which is typically a timer, to detect aliveness of the node. A node needs to heartbeat with its parents and children within its lease. Otherwise, it will be treated as expired and other nodes will notify *Rooter*, and then *Rooter* distributes such information throughout trees. Compared with primary-backup systems, *Rooter* acknowledges node status from other nodes and so the overhead of *Rooter* for failure detection is reduced. If the *Rooter* fails, the root of a tree which maintains the latest membership view among the root group will be elected as the new *Rooter*.

Membership updates and other messages are delivered along the interior-node-disjoint trees. We duplicate one message into several messages and deliver each message along each tree. So there are several paths for a node to get one message. However, each node has a footstep in every tree, causing the message to be recursively transferred endlessly. To address this problem, when deliver a message in a tree, we attaches a tag containing the tree color to the message. A node delivers messages to its children only when it has the same color with the message tag.

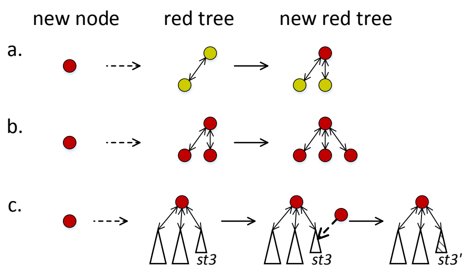

Fig. 2: add new red node into a tree with red color

This solution prevents leaf nodes from delivering message to nodes of other trees.

Construction of Hierarchical Topology. The number of children of each node and the height of each tree significantly impact load balancing, throughput and latency. If trees are high, it takes long to deliver information from the root to leaves. If a node has more children, it would suffer more loads on information dissemination. We use *MAX_COLOR_NUM* to denote the max number of trees in the forest and use *MAX_CHILDREN_SIZE* to denote the max number of children a node can have. These two attributes are set by *Rooter*.

A node in the forest contains the following attributes: *color*, *IP*, *parent*, *children*, *descendants* and *height*. *Color* is an integer to determine which tree a node belongs to, and *IP* is the address. *Parent* is a 1-dimensional array that contains node's parents in each tree. *Children* is a 2-dimensional array in which the items are the node's direct children in each tree. *Descendants* is a 1-dimensional array that contains the number of its descendants in each tree. *Height* is an array containing the distance of the node to leaf in each tree. Both *descendants* and *height* are essential attributes to indicate the overhead of dissemination and the message latency within each sub-tree.

How to build a forest topology is described as follows. Basically, there are two essential functions to build interior-node-disjoint trees: *insert_into_tree* and *remove_from_tree*, whose pseudo-codes are shown as Algorithm 1 and Algorithm 2 respectively. we perform the two functions into each tree in the forest when a node is added to the forest or removed from the forest. There are two basic steps in adding a new node to each tree in the forest: (1) select the shortest sub-tree with the smallest height; (2) continue to select the node which has the fewest descendants in the sub-trees. Then, the selected node will serve as the parent node of the new node to insert. When removing a node, we select a proper sub-node to replace it. In the procedure of building the forest topology, each node keeps track of the height and the descendant number of its children.

Function *insert_into_tree*, as shown in Algorithm 1 and Figure 2, is triggered upon receiving information from the registered node. A node is added into a tree based on three principles:

1) a node not with tree.color tends to be a child of the node with tree.color (lines 3-11 in Algorithm 1).
2) new node tends to be a child of unsaturated node with tree.color (lines 12-16 in Algorithm 1).
3) if a node is saturated, then new node tends to be a descendant of its child who has least height or fewest descendants (lines 17-19 in Algorithm 1).

Algorithm 1 insert node into tree

```
 1: procedure INSERT_INTO_TREE($tree$, $newNode$, $color$)
 2:     $root = tree.root$
 3:     if $root.color$ != $color$ and $newNode.color == color$ then
 4:         $newNode.children[color].first \leftarrow root$
 5:         $root.parent[color] \leftarrow newNode$
 6:         $tree.root \leftarrow newNode$
 7:         while $newNode.children[color].size < MAX\_CHILDREN\_SIZE$ do
 8:             move $node$ from $root.children[color]$ to $newNode.children[color]$
 9:         end while
10:         return
11:     end if
12:     if $root.children[color].size < MAX\_CHILDREN\_SIZE$ then
13:         $root.children[color][nextchild] \leftarrow newNode$
14:         $newNode.parent[color] \leftarrow root$
15:         return
16:     end if
17:     $n$ = select a $node$ in $root.children[color]$ which does not belong to the color set, or which has the least height or descendants
18:     $st$ = sub-tree of the $tree$ whose $root$ is $n$
19:     INSERT_INTO_TREE ($st$,$newNode$,$color$)
20:     return
21: end procedure
```

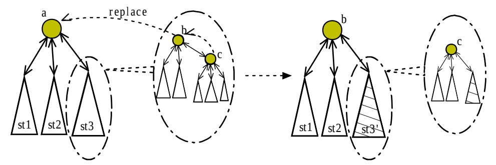

Fig. 3: Node departure in a tree with yellow color

We give an example to illustrate how to insert a new node to the tree, as shown in Figure 2. We suppose the new node is red, the three ought to be red and *MAX_CHILDREN_SIZE* is 3. **a.** The root of tree is not red so there are not red nodes in the tree. The new red node will become the new root and other nodes are its children or descents. **b.** The root of the tree is red and the number of children of the root is less than *MAX_CHILDREN_SIZE*. Then the new red node will become the child of the root. **c.** The root of the tree is red and the number of its children already reaches the *MAX_CHILDREN_SIZE*. Then a sub-tree *st3* is picked from the sub-trees of the root and insert the new node into *st3*. *st3* should have the least height or fewest descendants among all the sub-trees.

Function *remove_from_tree*, as shown in Algorithm 2, is triggered upon receiving notification that a node failed. To remove a node, one node to replace the failed node must be selected from the children set of the failed node (lines 2-4 in Algorithm 2). When replacing the failed node, if other children of the failed node are linked directly to the selected one, it may lead to over saturation (lines 5-9 in Algorithm 2). In this case, the sub-tree of the selected node needs to be adjusted. As shown in Figure 3, a given sub-tree *ST* with yellow color has a failed root *a*. *st3* is highest among all sub-trees of node *a*. In this case, node *b*, the root of *st3* is selected to replace *a*. And its child *c* becomes the root in the adjusted sub-tree *st3*.

B. Consensus Protocol

Group Paxos. *Group Paxos* is a new Paxos variant we propose. It aims to reduce the overhead of central node (leader)

Algorithm 2 remove node from tree

```
procedure POP_ROOT($rmNode$,$color$)
    $n$ = the $node$ which has the most height or descendants among $rmnode.children[color]$
    remove $n$ from $rmNode.children[color]$
    move $rmNode.children[color]$ to $nodeSet$
    while $nodeSet.size < MAX\_CHILDREN\_SIZE$ - 1 do
        move $node$ to $nodeSet$ from $n.children[color]$ which has the least height or descendants
    end while
    $newRoot$ = POP_ROOT($n$,$color$)
    add $newRoot$ into $nodeSet$ and let $nodeSet$ as $n.children[color]$
    return $n$
end procedure
procedure REMOVE_FROM_TREE($tree$,$ip$)
    $rmNode$ = the node in the $tree$ of which $node.ip$ is $ip$.
    $rpNode$ = POP_ROOT ($rmNode$,$tree.color$)
    if $tree.root$ is $rmNode$ then
        $tree.root \leftarrow rpNode$
    else
        $parent \leftarrow rmNode.parents[color]$
        remove $rmNode$ from $parent.children[color]$
        insert $rpNode$ into $parent.children[color]$
    end if
    return $rmNode$
end procedure
```

even when the scale of acceptors is increasing. It assembles acceptors into groups and hence reduces the number of nodes that central node manages directly. It is useful to solve the performance and scalability issues caused by central node in other paoxos protocols. And also Group Paxos achieves better robustness since *Group Paxos* recovers more quickly than other paxos algorithms when central node is failed.

Group Paxos is implemented on interior-node-disjoint trees, as shown in Figure 4.

- ***Group***: the collection of nodes which are of the same color. The number of groups is equal to the numberof trees in forest-topology.
- ***Group coordinator(gc)***: the root of a tree. The group coordinator is supposed to lead vote in its group and coordinate votes with a global leader. It is also a learner.
- ***Group acceptor(ga)***: the acceptor in a group, typically, the interior node of a tree. It is also a learner.
- ***Global leader(gl)***: the leader of all the group coordinators to guarantee order properties of proposals and transactions.

In the forest with different colors, we define a group with color ***k*** as ***Group-k*** or ***g-k***, the acceptors number of ***Group-k*** as ***Group-k-size***, the group coordinator of ***Group-k*** as ***Group-k-coordinator*** or ***gc-k*** and the group acceptor as ***Group-k-acceptor*** or ***ga-k***. In *Group Paxos*, we parallelize the Paxos phases among different groups and use global leader to pipeline messages proposed by proposers.

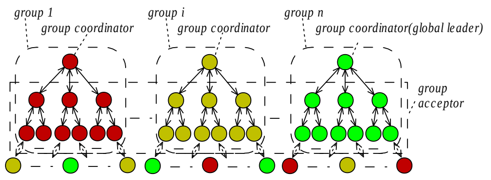

Fig. 4: Group Paxos deployment in duplicate interior-node-disjoint trees

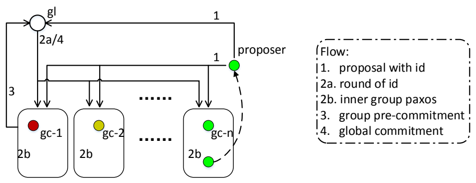

Fig. 5: Group Paxos. 1.A proposer sends proposal request to global leader and group coordinators. 2.The group coordinators will sends the request and gather the promises of its group acceptor. Meanwhile, global leader determines the order of proposal and sends it to group coordinators. 3.Group coordinators send pre-commit request to global leader if a quorum of acceptor in its group accept the proposal. 4.Global leader sends the commitment to all groups.

The procedure of *Group Paxos* is presented in Figure 5. There are 4 steps in total.

- **Step 1:** A proposer, which could be any node in the Giraffe cluster, sends to global leader and all the group coordinators the proposal message with unique id.
- **Step 2:**
  - **Step 2a:** Global leader sends proposal message with new round to group coordinators.
  - **Step 2b:** In the meantime, each group coordinator performs Paxos within its own group. When the group coordinator receives a majority of group acceptors from its group, it supposes that the proposal message can be committed in its group.
- **Step 3:** Then it sends to the global leader a group-precommit message consisting of the group size, round number and proposal id. We propose a mechanism to determine whether a proposal message could be committed or not. We use counter to calculate the sum of group size in group-precommit message for each proposal. Only when the counter value reaches half of the cluster size can the message be committed.
- **Step 4:** In this case, the global leader sends a commit message to all group coordinators and then each group coordinator commits such transaction in its group.

In our *Group Paxos*, global leader does much less work than leader in the classic Paxos and so it efficiently reduces overhead as well as the message lost rate. Each group coordinator does a similar work as the coordinator in the classic Paxos. However, it works in a group and commits transactions hierarchically. Multiple groups enable Paxos to run instances in parallel efficiently even in a large-scale system.

The key advantage of such a *Group Paxos* algorithm design is two-fold, described as follows:

- **Efficiency.** According to the algorithm of constructing interior-node-disjoint trees, the size of Group-k is approximately equal to that of other groups, ensuring the balance of each inner-group paxos. The *Group Paxos* instance would become efficient when each group takes the same time to finish its *inner-group paxos* procedure. In the classic Paxos, disk write is one solution for fault-tolerance, but a performance bottleneck of the Paxos procedure. We reduce to one disk write per

transaction commit by co-locate deployment in which coordinator and acceptor are also learners. In *Group Paxos*, coordinator in different groups manages majority actions of its group. Only when coordinator receives acknowledgements of one proposal from the majority of group acceptors, it writes the proposal to disk before sending *commitment* message to its acceptors.
As discussed in paxos and ZooKeeper, the time of leader election becomes longer as the election group becomes larger. In *Group Paxos*, the global leader is elected from a small group which is organized by group coordinators. As the number of group coordinators is much smaller than the total group of members in *Group Paxos*, the election time maintains stable. In this way, we achieve quick reaction of leader failure.

- **Fault-tolerance.** Global leader and group coordinator are two important roles in *Group Paxos*. Global leader handles global Paxos instances among group coordinators and makes decision globally. Group coordinator only handles local Paxos instances inside its group and serializes the Paxos events. Global Paxos and local Paxos can interleave with each other. So how to handle the failure of global leader and group coordination is important to the correctness of *Group Paxos*. We will discuss three situations in the following:
1. Suppose only global leader fails, one new global leader will be elected correctly among group coordinators by Paxos and then synchronize the states among group coordinators. As soon as the new global leader is elected, all the global Paxos is processed by the new one, so the old global leader will not effect the *Group Paxos* even if it comes up from failure.
2. Suppose group coordinator in Group-k fails, the failure will be detected by rooter and its children. The topology will be reconstructed and the new root of tree-k will become the new group coordinator in Group-k. The new group coordinator will synchronize state with global leader and its group acceptors. As shown in Figure 4, the acceptors in one group are also the learner of other groups. So the acceptors still can learn the commitment values from other groups even their group coordinator is failed. Also it is possible that in one group, there coexist two group coordinators including the old one. In this situation, we have more group coordinators than groups. Then f+1 will not be the true majority if the number of groups is 2f+1. To address this problem, we tag the messages of global Paxos by global epoch + group id + group epoch. The messages with the same group id are considered as the messages from the same group coordinator of the idth group. Even in the execution of one global Paxos instance, there are two co-existing group coordinators in the same group (such as group K), other group coordinators can tell the old one from the two group-K-coordinators by group epoch + global epoch and then ignore the messages from the old one.
3. Suppose both global leader and group coordinator fail, in *Group Paxos* the elections of the global leader and group coordinator are in different groups, so eventually there will be new correct global leader and group coordinator elected. Also it is important to synchronize the status after election. If global leader makes a decision to globally commit one event, but after sending the commitment to Group-k-coordinator, both global leader and Group-k-coordinator fail. *a)* If Group-k-coordinator fails before it send commitment to its group learners, the global commitment has not been learned before election, so after global leader is elected, it will synchronize status with all the group coordinators and commit the events that should be committed. *b)* If Group-k-coordinator fails after the commitment is learned by its group learners, it means that in the new global epoch the commitment should be committed before any new commitment. If the new global leader is elected before the new group coordinator, the global leader will commit all the events which has been pre-committed in some group. So all the pre-commitments in any group except Group-K will be committed. If the latest global commitment ***A*** was happened in Group-K and the event of ***A*** did not exist in the pre-commitments of other Group, it means that Group k is the majority quorum of global decision, so unless the new group-k-coordinator comes up there are none new commitment happened in the new global epoch. After new Group-k-coordinator is elected, it does not vote for the new proposal until it finishes synchronizing status with new global leader. So even global leader and group coordinator fail at the same time, *Group Paxos* can guarantee the consistency after election.

Atomic Broadcast Protocol. We implement an atomic broadcast protocol for coordination service upon *Group Paxos*. Our protocol consists of three phases: discovery phase, synchronization phase and broadcast phase. Discovery phase and synchronization phase are for global leader election and global synchronization. Broadcast phase is for committing proposals. In the following phase, gc.p means the latest promise message of gc while gc.a means the latest accepted message of gc.

Discovery phase:

Step gc.1.1 a group coordinator sends to the prospective leader gl its last promise (gc.p), a GEPOCH message
Step gl.1.1 upon receiving GEPOCH messages from a quorum Q of coordinator, the prospective global leader gl proposes NEWGEPOCH message (e') to group coordinators in Q. the new global epoch e' is larger than any e received in a GEPOCH message
Step gc.1.2 upon receiving the NEWGEPOCH(e') from gl, if $gc.p < e'$, then make $gc.p = e'$ and acknowledge the new epoch proposal NEWGEPOCH(e'). Acknowledgment ACK-E (gc.a, h) contains the id of latest accepted transaction and its history
Step gl.1.2 once it receives a confirmation from each group coordinator from Q, then it selects the history of one group coordinator gc in Q to be the initial history I'. gc satisfies that every gc' in Q, $gc'.a < gc.a$ or $(gc'.a = gc.a) \wedge (gc'.gid < gc.gid)$

Synchronization phase:

Step gl.2.1 global leader proposes NEWGLEADER

(e', I') to all group coordinators

Step gc.2.1 upon receiving NEWGLEADER (e', I') from gl, then executes if $gc.p = e'$

1. gc sets gc.a to e'
2. for each $< v, z >$ in I', accepts$< e', < v, z >>$, make h = I' <In this phase, gc broadcasts NEWGC$< e', I', groupid >$ to its group.>
3. gc acknowledges to gl

Step ga.2.2 once ga in group-id receives NEWGC$< e', I', groupid >$, ga sets ga.a to e', accepts every $< v, z >$ and makes h = I'

Step gl.2.2 upon receiving the acknowledgments from a quorum of group coordinator, gl sends commit message to all group coordinator

Step gc.2.2 once gc receives commit message from gl, gc broadcasts the commit message in its group.

Step ga.2.3 upon receiving commit message from gc, ga deliver transactions in I'

Broadcast phase:

Step gl.3.1 global leader gl sends proposal$< e', < v, z >>$ to all group coordinators in increasing order of zxid. epoch (z) = e'

Step gc.3.1 upon receipt of proposal$< e', < v, z >>$, group coordinator gc delivers group proposal$< e', < v, z >, groupid >$ to group acceptor along the tree

Step ga.3.1 when group acceptor gets group proposal, it accepts proposals, then acknowledges to group coordinator

Step gc.3.2 upon receipt acknowledgment from a quorum of group acceptors in its group, group coordinator append the particular proposal to its history, and acknowledges to global leader. Acknowledgment ACK-G contains group size and zxid

Step gl.3.2 once global leader gets acknowledgment from one group coordinator, it calculates sum of group size of ACK-G it have received for a given proposal. If the sum is greater than half of all members, global leader sends a commit COMMIT $(e', < v, z >)$ to all group coordinators

Step gc.3.3 once group coordinator receives COMMIT $(e', < v, z >)$ from global leader, if it has committed all transactions $< v', z' >$ such that $< v', z' >$ is in history and $z' < z$, group coordinator commits $< v, z >$, and broadcasts COMMIT-G $(e', < v, z >, groupid)$

Step ga.3.3 upon receipt of COMMIT-G $(e', < v, z >, groupid)$, if $< v, z >$ has not been committed yet, ga commits such transaction and delivers it along tree

IV. Performance Evaluation

We conducted a set of experiments. All experiments were run on a dedicated IBM Blade cluster in High Performance Computing Center at Huazhong University of Science and Technology. The IBM Blade cluster has 39 HS21 nodes; each HS21 node is equipped with a 2-way, 4-core Intel Xeon CPU E5345 running at 2.33GHz, and with 8GB memory and a 73GB SCSI hard disk. All the nodes are interconnected via a Gigabit Ethernet network and a 10Gbps Infiniband, and all nodes run the RedHat Enterprize Linux 5 (Linux 2.6.18-8.el5).

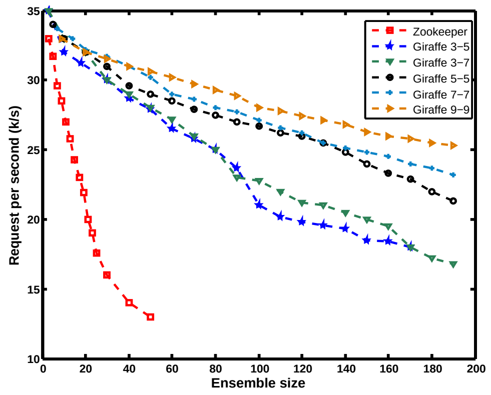

Fig. 6: Throughput of write operation on ZooKeeper and Giraffe.

For the existing related works, the ensemble of coordination service is unable to scale up to more than 10 servers. As studied in [11], when the ensemble is scaling, the bottleneck of central leader becomes a serious issue and update throughput drops down quickly. It would drag down the performance of application in write-sensitive scenario. So how to provide a large scale coordination service (such as more than 100 servers) is a challenge in distributed systems. The quality of coordination service is bounded with throughput, availability and latency. We divide the operations for coordination service into two types: read operation which only retrieves data from coordination service component, and update operation which modifies the data space. In both Giraffe and ZooKeeper, the throughput of read operations increases with the scaling of coordination service components as they need to read from local data space. In contrast, every update operation needs to be synchronized among service ensemble. It causes the decrease of write throughput with the scaling of service ensemble. Thus update operation is the key issue of throughput. Availability and latency are also the key characteristics for service .

A. *Throughput*

We conducted experiments in the server-client mode. We split servers into two parts: 4 HS21 nodes simulate 1000 clients and 19 HS21 nodes run up to 190 Giraffes varied from the number of Giraffe node. To compare the performance of Giraffe, we install ZooKeeper (3.3.6) in 10 HS21 nodes to run up to 50 ZooKeepers varied from the number of ZooKeeper node and 4 HS21 nodes simulate 1000 clients.

In the evaluation, client simulators generate the application request to Giraffe. The requests are data write operations which contains 1KB data. Different organizations of Giraffe cluster impact the throughput of system, so we set up a group of Giraffe nodes in different configurations of interior-node-disjoint trees. The configuration is formatted as t-c mode which is initialized in *Rooter*; t is the total number of trees while c is the maximum number of children of a node. In the evaluation, we prefer to build a forest in which the depth of each tree is not greater than 4. So typically with a t-c configuration the Giraffe

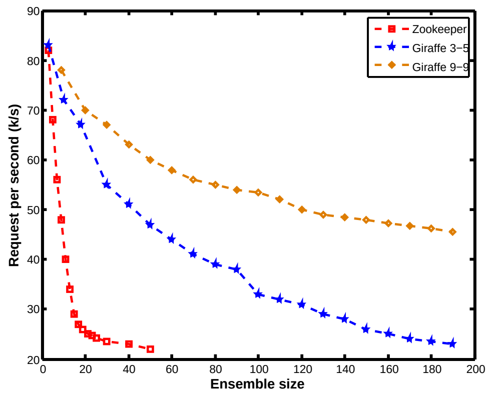

Fig. 7: Throughput of atomic broadcast component in isolation.

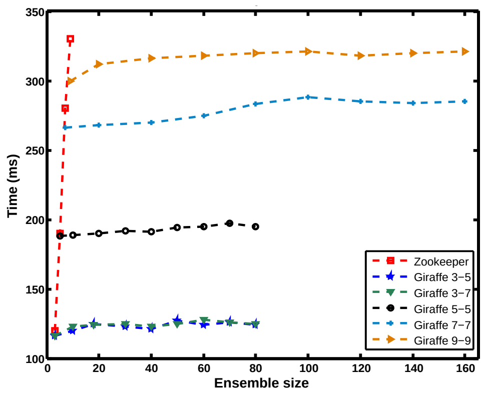

Fig. 8: Recovery of leader after failure injection.

cluster can hold up to $S_{max} = t * (1 + c + c * c)$ nodes. In this case, a cluster of 9-9 interior-node-disjoint trees can hold up to 819 nodes for 4-depth trees while a cluster of 3-5 trees hold up to 93 Giraffe nodes. We set up several t-c configurations to evaluate a saturated system on write operations.

As shown in Figure 6, Giraffe has a significant improvement on write operation over ZooKeeper. The experimental results show that the throughput of both ZooKeeper and Giraffe goes down while ensemble becomes larger, but ZooKeeper degrades more quickly than Giraffe especially in relatively large ensembles. When the ensemble size is 50 servers, the throughput of Giraffe is almost 300% as large as that of ZooKeeper. When the ensemble of Zookeeper servers is scaling, the central leader has to process more messages from its followers when write operations occur. And the limited process ability of the leader drags down the write throughput of the entire ensemble. Differently, Giraffe exploits group paxos to distribute the overload of global leader among group coordinators reducing the bottleneck of global leader. Also Giraffe introduces the hierarchical topology to reduce the dissemination load to its children. So Giraffe achieves better scalability and better write throughput than Zookeeper. Moreover, Giraffe with different initialised t-c performs differently in terms of throughput. With a greater t the slope of the line declines more gently. It is obvious as multiple groups running Paxos instances in parallel can actually accelerate the whole Paxos procedure. The slope of trend lines for 3-3 Giraffe and 3-5 Giraffe are almost the same. When the ensemble becomes larger than 100 servers, however, the throughput of 3-5 Giraffe is higher than that of 3-3 Giraffe. It is because the 3-3 Giraffe tree is much higher than that of 3-5 Giraffe in large ensemble, causing extra message delays and thus slowing down the write procedure.

Atomic broadcast protocol is the core of both Giraffe and ZooKeeper which limits the throughput of write operation. We benchmark the broadcast protocol of Giraffe and ZooKeeper by simulating client request directly to global leader of Giraffe and leader of ZooKeeper. In Figure 7, the throughput of atomic broadcast is much higher than 100% write in coordination service. It is obvious that client connections and request handles spend much CPU-bound of ZooKeeper and Giraffe. As shown in Figure 7, the atomic broadcast implemented in Giraffe generally outperforms that in ZooKeeper. Though the performance of Giraffe goes down as ZooKeeper does, the tendency is much gentler. The key reason is similar to that of write throughput. Such a performance gain is because of the distributed structure of the group paxos, so to reduce the cost of central leader, which is similar to that of write throughput.

In Giraffe, throughput of the topology with 9-9 configuration is more efficient than the topology with 3-5 configuration. It implies that group coordinator does more work in the atomic broadcast protocol. A Giraffe cluster has more groups in forest of 9-9 configuration than that of 3-3 configuration so that each group coordinator manages a relatively smaller group in 9-9 configuration than that in 3-3 configuration. However, it does not mean that more groups always have higher throughput. When the group number in the topology becomes larger, global leader has to manage more group coordinators, resulting in heavier overhead for global leader. When global leader becomes the bottleneck ahead of group coordinator, more groups may lead to poor performance.

B. *Availability*

High availability is one of the characteristics concerned mostly in coordination service. Figure 8 shows the recovery time of ZooKeeper and Giraffe after leader fails by failure injection. We find that the election time of global leader in Giraffe is stable even the ensemble becomes larger. However, the election time in ZooKeeper increases linearly with the increase of ensemble size. We can also see that in Giraffe, tree number is the essential configuration to influence the election time of global leader. The cluster with fewer trees takes less time to elect global leader, as shown in Figure 8. As reported in [11], leader is the major node for consistent update of the whole ZooKeeper cluster. Its crash leads to periodically unavailability and inconsistency of ZooKeeper ensemble. ZooKeeper uses the fast paxos to elect new leader from followers and synchronize the state of all followers when old leader has crashed. So when Zookeeper is scaling, the quorum of paxos becomes larger leading to long election time among Zookeeper followers. While in Giraffe, the crash of global leader will cause the update service unavailable and a new global leader will be elected from group coordinators. As the number of group coordinator is configured when system

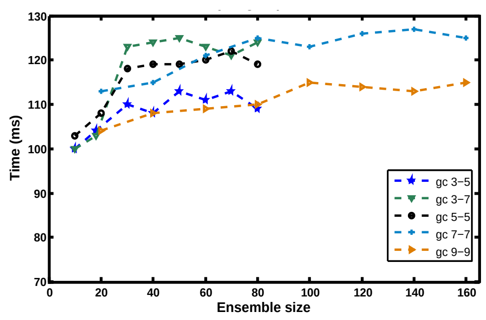

Fig. 9: Recovery of group coordinator after failure injection.

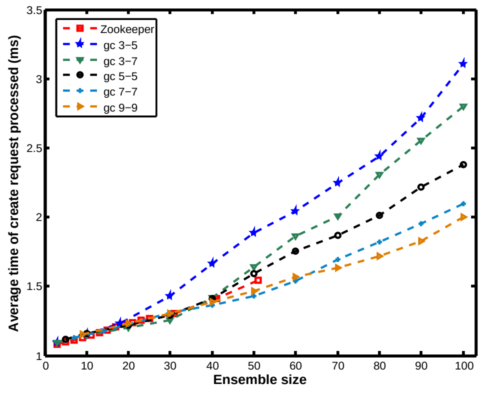

Fig. 10: Latency of create request.

is deployed, the election time of a new global leader is steady even when the Giraffe cluster is scaling.

Additionally, the crash of a group coordinator in Giraffe may lead to failure of Paxos procedure in a group and slow down the global Paxos commitment. Giraffe will rebuild the whole topology when the failure of a group coordinator is detected. Figure 9 shows the recovery time of group coordinator in different topologies with failure injection. It shows that the recovery time is almost between 100ms and 130ms, meaning the ensemble size does not affect the recovery time greatly.

C. Latency of requests

We follow the benchmark of ZooKeeper on latency of requests. We have started a worker to create 10,000 data nodes, each including 1KB data, on both Giraffe and ZooKeeper. The worker creates data nodes one at a time, waits for it to finish, sends an asynchronous delete to the node and then creates another one. We vary the number of ZooKeeper and Giraffe server when creating data nodes. We calculate the latency by dividing the total time by the number of create requests.

We show the request processing rate in Figure 10. When ensemble is small, the latency of Giraffe is similar to that of ZooKeeper. As the size of ensemble grows, the latency of both ZooKeeper and Giraffe becomes larger but acceptable. In some configurations such as 7-7 and 9-9, the latency of Giraffe is lower than ZooKeeper in moderate size of servers.

V. Related Work

As a typical protocol for asynchronous consistency, Paxos has many versions, e.g., classic Paxos [14], fast Paxos [7], Zab [13], and multi-coordinated Paxos [15]. In the classic Paxos, a coordinator plays an important role in voting. In a Paxos round, the coordinator sends phase1a and phase2a messages to a majority of acceptors. There are two issues to affect the whole Paxos consensus: 1) *quorum size*; 2) *coordinator failure*. As the quorum becomes large, the coordinator has to send more messages in a Paxos round that could be a heavy burden to the coordinator. Besides, once the coordinator is crashed, a new coordinator needs to be selected, typically from the coordinator quorum. In this period of time, Paxos becomes unavailable for proposers. In fast Paxos, fast rounds do not depend on a single coordinator, but do have stricter requirements on quorum sizes. Multi-coordinated Paxos extends classic Paxos to have multi-coordinators for higher reliability while maintaining latency and acceptor quorum requirements, but requires more messages and stricter conditions to determine a value. These Paxos variants do not deal with issue of acceptor size. *Group Paxos* has addressed these issues. Zab use a distinguished process (the leader) to guarantee the order of messages. Ring-Paxos [16] takes advantage of IP multicast and Ring topology for high performance. It executes consensus instances on ids which makes it efficient but only performs well in reliable network environments (LAN). In these leader-centric protocols, the leader does more work than other replicas, suffering from the limitation of scalability and performance.

S-Paxos[8] derives the fixed sequencer protocol. It distributes the work of traditional leader, including request reception, request dissemination and sending replies, across all replicas for high throughput. Mencius [17] and other moving sequencer protocols rotate the sequencer role among replicas. These protocol are efficient in WAN and failure-free environments. But the crash of any replica will stop the progress. EPaxos [18] is one paxos variant of virtual-synchronization version. It is efficient to reduce the latency of data replication in wide area. But it only performs well in small scale and does not satisfy the coordination service environment. Our *Group Paxos* in Giraffe distributes the load of leader by a hierarchical topology. With the disjoint property of topology, *Group Paxos* achieves more reliable Paxos instance even some replicas are failed. State partition is another way to solve leader-centric problem. [19] comes up with a highly available parallel B-tree service. Multi-Ring Paxos [20] addresses the problem of group communication protocols. It improves the scalability of group communication by partitioning the states in different rings. *Group Paxos* of Giraffe achieves scalability for full data replication. Even with failures, Giraffe can execute correctly.

Some existing systems have been implemented for coordination service. Chubby [10] raises the goals of high availability and strong consistency in distributed systems. It provides advisor lock primitive to implement complex distributed algorithms for the systems in Google. ZooKeeper [11] shares many design goals of Chubby. It outperforms Chubby on read by multiple servers providing service. Giraffe also exploits a hierarchical organization of data space, in-memory storage and file-system-like interface. And it combines blocking primitives and non-blocking primitives for various needs of scenarios.

Granola [21] implements a timestamp-based coordination mechanism to achieve order transaction and storage partition in several repositories. However, it requires fixed repository configuration and client participating to generate timestamps for transactional consistency. Giraffe exploits a different solution by implementing consensus protocol based on Paxos and trees topology. Scatter [22] implements a P2P storage layer targeting both consistency and scalability. Spanner [23] provides storage service for consistency replication across data centers and provides external-consistent distributed transactions. Differently, Giraffe stores configuration and other resources in memory. It implements a lighter and abstraction for transaction.

VI. Conclusion

Compared with ZooKeeper, Giraffe has higher performance in terms of scalability and strong consistency. Giraffe adopts a new scheme for membership management and a novel consensus protocol based on Paxos for consistent transactions. We evaluate the throughput, availability and latency of Giraffe and Zookeeper. Some key findings are listed below:

- Our Group Paxos helps to reduce the overload of centric leader and improve the scalability. The inner-group paxos which interleaves with global paxos accelerates the votes among large-scale acceptors. Giraffe is nearly 300% faster than ZooKeeper on update operations when ensemble size is with 50 nodes.
- Giraffe is more reliable than Zookeeper. Global leader of Giraffe recovers fast and steadily even when the group acceptors are scaling up.
- The depth of the forest effects the write throughput and the latency of commitment. In our evaluation, the deeper topology (e.g., t-c = 3-5, group number is 3 and max-children-number is 5) performs less efficient throughput and longer latency than others (e.g., t-c = 9-9).

Acknowledgments.: We would like to thank Franck Cappello, Qi Huang, Thomas Ropars, Chengwei Wang, and Xiaobo Zhou for their valuable Comments, as well as Iulian Moraru, Martin Biely for discussion on EPaxos and S-Paxos. This work is supported by the NSFC under grants No. 61370104 and No.61133008, National Science and Technology Pillar Program under grant 2012BAH14F02, MOE-Intel Special Research Fund of Information Technology under grant MOE-INTEL-2012-01, Chinese Universities Scientific Fund under grant No. 2014TS008, and the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] G. Fedak, H. He, and F. Cappello, "Bitdew: a programmable environment for large-scale data management and distribution," in *SC08*, Piscataway, NJ, USA, 2008.

[2] E. Okorafor and M. K. Patrick, "Availibility of jobtracker machine in hadoop/mapreduce zookeeper coordinated clusters," *Advanced Computing: An International Journal (ACIJ)*, vol. 3, no. 3, p. 19, 2012.

[3] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI*, 2010, pp. 251–264.

[4] 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, Jun. 2008.

[5] V. K. Vavilapalli, S. A. M. K. R. E. T. G. Arun C Murthy, Chris Douglas, S. S. Jason Lowe, Hitesh Shah, O. O. Bikas Saha, Carlo Curino, B. R. Sanjay Radia, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*, 2013.

[6] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC'07*. New York, NY, USA: ACM, 2007, pp. 398–407.

[7] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[8] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *SRDS*, 2012, pp. 111–120.

[9] K. Birman and R. Cooper, "The isis project: real experience with a fault tolerant programming system," in *ACM/SIGOPS European Workshop on Fault-Tolerance Techniques in Operating Systems*, ser. EW 4. New York, NY, USA: ACM, 1990, pp. 1–5.

[10] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI'06*. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX ATC'10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[12] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *NSDI*. USENIX Association, 2011.

[13] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *DSN*, 2011, pp. 245–256.

[14] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[15] L. J. Camargos, R. M. Schmidt, and F. Pedone, "Multicoordinated paxos," in *PODC '07*. New York, NY, USA: ACM, 2007, pp. 316–317.

[16] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *DSN*, 2010, pp. 527–536.

[17] Y. Mao, F. Junqueira, and K. Marzullo, *Mencius: Building efficient replicated state machines for WANs*. Department of Computer Science and Engineering, University of California, San Diego, 2008.

[18] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 358–372.

[19] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *DSN*. IEEE, 2011, pp. 454–465.

[20] ——, "Multi-ring paxos," in *DSN*. IEEE, 2012, pp. 1–12.

[21] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *USENIX ATC*. Boston, MA, USA: USENIX, Jun. 2012, pp. 21–21.

[22] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *SOSP '11*. New York, NY, USA: ACM, 2011, pp. 15–28.

[23] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *OSDI'12*. Berkeley, CA, USA: USENIX Association, 2012.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.