ZooKeeper+: The Optimization of Election Algorithm in Complex Network Circumstance

Xinyan Zhang, Zhipeng Tan(✉), Meng Li, Yingfei Zheng, and Wei Zhou

School of Computer Science and Technology,
Wuhan National Laboratory for Optoelectronics,
Huazhong University of Science and Technology, Wuhan 430074, China
xyzhangcs@gmail.com, zhipengtan@163.com

Abstract. Dynamic configuration management brings challenge for the distributed file systems while keeping the normal service. In this paper, we describe a robust election algorithm based on ZooKeeper, we realize the dynamic addition and deletion of servers without service interruption. There is only one clustered mode for servers without switching between prior two modes, it also speeds up the leader election. The leader maintains an active server list which speeds up handling of the transaction. The algorithm also ensures the data consistency and system stability against all possible issues. Through the evaluation, it takes not much overhead to realize the addition and deletion of servers and the recovery of crashed servers under various complex network circumstances, and it takes little more election time and initialization time of service to obtain the greater scalability.

Keywords: Election algorithm · ZooKeeper · Consistency · Scalability

1 Introduction

The clustered file system with multivariate metadata servers is an inevitable trend for the growing needs of performance, capacity and scale. It is critical for the clustered file system to guarantee the data consistency when providing normal service, it usually takes the coordination service to harmonize the nodes in cluster and ensures data consistency.

The difficult of designing distributed file system is handling the partial failure, it means the network may happen failure when message transfers from one node to another. The message sender cannot ensure whether the receiver has received the message or not. The receiver may receive the message before the network failure or not, and the receiver thread may have been broken down.

ZooKeeper [1] is an architecture to resolve this partial failure, it is an open source distributed coordination services. It is a centralized service that encapsulates public services, such as naming, configuration management, synchronization and cluster service into a simple interface. All of these kinds of services are widely used in many distributed applications such as Hadoop, KafkaMQ [2], Storm [3], etc.

X. Huang et al. (Eds.): GPC 2016, LNCS 9663, pp. 42–60, 2016.
DOI: 10.1007/978-3-319-39077-2_4

ZooKeeper keeps data consistency through a set of guarantees, such as sequential consistency, atomicity, single system image, reliability and timeliness. It provides transparent services for outside distributed system through a cluster that consists of $2n + 1$ servers who know the existence of each other. Its service mechanism is summarized as following.

(1) Servers in ZooKeeper can communicate with each other. They maintain the server's status, save the operations' logs and generate periodic snapshots in memory. The ZooKeeper can work only if more than half servers are correct (i.e., with $2n+1$ servers we can tolerate n but not $n+1$ failures). All proposals should be subject to approval by the leader, the followers then are informed of the changes by using Zab protocol [4].
(2) Client can choose anyone of cluster servers to read or write data. Operations will be synced to all servers, and each server maintains the memory state image, snapshots, and transaction logs of persistent storage. Once connected to a server, client maintains a TCP connection to submit requests, get response, get monitoring events and send heartbeats. If the TCP connection gets broken, client will connect to another server to ensure service continuity.

This paper designs a robust leader election algorithm that can adapt to arbitrary topological changes in complex environments such as network partitions, servers crashed, etc. There is only one clustered mode for servers.

The rest of this paper is organized as follows. Section 2 presents the related works. Section 3 describes the robust consistent election protocol. Section 4 analyzes its data consistency guarantee under various complex situations and then proves correctness of the robust algorithm. In Sect. 5, we demonstrate the effectiveness of robust election algorithm versus Zookeeper through experiments. And the last section concludes the paper.

2 Related Works

For ensuring the high availability of ZooKeeper service, it needs to do the multiple redundant backups, the write operations to these multiple backups bring the consistent problem. It is critical to guarantee the data consistency between these redundant backups.

Eric Brewer proposed the CAP theorem [5,6] that there is a fundamental trade-off between consistency, availability, and partition tolerance when designing the distributed file system. According to the degree of consistency, the paper [7] divided it into three parts: strong consistency, weak consistency and eventual consistency. The consensus problem existed in the multiple backups can be dived into two parts: the consistency of any updates and multiple updates.

The cluster needs to reach a consensus by leader election so as to coordinate the servers. The existing studies of the leader election algorithm have proved that it cannot keep the consistency about the leader election without any assumption on the premise of the network environment [8]. Considering the speed of

message delivery during the network communication, the current leader election algorithms can be divided into the following three sorts.

(1) The election algorithm of strong consistency. These election algorithms aim at the consistency of any updates. The election algorithm proposed in paper [9] ensures the voters can finish the updaing operation in a limited time. The paper [10] uses heap tree method to elect the leader with the lesser complexity. The Raft election algorithm [11] and the Ark algorithm [12] both guarantee the consistency of any updates.
(2) The election algorithm of weak consistency. These algorithms improve the high concurrency of single update and guarantee the consistency of multiple updates. The algorithm in paper [13] allows the network partition has its own leader when network partition occurs, it will elect the unique leader finally after system merges these partitions. The algorithm in paper [14] based on TORA guarantees the system can elect the unique leader finally after the partitions restore the connection. The algorithm in paper [15] tolerates frequent topology changes and finally elects the unique leader, but it cannot guarantee a unique leader at all times.
(3) The election algorithm of eventual consistency. It only needs to ensure the updates keep consistent finally. The paper [16] gives a complete network with N nodes, it tolerates up to $N/2-1$ links incident failure on each node, so the system tolerates up to $N^2/4 - N/2$ links failure. The algorithm in paper [17] eventually uses only n links to carry message and it ensures that a leader is elected in constant time when the system is stable. The Fast Leader Election algorithm [18] that used in ZooKeeper architecture set a half bound variable-*quorum*, only the number of active nodes are more than quorum can the network partition elects the leader.

The reconfiguration protocol for Primary/Backup replication systems exploits primary order [19], it needs to persist information about new configuration S' on stable storage at a quorum of current system S, deactivate the current configuration, then identify and transfer all committed state from S to S'. As this algorithm needs to write new configuration S' to a quorum of S at least, a quorum of S should execute these operations at least, it has low efficiency.

We analyze the Release_3.4.5 of ZooKeeper and we find it simplifies the configuration management strategy by using a static configuration to deploy both clients and servers. It needs to restart cluster multiple times to reconfigure the system, this leads to service interruptions unavoidably. Its scalability is limited and its weakness can be summarized as follows.

(1) Zookeeper uses two deployment modes: standalone mode and distributed mode. The standalone mode is usually used for test or demos. It is not flexible to switch between two modes.
(2) It limits the number of servers in cluster, only more than three servers can form the cluster to provide the service outside.
(3) It cannot add or delete any servers dynamically during runtime of the cluster, the multiple servers need to be shut down and restarted if it needs to reconfigure the cluster.

(4) The leader sends commands and broadcasts to the ensemble when handling transactions, the messages send to the crashed servers are unnecessary.

Based on the above weakness of ZooKeeper, we propose a robust consistent election algorithm with the following contributions.

(1) It takes the unique clustered mode to replace the original standalone and distributed modes. It avoids the overhead brought by the conversion between the prior two modes.
(2) The cluster scale is unrestricted, one or two servers can also elect the leader through election algorithm. It can speed up the leader election through configuring a single participant and multiple observes at startup.
(3) It realized the dynamic server addition and deletion when the cluster provides the normal service. It reduces the service interruptions brought by the restart operations, it also reduces the redundancy of the persistent server list.
(4) The proposals and commands are only send to the servers in the active server list which is maintained by the leader. It improves the efficiency by reducing the overhead that is send to the inactive servers.

For the complex environment that may have the split brain, data corruption and crashed servers, the algorithm guarantees the data consistency during the proceeding of server addition, deletion and recovery, and it improves the efficiency of handling transactions. The algorithm obtains better performance and gains greater availability and scalability.

3 The Consistent Election Algorithm

Specific implementation details are given in this section, such as the design of start mode, server addition and deletion, and how to ensure the consistency of the data during the ensemble dynamic changing procedure.

(1) The cluster's initial startup
  (1-1) If the administrator needs to start up n servers with initial mode, he should configure n servers' information in each server' persistent list so that all servers can keep consistent at startup.
  (1-2) Each server initializes its persistent server list version and zxid to 0, and sets quorum to $\lfloor n/2 \rfloor$.
  (1-3) Each server starts with the initial mode, and executes the leader election step 2.
(2) Leader election
  (2-1) Set each participant's status as *Looking*.
  (2-2) Each participant sends votes to those participants in its persistent server list one by one.

(2-3) It judges whether its own *version* of the persistent server list is bigger than the submitter's *version* and is the maximum among the known *version* when a participant receives one vote message. It will put self's server list, *version* and *zxid* into the next vote if the judgments above are true. The server will update self's persistent server list and restart a new vote according to the updated server list if self's *version* of the persistent server list is smaller than the *version* of the received votes and these received votes contain the information about the persistent server list.

(2-4) It judges whether more than $\lfloor n/2 \rfloor$ participant's *version* are the biggest and the same once it finishes traversing the n participant of the latest persistent server list. Execute the next step if it meets the conditions, otherwise jump to step (2-2).

(2-5) Every participant votes the server whose *zxid* is the biggest among all survived participants as the leader. It chooses the server with biggest ID number as the leader if all the *zxid* are the same. Then leader's epoch plus 1 and updates the quorum to the $\lfloor n/2 \rfloor$. The leader election finishes and then turns to the registration procedure (3).

(3) Server registration

(3-1) Learner sends registration request that includes information about the *zxid* and *version* to leader.

(3-2) Leader receives the registration request from learner and replies its own *zxid* to the learner.

(3-3) Learner replies the data synchronization type including OK, $SNAP$, $TRUNC$, and $DIFF$ to leader after comparing its own *zxid* with leader's *zxid*.

(3-4) Leader sends the necessary *UpToDate* data according to the data synchronization and judges whether its own persistent server list to be sent according to learner's version information.

(3-5) Learner updates the *UpToDate* data, persistent servers' list, *version*, and *zxid*, then it returns an *ack* back to leader, and finally turns to heartbeat procedure (4).

(3-6) After receiving more than quorum participants' acks, Leader generates active server list and then turns to heartbeat procedure (4).

(3-7) When a learner recovers from failure, it executes leader election procedure (2) first, then sends registration request to leader, who will add the learner to its active server list after leader and learner execute the procedures from (3-1) to (3-5), and returns a registered success message to the learner, thus finishing the crashed servers' recovering procedure.

(4) Server heartbeat

(4-1) Leader sends ping message to all learners in the active server list every tickTime, it will delete the learner from the active list if no *ack* returns of successive three times. Once it finds that the quantity of participants in the active server list is less than the quorum, it will turn to the leader election procedure (2).

(4-2) Learner detects the ping message from leader and returns *ack*. If it hasnot received leader's ping message in three $tickTimes$, it will turn to the leader election procedure (2).

(4-3) Particularly, leader sends the detecting message which comes with its own server information (ID number, IP address, client port, role) to all inactive learners every two T times. When inactive learner recovers from the crash, it begins to register to leader once it received the detecting message from leader. This procedure guarantees special crashed servers' recovery, such as server that stays in failed section for a long time and other servers in its own server list have all been deleted.

(5) Data synchronization

Just like the original ZooKeeper system, learner sends request to leader, leader uses two-phase protocol which is similar to the Paxos algorithm. The adding and deleting operation of our algorithm do not impact on the data synchronization. It should be noted that because of the active server list is to be used once leader launches data synchronization proposal, so coupling between data synchronization and servers' addition and deletion is the change of the active server list. The new active server list can be used for synchronization at the next time to commit the proposal after changing active server list.

(6) Server addition

(6-1) The administrator needs to configure a latest persistent server list for NS (new server) when a new server wants to add into the cluster.

(6-2) The NS starts up with the adding mode.

(6-3) NS sends the participants in its persistent server list the message about whether it exists in the others' persistence server list and which server is the leader successively. It will execute the leader election procedure (2-2) if more than quorum of the participants returns the certain answer, otherwise turn to (6-4) if it just know the leader information.

(6-4) NS sends an add request to the leader.

(6-5) Leader broadcasts the proposal about adding a server to the active participant after receiving the add request from the NS, each participant returns an *ack* after receiving the proposal broadcast.

(6-6) If the leader receives more than quorum participants' *ack*, it will update itself persistent server list and plus 1 to its $version$, then sends performing operations to all the active participant.

(6-7) Participants update their persistent server list and version according to the information of NS in the proposal after receiving the commit message, then each returns an *ack*.

(6-8) The leader sends a successful message about addition to the NS after receiving more than quorum participants' acks.

(6-9) NS executes the registering procedure (3-7) after receiving the successful message about addition from the leader, or restarts executing the procedure (6-3).

(6-10) Leader updates the quorum number to the $\lfloor n/2+1 \rfloor$ according to the latest participants' number after receiving the NS's registering message at procedure (3-7).

(7) Server deletion

(7-1) Administrator just needs to call the deleting interface from any server (operation server, hereinafter referred to as the OS) in the cluster when he wants to delete one server from the current system.

(7-2) OS sends request about deleting DS (the server to be deleted) to the leader.

(7-3) Leader checks whether DS existed in its server list after receiving the deleting request, it returns 'X may be not existing' if DS does not exist in its server list, otherwise sends message of stopping service of DS, removes the DS from its own server list and updates quorum number to $\lfloor n/2\text{-}1 \rfloor$ and version. If the DS is an active server, it will shut down itself directly after receiving the message of stopping service from the leader, it will also turn to election procedure if it has not received the ping message from leader, then shut down itself through knowing itself do not exist in the current system from other servers' feedback; if DS is a crashed server, leader will delete it from its own persistent server list and stop send detecting message, which in order to prevent the crashed server recovering and adding in the cluster.

(7-4) Leader broadcasts the proposal to active participants, participants delete the DS and update its persistent server list, and then each returns an ack to leader.

(7-5) Leader sends message about deleting successfully to the OS if it receives more than quorum participants' acks.

(7-6) The delete operation is completed after the OS receiving the successful message about deletion from leader, otherwise OS restarts executing the procedure (7-2) if it receives the message about deleting failed or waits for a timeout.

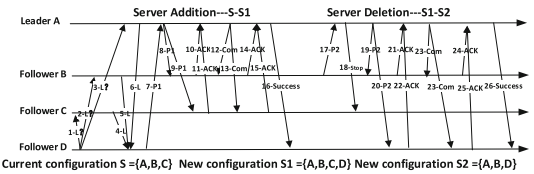

Fig. 1. Reconfiguration with server addition and deletion

Here is an example about the reconfiguration with the server addition and deletion which is shown in the Fig. 1. The current configuration S contains server A, B and C, A is the leader, B and C are followers.

A new server D begins to add into the cluster at a time and its persistent server list is $[A, B, C, D]$, it firstly sends messages whether it exists in the current

cluster and the leader information to the server A, B and C, all of them returns leader information. Then it sends adding request to Leader A, Leader A broadcasts the proposal about adding D to the active server B and C, B and C return ACK back to leader A, Leader A updates its persistent list to $[A, B, C, D]$ after receiving more than quorum ACK, and sends committing message to server B and C, the server B and C updates to the latest persistent list $[A, B, C, D]$ and return ACK back, the leader A sends the addition success message to server D after receiving more than quorum ACK, server D turns to registers to leader A and leader A updates its quorum number from 2 to 3. Server D successfully adds into cluster.

The leader A sends message about stopping service to server C after receiving request to delete server C from server B. Server C shuts down itself directly after receiving the command to stop service from the leader A. Leader A removes server C from its own persistent list, updates quorum from 3 to 2 and changes the version, then broadcasts proposal about deleting server C to the active server B and D. Server B and D keep it in the local disk by the transaction log mode and return an ACK back after receiving the proposal broadcast, the leader A receives the ACK messages from server B and D, then sends commit commands to server B and D. The server B and D update their persistent list to $[A, B, D]$ after receiving the commit commands and plus 1 to their version, then return an ACK back. The leader A sends deletion successful message to server B after it receives more than quorum ACK.

4 Algorithm Analysis and Proof

4.1 Guarantee of Consistency

The algorithm in this paper will modify and synchronize the persistent server list when add and delete servers. The consistency of persistent server list is affected by the complex network circumstance (message loss, server crashes, and network partitions). We should know how consistency problem of the persistent server list happened before proving the correctness of algorithm, because it directly influences the motive and principle of the algorithm.

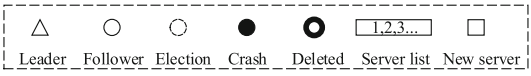

Fig. 2. The custom icons

The consistency problem that may occur during the execution of the algorithm, it can be boiled down to the following two categories:

(1) As servers in normal partition may enter into crashed partition at any times and servers in crashed partition cannot contact with the servers in normal

partition, the crashed partition cannot check the change of persistence server list in normal section, which leads to the disaccord between the crashed partition and normal partition.

(2) In normal partition, if the message losses or the leader exits during the process of synchronization about persistent server list when the leader executes the adding or deleting operations, the normal partition's inner persistent server list will not keep consistency.

There are three extreme cases of the consistency problem in the running process of cluster. First, we give the custom icons used in following steps in Fig. 2.

Consistency Guarantee in Server Addition. The following Fig. 3 shows how it keeps consistency of server list in server addition step when the leader goes down.

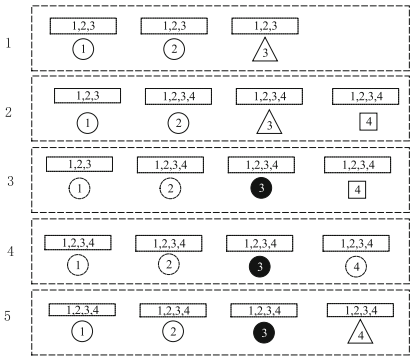

Fig. 3. Consistency guarantee in server addition

Step 1: There are three servers at startup of cluster, server 1, 2 are followers and server 3 is leader. All the persistent server lists are [1, 2, 3].

Step 2: Server 4 begins to add into cluster, it polls its persistent server list to look for the leader, then it sends adding request to server 3, server 3 updates its server list to [1, 2, 3, 4] and executes two-phase commit protocol after receiving adding request from 4. Server 2 updates its server list to the latest while server 1 does not update for some reasons such as unreliable network or dropped packets, etc.

Step 3: Leader server 3 crashes before it returns successful message back to server 4, server 1 and 2 restart leader election.

Step 4: For server 4, it waits for timeout and restarts to add into cluster, but it finds cluster is electing the leader, the server 1 and 2 have updated their server list to the latest [1, 2, 3, 4], server 4 joins into leader election.

Step 5: Server 4 is elected as the leader because of the biggest *zxid*, server 4 adds successfully.

Consistency Guarantee in Server Deletion. The following Fig. 4 expounds how it keeps consistency of server list in servers' deletion when leader fails.

Step 1: There are 3 servers in cluster, and 3 is leader, now server 4 adds into the cluster successfully.
Step 2: Leader 3 receives the request of deleting server 4, and sends message of stopping service to server 4. The message loses because of network malfunction or dropped packets, but server 3 updates its server list to [1, 2, 3].
Step 3: Leader server 3 sends deletion proposal to active servers, only server 2 responses and updates its server list to [1, 2, 3]. Server 4 restarts to leader election as losing heartbeat from leader 3.
Step 4: If server 3 crashes now, server 1, 2 will restart to leader election.
Step 5: Server 1, 2, 4 send votes to each other and update to the latest server list ([1, 2, 3]), server 4 finds itself is not in the latest server list and will shut down. Server 2 becomes the leader because of the bigger zxid among the active servers.

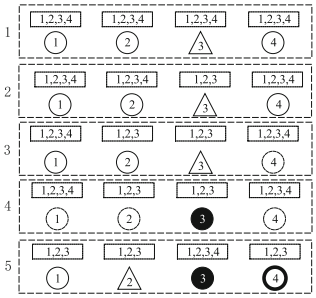

Fig. 4. Consistency guarantee in server addition

Consistency Guarantee in Server Recovery. The following Fig. 5 exposits how it keeps consistency of server list in servers' recovery when leader goes down.

Step 1: there are three servers in the cluster, server 1 was isolated due to network partition and it is in state of leader election, server 3 is elected as the leader.
Step 2: server 4, 5, 6 add into cluster successfully some time later.
Step 3: server 3 failed after deleting server 2 successfully, the rest servers restart to leader election.
Step 4: server 6 is elected as the leader, then cluster deletes server 3 successfully.
Step 5: server 1 recoveries from the crash as network partition disappears, it sends votes to server 2 and 3 but gets no reply. Server 1 receives the exploring message from leader 6, gets the latest server list and registers

to server 6, and then it adds into cluster successfully. (For crashed server, leader sends exploring message to them every 3 T times for detecting their recovery).

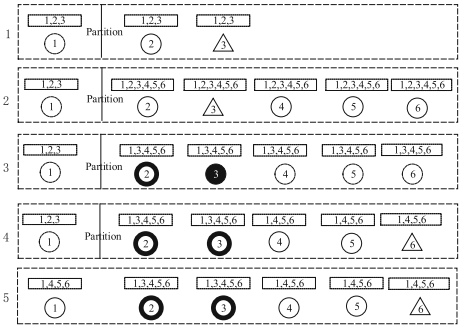

Fig. 5. Consistency guarantee in server addition

4.2 Algorithm Proof

The algorithm in this paper supports adding and deleting server dynamically when the system is running, and ensures the correct operation of the system under the complex network circumstance (message loss, server crashes, network partitions) at the same time. The algorithm meets the following two characters:

Property 1. For any two servers (A, B) of the cluster, if their versions of persistent list meet $Version(A) > Version(B)$, then the persistent server list of A must contain server B.

Illustration: Property 1 makes sure that any two participants can exchange their persistent server list according the election step (2-2) -(2-4) when partition contains any number of servers. So it will not happen that servers cannot communicate with each other because of the inconsistency of the persistent list, thus providing the safeguard for subsequent algorithm.

Proof.

(1) In procedure (1-1), all servers' persistent list keep consistency at startup, the version is 0, and each server's persistent list concludes all servers' information, so Property 1 is corrected.
(2) In procedure (6-1), before adding a new server, the new server should know all servers' information and obtain the biggest version of persistent server list from the leader after the success of the addition, thus Property 1 is corrected.

(3) In procedure (7), the service of the DS needs to be stopped first, leader then deletes it from its persistent list. It will not recover for a deleted server, so it will never happen that the deleted server is still working but not existing in other servers' persistent list, thus proving the correctness of the Property 1. □

Property 2. Let n denotes the number of participants in the system (including active and crashed servers), then the system will elect a unique leader eventually and provide the service in a network partition that its numbers of active participant are not less than $\lfloor n/2+1 \rfloor$.

Illustration: Property 2 guarantees that the algorithm can elect a unique leader to provide service when more than quorum of participants are active and they can communicate with each other inner one partition, so the consistency problem caused by multiple leaders that run concurrently.

Proof.

(1) The persistent list of all servers is consistent at the startup, so it can ensure the correctness of Property 2 by the quorum principle in the electing procedure (2).
(2) Only more than quorum of participants' persistent list keep consensus can the leader provide service in procedure (3).
(3) Only leader ensures that more than quorum of participants' persistent list keep consensus can it returns the successful message about server addition or deletion in procedure (6) or (7) of algorithm steps.
(4) There are more than quorum of participants inner current system have updated to the latest persistent list when the system returns the successful message about adding or deleting server, these participants make up set A, and because more than quorum of participants in current system are survival, these servers make up set B, so set A and set B have an intersection and set B must have some servers that own the latest persistent list, then participants inner set B will finally update to the latest persistent list according to procedure (2), this guarantees the correctness of the Property 2.
(5) There are two cases when system returns failed message about adding or deleting server:
    (5-1) Only less than quorum of ensemble inner system update to the latest persistent list;
    (5-2) There are more than quorum of participants updating to the latest persistent list, but the successful messages about adding or deleting server are lost in the procedure (6-8) or (7-5) of algorithm steps. The servers submitted request wait to time out.

So it can be concluded from the executor: successful addition or deletion is certain, but failed addition or deletion maybe not. If one sends message to the other when any two servers communicate with each other inner unreliable network circumstance, only one receives the reply message can we make sure the other have received the sending message, otherwise it is not sure whether the other receives the sending message.

Starting from this logic, when the system returns the failed message about server addition or deletion, the algorithm will retry constantly and not execute the next addition or deletion operations until it succeeds. This can finally make up system's consensus problem and provide guaranty for the Property 1 and 2. From the sight of the system, the system is problematic if it returns failed message about server addition or deletion, it makes no sense to execute the follow-up operation of addition and deletion. □

Property 3. Let n denotes the number of participants in the current system (including active and crashed servers), it will not elect a leader inner one partition that its active participants are less than the $\lfloor n/2+1 \rfloor$.

Illustration: The Property 3 avoids more than one leader appearing in the system, it will not elect a leader when less than quorum of participants constitutes a partition because of the inconsistent problem of persistent list (It may reduce the quorum requirements because the server list in this partition may be old and participants in its persistent list are limited).

Proof.

(1) There is a set A consists of m active participants in a partition that has less than quorum of participants, $2 * m <= n$;
(2) Let a denotes the participant whose version of persistent list is the biggest in set A.
(3) According to the electing procedure (2), the persistent server list of active participants in set A will keep consistent with server a.
(4) By reduction to absurdity, assumes that the Property 3 is false: set A can elect a leader, then if persistent list of server a contains k servers, it must satisfy this conditions: $k < 2 * m$.
(5) Because $k < 2 * m$ and $2 * m <= n$, so $k < n$, which means that server a malfunctions at some point during the system running and it have not updated to the latest version that contains n servers' information.
(6) According to the Property 1, server a has the biggest version in the set A, so its persistent list inevitably contains all participants of set A, this means set A has m participants at least.
(7) There must contain serval servers' addition and deletion from the former system that it contains k participants when server a fails to the current system contained n participants, and only more than quorum of participants' update to the latest persistent list successfully can the leader returns the successful message. Let set B denotes the quorum of participants before server a fails, so only make sure more than servers in set B are survival at least before server a malfunctions can we ensure the successful execution of the follow-up servers' addition or deletion.
(8) As participants in set B execute servers' addition or deletion successfully after server a fails, so the versions of all the participants' persistent list in set B are certain bigger than the versions in set A. So the set A and set B must be no intersection.

(9) Combined with the above procedures (6), (7), (8), we have the following conclusions:
  (9-1) Set A has m participants at least.
  (9-2) Set B should have more participants than set A, so set B has $m + 1$ participants at least.
  (9-3) The persistent server list of server a must contains set A and B, and set A and B must be no intersection.

It can conclude that $k \geq 2 * m + 1$, while this is the contradiction with the assumption about $k < 2m$ in procedure (4). Thus proving the correctness of Property 3. □

5 Evaluation

In this part, we mainly evaluate the robust election algorithm supporting dynamical reconfiguration from the following two parts: function test and performance. We used 9 servers to build a ZooKeeper cluster for the evaluation, each server has one Inter Xeon dualcore 2.4 GHz processor, 4 GB of RAM, 1 Gigabit Ethernet, and two SATA drives. The operating system is Red Hat Enterprise Linux Server release 5.4 (Tikanga), Linux Kernel version is 2.6.18 and the java version is 1.7.0_51.

5.1 Function Test

The function test aims at ensuring the cluster can work normally under various complex situations, we devided it into three parts: test for server addition, deletion and recovery. The cluster scale consists of three cases: single participant (P), two participants (P_2, PO) and multiple followers and observers ($P_m O_n$). The moment for the server addition, deletion and recovery should consider the current cluster status (normal, electing, crashed). The operated server status includes follower and observer.

It needs to be considered for server addition and deletion is whether servers can be successfully added or deleted or not when the crashed servers happen to recover from the crash at the moment of server addition and deletion. And for the sever recovery, after multiple servers' addition and deletion, some crashed server may no longer have any current cluster's configuration information, but more than quorum active servers still keep the crashed server's information, the crashed server can add into the cluster by leader's exploring message when it is recovered from crash. In an extreme case, the persistent server list of crashed server contains $[1, \ldots m]$ while the current cluster is $[m, \ldots n]$. The crashed server is m. It can register to the leader and successfully add into the cluster after it recovers from the crash and receives the leader's exploring message.

SID	ZooKeeper+			ZooKeeper		
	num	>10	tim	num	>10	tim
65	230469	9	0. 256044	236453	8	0. 249761
66	209580	9	0. 282553	235310	8	0. 250865
67	leader			leader		
69	216969	9	0. 272927	233788	7	0. 252721
70	153491	9	0. 386894	142352	8	0. 417019
71	195853	8	0. 302525	177402	7	0. 333836
72	164944	10	0. 359944	172545	8	0. 343431
73	216253	11	0. 273754	213678	8	0. 276487
sum	1387559	65	414650. 2633	1411528	54	414094. 0719
ave		4. 7E-05	0. 298834329		3. 8E-05	0. 293365822

Fig. 6. The comparison on operations

5.2 Performance Test

We used the ZooKeeper Benchmark tool provided by the Computer Science department of Brown University which measures the perrequest latency of a ZooKeeper ensemble for a predetermined length of time [20] to the throughput.

The benchmark exercises the ensemble performance by handling znode reads, repeating writes to a single znode, znode creation, repeating writes to multiple znodes, and znode deletion. These tests can be performed with either synchronous or asynchronous operations. The benchmark connects to each server in the ZooKeeper ensemble using one thread per server. In synchronous operation, each client makes a new request upon receiving the result of the previous one. In asynchronous operation, each client thread has a globally configurable target for the number of outstanding asynchronous requests. If the number of outstanding requests falls below a configurable lower bound, then new asynchronous requests are made to return to the target level.

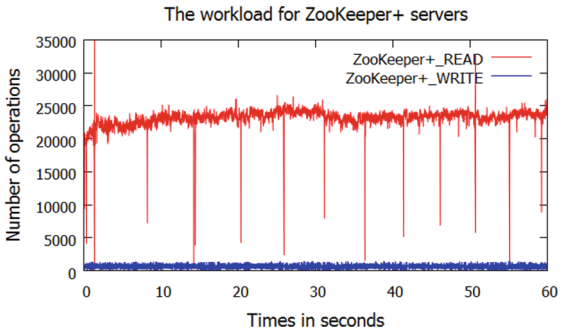

Fig. 7. Workload for ZK servers with znode read and creation

We use 9 machines to evaluate its performance, the server ID is from 65 to 73, 68 is used for running benchmark, we configure the cluster with three participants, five observers and one client server. The server 65, 66, 67 are configured as the participants, the rest are set as observers. We named the robust

election algorithm ZooKeeper+ to facilitate the comparison test. We use the Release_3.4.5 of ZooKeeper as the comparison. We donot run benchmark on the leader. During the benchmark, the current rate of request processing is recorded at a configurable intermediate interval to one file per operation: READ.dat, SETSINGLE.dat, CREATE.dat, SETMULTI.dat, and DELETE.dat. the operation time is 600 s, and the request frequency is 800,000 requests per second for each client. We analyze the *.dat and count the number of operations per node. Meanwhile, we analyze the ZooKeeper.out under the bin directory of each node and calculate the election time and initialization time. We mainly make the comparison about znode reads and creation. We run each cluster for 10 min, and make the comparison between ZooKeeper and our robust algorithm.

Fig. 8. Comparison between two versions on znode creation

The above Fig. 6 shows comparison on operations in *.dat, num represents the total number of operations, >10 represents the number of outliers that are more than 0.01, and tim represents the arithmetic average of the time. It can be not hard to see the operations in ten minutes between two versions do not have much difference, and the number of outliers are nearly the same compared with the huge number of operands. Meanwhile, there is little variation among these two version on the numbers of total operations.

The Fig. 7 shows the read and write traffic for the cluster during one hour, each point corresponds to the number of operations in that second. We observe that the read traffic is much higher compared to the write traffic, that because the read operations in this workload are getData(), getchildren(), and exists(), in increasing order of prevalence.

The Figs. 8 and 9 show the comparison between ZooKeeper+ and ZooKeeper on znode writes and reads during the cluster working normally in one hour, the ensemble operations are all in the same range, it is nearly the same with the ZooKeeper, which means ZooKeeper+ works stably as well as the ZooKeeper.

As shown in Fig. 10, it provides the leader election time and initialization time of two versions. The election and initialization time of ZooKeeper is little less than the ZooKeeper+, the reasons are as follows. Firstly, server needs to

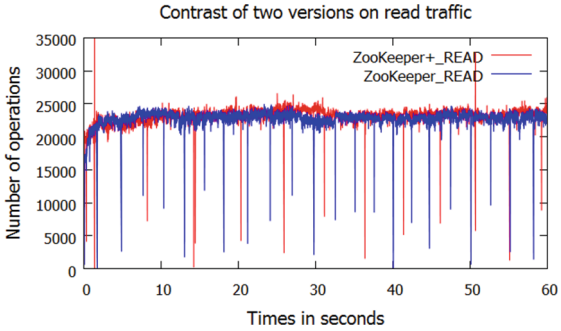

Fig. 9. Comparison between two versions on znode read

choose the different startup mode according the variable startType, and leader needs to monitor the adding request from the new server. In addition, servers need to judge value of the vote information of persistent server list contained in the votes, they may need to update the persistent server list if its vision is old. But the above all realize the dynamical server addition and deletion without restarting the cluster and causing the current service interruption, it gains the greater scalability.

This section mainly evaluates the robust algorithm from the function and performance parts. Through the function test, it concludes that the robust consistent election algorithm greatly realizes the expansion of cluster from the single node to the multiple nodes. It can add new server and delete the servers successfully, and it guarantees the recovery of the crashed servers, ensures the availability and improves the scalability of the cluster. Through the performance test, compared with the ZooKeeper, the number of operations of two versions in the same time have no difference nearly, the traffic of the reading and writing operations are essentially flat, the election and initialization time is little more than ZooKeeper, but it is acceptable as it gains greater scalability.

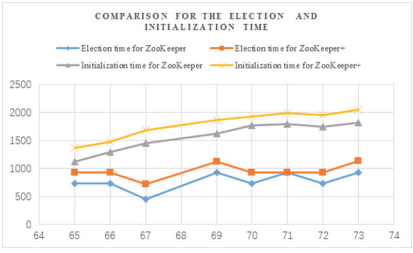

Fig. 10. Comparison for the election and initialization time

6 Conclusion

This paper is focus on the leader election problem in distributed file systems. Based on the architecture of ZooKeeper, a robust election algorithm supporting dynamic reconfiguration is proposed. The original standalone mode and distributed mode are replaced by the unique cluster mode without switching between prior two modes. The cluster scale is unrestricted, one or two servers can also elect the leader, it can speed up the leader election through configuring a single participant and multiple observers at startup. It realized the dynamic server addition and deletion without the service interruptions. It speeds up handling of the transaction as the release of proposals and commands is only send to the servers in the active server list which maintained by the leader. Then the robust algorithm and traditional ZooKeeper are fully implemented and tested under different circumstances. Test results demonstrate that the consistent election algorithm supporting dynamical reconfiguration has the same performance with the traditional model and gains greater availability and scalability. In conclusion, the robust algorithm is meaningful and it makes ZooKeeper more practicable and flexible.

Acknowledgments. This work is supported by 973 project 2011CB302301, the National Basic Research 973 Program of China under Grant by National University's Special Research Fee (2015XJGH010).

References

1. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: waitfree coordination for internet-scale systems. In: USENIX Annual Technical Conference, vol. 8, p. 9 (2010)
2. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: NetDB (2011)
3. Ranjan, R.: Streaming big data processing in datacenter clouds. IEEE Cloud Comput. **1**, 78–83 (2014)
4. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), pp. 245–256. IEEE (2011)
5. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News **33**(2), 51–59 (2002)
6. Gilbert, S., Lynch, N.A.: Perspectives on the CAP theorem. Institute of Electrical and Electronics Engineers (2012)
7. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
8. Garcia-Molina, H.: Elections in a distributed computing system. IEEE Trans. Comput. **100**(1), 48–59 (1982)
9. Huang, S.T.: Leader election in uniform rings. ACM Trans. Program. Lang. Syst. (TOPLAS) **15**(3), 563–573 (1993)
10. EffatParvar, M., Yazdani, N., EffatParvar, M., Dadlani, A., Khonsari, A.: Improved algorithms for leader election in distributed systems. In: 2010 2nd International Conference on Computer Engineering and Technology (ICCET), vol. 2, pp. V2–6. IEEE (2010)

11. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp. 305–319 (2014)
12. Kasheff, Z., Walsh, L.: Ark: a real-world consensus implementation. arXiv preprint arXiv:1407.4765 (2014)
13. Fetzer, C., Cristian, F.: A highly available local leader election service. IEEE Trans. Softw. Eng. **25**(5), 603 (1999)
14. Malpani, N., Welch, J.L., Vaidya, N.: Leader election algorithms for mobile ad hoc networks. In: Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, pp. 96–103. ACM (2000)
15. Vasudevan, S., Kurose, J., Towsley, D.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: Proceedings of the 12th IEEE International Conference on Network Protocols, 2004, ICNP 2004, pp. 350–360. IEEE (2004)
16. Singh, G.: Leader election in the presence of link failures. IEEE Trans. Parallel Distrib. Syst. **3**, 231–236 (1996)
17. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
18. Lamport, L.: Fast paxos. Distrib. Comput. **19**(2), 79–103 (2006)
19. Shraer, A., Reed, B., Malkhi, D., Junqueira, F.P.: Dynamic reconfiguration of primary/backup clusters. In: Presented as Part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 425–437 (2012)
20. Ferguson, A., Liang, C.: ZooKeeper-benchmark (2012). https://github.com/brownsys/zookeeper-benchmark