STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVI**, Number 1, 2021
DOI: 10.24193/subbi.2021.1.07

LEADER ELECTION IN A CLUSTER USING ZOOKEEPER

MANUELA PETRESCU

Abstract. This paper presents an algorithm for flexible and fast leader election in distributed systems using Apache Zookeeper for configuration management.

The algorithm proposed in this paper is designed for applications that do not use symmetric nodes so they need a specialized election process or for applications that require a more flexible approach in the leader election process. The algorithm proposes a different approach as it allows assigning prioritizations for servers in the cluster that are candidates to become a leader. The algorithm is flexible as it takes into consideration during the leader election process of the different server settings and roles, network properties, communication latency or specific application requirements.

1. INTRODUCTION

In general, distributed systems are designed to use symmetric nodes - all nodes have similar roles or responsibilities. However there are situations where a specific type of processing must be done on a single node, critical processes or there are situations when it is more efficient to do the processing on a single node at a time. In this case, in order to ensure a high degree of availability in case there is a failure of the leader node, any other viable node from the cluster can and should assume the leader role. So far, most election algorithms focused on efficiency in terms of size of communication between nodes and maximum latency until a leader is elected. Also, many algorithms assign a uniform role to each node during the election procession as each node can vote either for itself or for any other node, in general leading to a broadcast type of communication until a leader is elected.

In this paper we propose a different approach, whereby using a third party coordination tool we can complete the leader election in fewer steps but relying on a temporary election node.

Received by the editors: 25 September 2020.

2010 *Mathematics Subject Classification.* 68U99.

1998 *CR Categories and Descriptors.* C.2.4 [**Computer-Communication Network**]: Distributed Systems– *Distributed applications*;

Key words and phrases. leader election, algorithm, cluster, ZooKeeper.

Why another algorithm?

The algorithms running over ZooKeeper subject raise interest as the scientific community is still trying to find a solution to improve their consistency and performance, as recently presented papers prove. In 'Strong and Efficient Consistency with Consistency-Aware Durability' (2020), ORCA algorithm is proposed[1]; ZabAA and ZabAC algorithms were proposed in [6]; ZabCT algorithm in [7]. ORCA is presented as a modified version of ZooKeeper that implements CAD (Consistency-aware Durability) and also cross-client monotonic reads. There are experimentally results that suggest that ORCA provides strong consistency while closely matching the performance of weakly consistent ZooKeeper[1]. The fact that a communication network latency influences the leader election process is treated in other research papers that implemented a prototype algorithm based on ZooKeeper in order to emulate wide area systems in which the transmission delays can have a huge impact over the efficiency[4]. Other research proposes a model based on watchers in ZooKeeper and define a watch as a trigger that causes an event to be dispatched to the client whenever the watched resource changes its state. Due to the fact that the processes are asynchronous and as a consequence, the network latency gives rise to multiple possible orderings of network messages; so the model was improved in order to enable consistency [2,3]. Another proposal for a leader election algorithm for replicated services that are based on a leader, updates propagation and client request was POLE (Performance-Oriented Leader Election)[5], the algorithm selects the leader depending on an application specificity. The specificity can be defined as a metric, for example the recovery time or request latency can be used. The Pole algorithm was evaluated using ZooKeeper and the results showed that just optimizing the latency of consensus does not translate into lower latency for clients. An important conclusion from our results is that obtaining a general strategy that satisfies a wide range of requirements is difficult, which implies that configurability is indispensable for practical leader election [5].

However, none of the above algorithms relate to applications that have apart from generic constraints (server capacity, network latency), other constraints, for example the new leader should belong to a cluster that is geographically located in a different cluster from the previous leader. This behaviour differentiates it from the other algorithms, thus, the proposed approach is generic and flexible.

2. Apache Zookeeper

Zookeeper is an open source Apache project, which was designed as a service that propagates changes in the distributed systems using an improved, reliable and easy to understand method. It offers a centralized service that provides configuration management capabilities, naming information, distributed synchronization, group services, configuration information and leader election receipts over clusters in distributed systems [10,11,14].

2.1. **Leader Election Process in Zookeeper.**

As soon as a new leader is elected, it begins to serve the client's requests. Each client request contains a command with data to be applied to the state machine. The leader appends the command to its log and begins the notification process for the other servers. After the log entry from the leader was replicated on the majority of servers, the leader applies the command in its state machine. In fact, the entry is committed and the leader sends an acknowledge message to the client and informs the other read replicas servers (followers). When a follower receives the acknowledge message regarding a committed entry, it updates its own state machine based on that message. The inconsistencies that might appear between the leader's log and the follower's logs are solved by pushing the server's log version to the follower's log versions [8,9]. The protocol used in case of network errors is that the leader should try indefinitely to send messages to the followers. Data consistency is guaranteed by timers usage, so the followers logs will contain only valid data [12].

In the following we present some definitions related to Zookeeper [15]:

- **znode** - The basic data structure used by Zookeeper. It can contain some data, additional z-nodes children and several attributes (creation time, version number, so on.)
- **zk-session** - A standard TCP session established between the client and the Zookeeper server. The Zookeeper server permanently monitors the session for interruptions or timeouts by sending periodical probes. If the client fails to respond within the configured timeframe, a session may be expired and all the ephemeral z-nodes are automatically removed. A connection is established with any Zookeeper node from the cluster. If the chosen node fails, the connection migrates to another available node. This is transparent for the client.
- **watches** - A zookeeper client can configure various watches on selected z-nodes so it is informed of any change happening on these z-nodes.

2.2. **Consistency guarantees.** According to the specification [15], Zookeeper provides the following consistency guarantees:

- **"Sequential Consistency** : Updates from a client will be applied in the order that they were sent.
- **Atomicity** : Updates either succeed or fail – there are no partial results.
- **Single System Image** : A client will see the same view of the service regardless of the server that it connects to. i.e., a client will never see an older view of the system even if the client fails over to a different server with the same session.
- **Reliability** : Once an update has been applied, it will persist from that time forward until a client overwrites the update. This guarantee has two corollaries:
  - If a client gets a successful return code, the update will have been applied. On some failures (communication errors, timeouts, etc) the client will not know if the update has been applied or not. We take steps to minimize the failures, but the guarantee is only present with successful return codes.
  - Any updates that are seen by the client, through a read request or successful update, will never be rolled back when recovering from server failures.
- **Timeliness**: The clients view of the system is guaranteed to be up-to-date within a certain time bound (on the order of tens of seconds). Either system changes will be seen by a client within this bound, or the client will detect a service outage."

By providing the above mentioned consistency guarantees Zookeeper can be used to build higher-order primitives such as queues, locks, two-phase commit protocols and leader elections for other solutions.

2.3. **Leader election in ZooKeeper using SEQUENCE—EPHEMERAL flags algorithm.** In ZooKeeper documentation, the proposed leader election algorithm is based on the usage of two flags called *SEQUENCE |EPHEMERAL*. The flags are used when creating znodes that represent "proposals" of clients. The ephemeral znodes exist as long as the session that created the znodes is active; when the session ends the ephemeral znodes are deleted. For the sequence znodes: based on a request issued to Zookeeper when creating a z-node, Zookeeper can append a monotonically increasing counter to the end of path. The appended counter is unique to the parent znode [16].

The basic idea is to have a znode, named "/election", and that each znode creates a child znode "/election/guid-n_" with both flags Sequence and Ephemeral. The sequence flag is used to automatically append a sequence number greater than any one number previously appended to a child of "/election". The implications are that the process that created the znode having the smallest appended sequence number is the leader node [13].

Additionally, the leader failure case must be treated in order to insure the selection of a new node to become a leader. The simplest solution is to have all application processes checking constantly the current smallest znode, and in case the smallest znode is not replying checking if they should be the new leader. However this approach causes an undesired effect as all the processes receive a notification after the leader has failed, and they initiate the process to obtain the current list of children nodes from "/election". The number of the servers/znodes is directly proportional with the number of operations that ZooKeeper servers have to process. The optimization used in order to avoid this scenario is to check the next znode down on the znodes sequence. The algorithm written in pseudocode is the following [13]:

```
Create znode z with path "ELECTION/guid-n_" with both SEQUENCE and EPHEMERAL flags;
Let C be the children of "ELECTION", and i be the sequence number of z;
Watch for changes on "ELECTION/guid-n_j", where j is the largest sequence number such that j ¡ i and n_j is a znode in C;
Upon receiving a notification of znode deletion:
Let C be the new set of children of ELECTION;
If z is the smallest node in C, then execute leader procedure;
Otherwise, watch for changes on "ELECTION/guid-n_j", where j is the largest sequence number such that j <i and n_j is a znode in C;
```

Although we understand that this algorithm is just a basic example and it was not designed to be used directly in production as is, we believe that it is useful to analyse some of the shortcomings of this proposed algorithm and to provide an improved alternative.

Based on our experience the algorithm proposed by Zookeeper team has the following issues:

- No flexibility regarding leader election - the oldest node alive, with the lowest sequence is always elected as leader.
- The fact that a leader is elected does not always translate into that node actually becoming a leader. Depending on application the transition to leader status can be an elaborate process which may last

longer or it may fail. Only this transition has completed the leader is actually active and this moment should be used to notify the other nodes that the election process has been completed.

3. Algorithm description

As the previous paragraph detailed, most of the election algorithms in distributed systems were focused on efficiency in terms of size of communication between nodes and maximum latency until a leader is elected. However the efficiency in the election process does not guarantee the efficiency of the system during the processing phase. Moreover, many algorithms assign a uniform role to each node during the election process, the nodes are equals and each node can vote either for itself or for any other node. This approach is leading in general to a broadcast type of communication until a leader is elected. In this paper we propose a different approach, in which the nodes are assigned different priorities, their vote can have a different impact and weight. The algorithm uses Zookeeper as a third party coordination tool; using this tool, the leader election process can be completed using node's predefined priorities and can provide additional guarantees regarding the election process.

3.1. **Node roles.** Although, in general, all the nodes in the cluster can be identical, they perform various roles during the operational lifetime.

- Election node - This node runs the election process.
- Leader node - This node performs some critical activity which must be done on a single instance at a time.

These roles are dynamic and transitory, meaning that, in general, there is no static configuration regarding which node is a leader or an elector node. Any valid node can assume these roles.

3.2. **Zookeeper data structure.** In order to manage the cluster configuration and the election process the algorithm uses three parent znodes:

- **nodes** - contains one ephemeral *znode* for each active node. Each znode contains more detailed information about each cluster member.
- **election** - contains one *znode* with emphflags ephemeral |sequential for each active node. Used to select the election node, by default the node with the lowest sequence.
- **leader** - contains only two *znodes*:
  - **elected** - created by the election algorithm, identifies the next leader candidate
  - **current** - created by the leader candidate

3.3. **Leaders.** Both solutions use the concept of Leaders for long-term, steady operations. This decision is in contrast with Paxos family of algorithms where each operation is voted by a majority of nodes, a method which requires more round-trip communications between nodes. Using a master node, on the other hand, involves a much simpler communication between the leader and its followers. The leaders are elected using a consensus algorithm between the candidates or the up-to-date replicas.

3.4. **Process startup.** First of all, during startup each process must register in the cluster by connecting to a common Zookeeper cluster. This involves the following steps:

- Connecting to Zookeeper which starts a *zk-session*.
- Creating a *znode* under a certain path with node information (nodeId). This z-node is ephemeral, meaning that is automatically removed when the *zk-session* times out. This node is not used in the election process, but it only contains some useful instance information such as IP address, location (site) and possible other info.
- Creating a *znode* in order to register as a potential election node. This node is created with a sequential flag, meaning that Zookeeper will allocate a unique, sequential id to each node. The data value for this node is also the *nodeId.* This is used to select the election node - the node which will run the election process.
- Registering Zookeeper watches on cluster *znode, election znode* and leader *znode.*

3.5. **Election process.** The election process is triggered by any change in the list of *znodes* under the *election path.* Every time a new node is added to the cluster or there is a failure and one node stops, the associated *zk-session* is timed-out and the ephemeral *znodes* created by this process are removed from the Zookeeper repository. These changes are notified immediately to all the remaining nodes. By reading the remaining *election z-nodes* and comparing its own *nodeId* only the oldest process alive (with the lowest sequence assigned by Zookeeper) will execute the election process. This node will assume the temporary role of ***elector node***.

The election process can be designed to be highly flexible by assigning different priorities to different nodes. Some of these election strategies are discussed in the next section. But, in all cases, at the end of the election process the algorithm chooses one candidate as the next leader. There are cases when this candidate is the actual leader because a non-leader node exited the cluster, so nothing else happens and the process stops here.

At the end of the election process, if the elected node is different from the current leader, we create a new ***elected znode*** with the nodeId of the new leader candidate.

From here the next processing happens in parallel as all the nodes also monitor the znodes under the *leader* path:

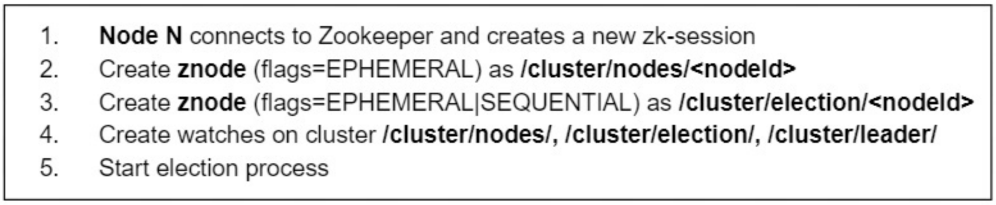
The existing leader node: For it, the presence of a new elected leader may mean that it must voluntarily give up the leader role. At the end of this process it deletes the ***leader znode***. A leader or candidate node monitors the *"elected znode"* and if it was replaced by a different z-node it must immediately stop the leader role or the leader transition process.

The elected leader node: When a node detects the presence of a *new elected zone* which matches its own id, it automatically starts a process to become a leader. But, before that, it announces its intention to become a leader by creating the ***leader znode*** with a specific status - PROGRESS. If the *leader znode* already exists - maybe because the existing leader has not removed it yet, then this creation is retried after a short delay. After successfully creating the *leader znode* it executes the required procedures and after that it updates the *leader znode* with status READY, meaning that the cluster has a new leader which is ready for processing.

3.6. **Leader transition watchdog.** Additionally, for improved robustness of the solution we can include a leader transition watchdog. If an elected leader does not manage to become leader in X seconds, the election node will run the algorithm again by excluding the previously selected leader.

3.7. **Additional considerations.** The election algorithm runs in the callback thread used by Zookeeper client library for notifications, which means that the process is not re-entrant. If the cluster configuration changes while the election process is running, the process simply runs again when the next notification is delivered. This implies that an elected node must always be ready to abort the leader transition at any time, even if just started.

3.8. **Principal methods.** A. Node startup:

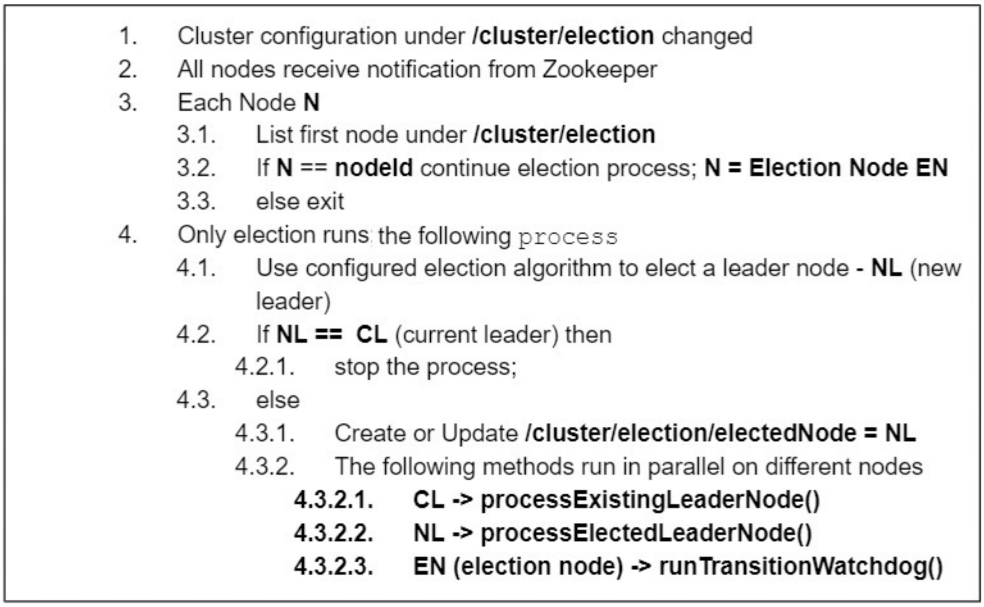


1. **Node N** connects to Zookeeper and creates a new zk-session
2. Create **znode** (flags=EPHEMERAL) as **/cluster/nodes/<nodeId>**
3. Create **znode** (flags=EPHEMERAL|SEQUENTIAL) as **/cluster/election/<nodeId>**
4. Create watches on cluster **/cluster/nodes/, /cluster/election/, /cluster/leader/**
5. Start election process

Election process is also triggered when any node under **/cluster/election/**

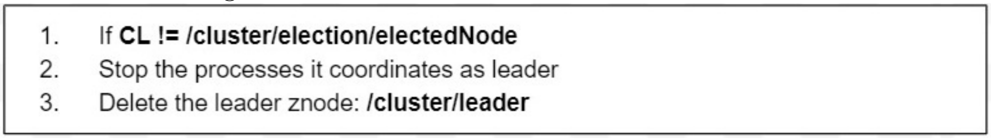
changes, which means when one node disconnects or when a new one re-joins the cluster.

B. Election:



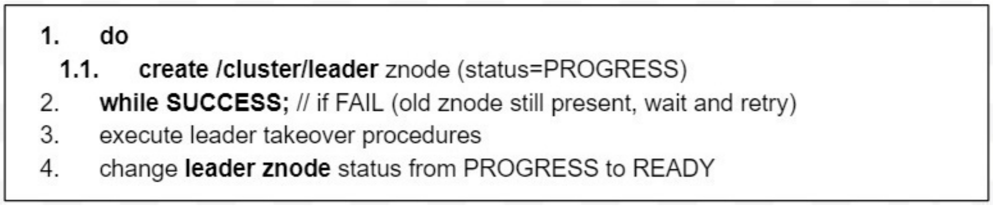
```
1.  Cluster configuration under /cluster/election changed
2.  All nodes receive notification from Zookeeper
3.  Each Node N
    3.1.  List first node under /cluster/election
    3.2.  If N == nodeId continue election process; N = Election Node EN
    3.3.  else exit
4.  Only election runs the following process
    4.1.  Use configured election algorithm to elect a leader node - NL (new
          leader)
    4.2.  If NL ==  CL (current leader) then
        4.2.1.  stop the process;
    4.3.  else
        4.3.1.  Create or Update /cluster/election/electedNode = NL
        4.3.2.  The following methods run in parallel on different nodes
            4.3.2.1.  CL -> processExistingLeaderNode()
            4.3.2.2.  NL -> processElectedLeaderNode()
            4.3.2.3.  EN (election node) -> runTransitionWatchdog()
```

C. ProcessExistingLeaderNode



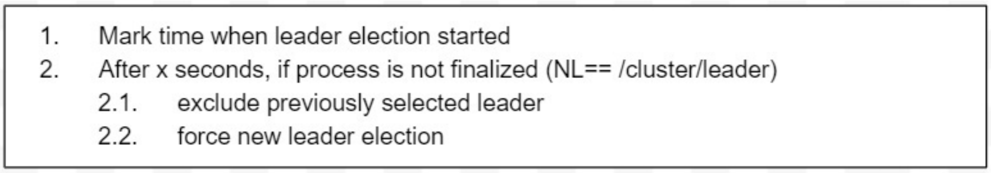
```
1.  If CL != /cluster/election/electedNode
2.  Stop the processes it coordinates as leader
3.  Delete the leader znode: /cluster/leader
```

D. ProcessElectedLeaderNode(candidate znode)



```
1.  do
  1.1.  create /cluster/leader znode (status=PROGRESS)
2.  while SUCCESS; // if FAIL (old znode still present, wait and retry)
3.  execute leader takeover procedures
4.  change leader znode status from PROGRESS to READY
```

E. RunTransitionWatchdog



```
1.  Mark time when leader election started
2.  After x seconds, if process is not finalized (NL== /cluster/leader)
    2.1.  exclude previously selected leader
    2.2.  force new leader election
```

3.9. **Election strategies or policies.** The most simple election strategies would be to simply pick the first available node, based on their nodeId order or, alternatively, to use a round-robin method.

Another possibility is to assign all the nodes to different sites or data centers, based on their geographical location. For various reasons, nodes belonging to a particular site are preferred over others. This site preference or priority is not static, but it change dynamically during normal operation:

- If some critical error disables an entire site, for instance due to failure of some shared network or storage equipment. In these cases, even if there are nodes available in the primary site, it may be better/safer to move the processing to the backup site.
- If there is a planned maintenance operation which impacts all the nodes in site, the administrator can simply move the leader to another site by temporarily assigning a higher priority to the backup site.

The election algorithm can be configured to support multiple strategies and pick the most appropriate one based on the exact circumstances when the election is run.

4. Conclusion and Future Work

There are applications that have specific rules, applications that are processing sensitive data and for which a cloud installation is out of discussion. For contingency reasons, the servers are split into clusters located in different places and have additional constraints such as: the new leader should belong to a different cluster from the previous leader. None of the mentioned algorithms is enough flexible to allow this approach. All the algorithms have a predefined standard set of constraints and they adjust the election process and the algorithm behavior based on the same parameter or set of parameters. Our algorithm permits to set different priorities for the znodes, influencing the chances to be elected and offering a wider set of methods to customize the election process.

The proposed algorithm was designed to offer a lot of flexibility regarding the criterias used in a leader election process, so it maps on a range of applications that require a customized approach. Even if it adds a new layer of processing, it allows to prioritize the servers in the election process, thus enabling a high degree of customization for each application type, taking into account not only different metrics such as latency, but also requirements such as the locations of the leader server. There are critical applications that require a DR site (disaster recovery site), where the nodes should replicate the information posted and processed in the live site. For these types of applications, the leader election process has other constraints: in case a leader fails, the processing should be automatically moved to the other site and the new leader should be selected as one of the nodes from that site. The proposed algorithm addresses these requirements and can also accommodate other application's specific requirements.

Another benefit added by our proposed algorithm is that the leader transition happens in two stages: first the new leader is notified and second, only after the successful completion of the transition process the new leader announces that the new leader is ready to receive requests. This ensures that if the leader transition does not proceed as planned, the process can be retried by another candidate.

The algorithm can be easily extended into a multi-tenant operation, where there are multiple leaders at the same time, one for each critical resource. This can be achieved simply by using the Zookeeper znode hierarchy which is modeled like a tree. In such a multi-tenant operation we would have one dedicated data structure as described in the Process Startup phase for each tenant, so that each tenant runs completely isolated from others.

Another possibility is to slightly change the algorithms to support more than one leader at the same time for the same resource. Such an approach is similar to a configuration with multiple Active and Spare nodes (or primary/backup architectures) - where spare nodes are not actually stopped, but idle and waiting to resume processing or take over the leader/active node as required.

The future work will consist in developing and running a set of tests in order to check how the system will behave under heavy loading and also to try to find out if there are vulnerabilities in the proposed algorithm.

References

[1] Artho, C., Banzai, K., Gros, Q., Rousset, G., Ma, L., Kitamura, T., Yamamoto, M., Model based testing of Apache ZooKeeper: Fundamental API usage and watchers. Software Testing, Verification and Reliability, 2019, DOI:10.1002/stvr.1720

[2] Artho C, Gros Q, Rousset G, Banzai K, Ma L, Kitamura T, Hagiya M, Tanabe Y, Yamamoto M. Model-based API testing of Apache ZooKeeper. Proc. 2017 IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2017): Tokyo, Japan, 2017; pp. 288-298.

[3] Becker D., Junqueira F., Serafini M., Leader Election for Replicated Services Using Application Scores.,2011, DOI 7049. 289-308. 10.1007/978-3-642-25821-3 15.

[4] EL-Sanosi I.,Ezhilchelvan P.,Improving the Latency and Throughput of ZooKeeper Atomic Broadcast, Imperial College Computing Student Workshop, 2018, pp. 3:1–3:10, ISBN 978-3-95977-059-0, DOI 10.4230/OASIcs.ICCSW.2017.3

[5] EL-Sanosi I. , Ezhilchelvan, P., Improving ZooKeeper Atomic Broadcast Performance by Coin Tossing, Lecture Notes in Computer Science, 2017, pp.249-265. DOI:10.1007/978-3-319-66583-2_16

[6] Ganesan A., Alagappan R., Arpaci-Dusseau A., Arpaci-Dusseau R., Strong and Efficient Consistency with Consistency-Aware Durability, 18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, 2020, ISBN 978-1-939133-12-0

[7] Hunt P, Konar M, Junqueira F, Reed B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. Proc. USENIX Annual Technical Conf., USENIXATC, USENIX Association: Boston, USA, 2010; 11'11. DOI:doi=10.1.1.178.5750

[8] Junqueira F., Reed B. ZooKeeper: Distributed Process Coordination. O'Reilly, 2013, ISBN-13: 978-1449361303

[9] Medeiros A., ZooKeeper's atomic broadcast protocol: Theory and practice, Helsinki University of Technology, 2012, DOI: 10.1.1.473.1373

[10] Medeiros A., ZooKeeper's atomic broadcast protocol: Theory and practice,2012, retrieved from http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf

[11] Petrescu M., Replication in Raft vs Apache Zookeeper, Advances in Intelligent Systems and Computing, 2020, ISSN 2194-5357

[12] Petrescu M., Petrescu R., Log replication in Raft vs Kafka, Studia Universitas Babes-Bolyai, 2020, DOI 10.24193/subbi.2020.2.05

[13] Santos, N. H., Andre M.S., Latency-aware Leader Election.,2009, DOI 10.1145/1529282.1529513.

[14] ZooKeeper 3.6 Documentation / ZooKeeper Recipes and Solutions, 2020, retrieved from https://zookeeper.apache.org/doc/r3.6.2/recipes.html

[15] ZooKeeper 3.6 Documentation / ZooKeeper Programmer's Guide, 2020, retrieved from https://zookeeper.apache.org/doc/r3.6.2/zookeeperProgrammers.html

[16] ZooKeeper 3.6 Documentation / The ZooKeeper Data Model, 2020, retrieved from https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html#Ephemeral+Nodes

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeș-Bolyai University, 1 Kogălniceanu St., 400084 Cluj-Napoca, Romania

Email address: `mpetrescu@cs.ubbcluj.ro`