When Machine Learning Meets Raft: How to Elect a Leader over a Network

Kostas Choumas and Thanasis Korakis
Dept. of ECE, University of Thessaly, Volos, Greece
Email: kohoumas, korakis@uth.gr

Abstract*—Numerous well-known applications use the Raft consensus algorithm to maintain consistent replicas of their data on distributed nodes. Raft is based on a dynamically elected leader who is one of the distributed nodes, and its operations are unfortunately suspended during the election of the leader. Elections can be triggered by the failure of the current leader, in which case they are unavoidable, or by a network disconnect between the leader and another node, in which case a new inefficient leader will likely replace the previous one at the expense of additional system downtime. In this paper, Raft messages are monitored at every node, and Machine Learning is used to classify the aforementioned causes of each election. This data is used to increase the system's availability by decreasing the total number of elections that could be conducted in a given time unit. Three supervised classifiers were trained with messages generated in a real Raft-operated distributed system that was deployed on a testbed and where multiple events triggering elections were applied. All classifiers are nearly 97% accurate at classifying the causes of these elections, approaching even 100% in some cases.

***Index Terms*—Machine Learning, Raft, testbed experimentation.**

I. Introduction

Clusters of multiple distributed nodes are utilized by numerous applications to execute their mission-critical operations, resulting in their increased fault tolerance. These nodes function as state machines that maintain identical copies of the same data and continue to support the application operations even if some nodes have failed. The majority of these applications (including OpenDaylight, ONOS, Kubernetes, OpenStack, Docker, and Hyperledger-Fabric, just to name a few) use the *Raft consensus algorithm* [1], [2] for maintaining consistent replicas of the same data across the nodes. The data replication, according to Raft, is carried out by a single leader that is dynamically elected from among the nodes. Regardless of which node introduces a data change, this change is ultimately forwarded to the leader for replication. As a result, the application is not available during the election process, since there is no leader. Consequently, the application availability declines as the number of elections per time unit increases.

The spatial placement of the distributed nodes is an additional reason for triggering elections and decreasing application availability. It increases the application resilience to natural disasters and power outages, but also introduces *link failures*, which are distinct from the *node failures* that Raft is designed to handle. According to the Raft design, election is activated when at least one node realizes it is no longer connected to the leader, assuming that the leader is failed, and

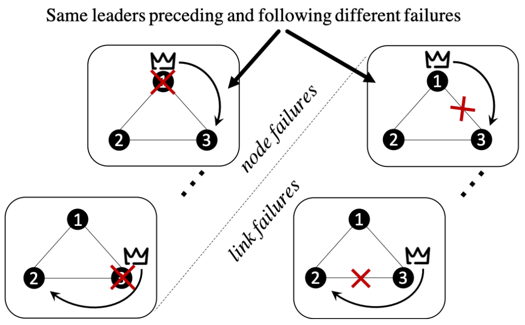

Fig. 1. ML classification as either node or link failures.

it is likely that this node will succeed the leader. In the event of disconnection due to link failure, and assuming that the link is unstable and goes up and down numerous times, the same number of elections are triggered to transfer the leadership between the two nodes. If a third node, connected to all other nodes, had been chosen as the new leader, these elections could have been avoided.

Our contribution: *Machine Learning* (ML) is used in this paper to classify the failures triggering elections as either link or node failures. As depicted in Figure 1, at each node, a model generated by a ML classifier reads the Raft messages to predict the failure type. Once a new leader has been elected, each node is aware that a failure triggered the leadership transition and uses its ML model for the classification of this failure. This is useful because a set of actions should be carried out, such as transferring the leadership to another node, if the same link has repeatedly failed and triggered numerous elections. The effectiveness of three ML classifiers in categorizing the aforementioned failures is assessed and presented below.

The rest of the paper is organized as follows: Section II presents related work to Raft optimizations that primarily take use of leadership transition, whereas Section III provides an overview of the Raft algorithm. Section IV describes how ML can be used to enhance Raft, whereas Section V provides the results of testbed experiments to evaluate the ML assistance in Raft. Finally, Section VI concludes the paper and discusses implications for future research.

II. Related Work

In [3] and [4], heuristics for the placement of a distributed cluster of Software Defined Networking (SDN) controllers using Raft for synchronization are presented. Regardless of which controller is the master, which is also the Raft leader, the presented solutions reduce the overall control overhead. The modeling and numerical evaluation of Raft-operated distributed clusters of SDN controllers are also presented in [5], utilizing Stochastic Activity Networks and estimating the effect of various hardware and software failures on the response time, irrespective of the network's impact on the election process. In contrast, in [6] and [7], the authors concentrate on the Raft leader and investigate the network effect on the leader election process, and consequently on the Raft performance. [8] also considers the network effect on the Raft operation, proposing the use of dedicated P4-based network devices to offload a portion of the Raft operation to the network. In [9], the underlying network connecting the cluster of ONOS controllers is also considered for the master controller election, which is determined by the Raft leader.

In [10], an appropriate configuration of the parameters of the election process is proposed in an effort to compel the replacement of an overloaded but still-living leader. The authors argue that an overloaded leader sends messages with a high time deviation; consequently, they recommend that receivers campaign for the next leadership if they detect a time difference between the leader's messages. In [11], two algorithms to minimize the number of elections without a winner are proposed (due to the *split vote*, that will be further explained later).

In [12], a leadership transfer algorithm is proposed to prevent the network from splitting by replacing the current leader if its links to the other nodes are mostly unreliable. A leader selection algorithm is designed to select the next leader based on a reputation model that evaluates the stability of all network nodes. Similarly, in [13], the authors propose three criteria and suggest exploiting federated learning to evaluate them for selecting a better next leader that increases network stability. Lastly, in [14], the election parameters are optimized to minimize the probability that the majority of the cluster cannot reach the leader due to packet loss on the network links. All these works focus on manipulating the succession of Raft's leadership in different ways, thereby avoiding multiple ineffective elections that reduce availability. In this work, we employ ML to track the history of Raft messages and intervene as necessary to prevent ineffective elections.

III. Raft overview

As introduced previously, a Raft-operated application utilizes a cluster of nodes that are state machines. Their state reflects the data of the application. They begin in the same state and only change it after executing the same sequence of deterministic *commands* in the same order. This sequence of commands is named *log* and is replicated among the nodes. Keeping the replicated log consistent among all nodes is the contribution of the Raft consensus algorithm, which is achieved with the assistance of a *leader*. The leader is elected when cluster begins or the current leader fails. Between two successive leader elections, there is at most one leader and the other nodes are *followers*, whereas during the elections there is no leader and some nodes become *candidates* when they realise the absence of a leader. The periods between the elections are identified by a consecutive integer number, named *term*.

Each node begins as a follower and remains as such as long as it continues to receive requests from the leader or a candidate. The leader sends periodic *heartbeat requests* to all followers for maintaining its leadership and collects their *heartbeat replies*. The *heartbeat timeout* is the period between heartbeats, which should be slightly longer than the average time required for a node to send requests to all other nodes in parallel and receive their responses. If a follower does not receive heartbeat over a random time period called *election timeout*, which should be an order of magnitude longer than the heartbeat timeout, then the follower assumes there is no viable leader and begins an election, increasing also its term by one. This occurs for all nodes upon startup, since none of them is leader.

The follower initiates an election by transitioning to candidate, voting for itself, and simultaneously issuing *vote requests* to other nodes. Then, one of three outcomes occurs: (a) the candidate becomes leader after receiving positive *vote replies* from more than half of the cluster, (b) another candidate establishes itself as leader and this candidate receives a heartbeat from the new leader and transitions back to follower, or (c) there is no winner due to a split vote, as none of the candidates collects the required number of positive votes. When a split vote occurs, each candidate waits again for a random election timeout to expire in order to increase its term, and then initiates a new round of vote requests, starting a new election. The packets sent by each node are labeled with its term, which may differ from the terms of the other nodes. Noting that an odd number of cluster nodes reduces the likelihood of a split vote, it is recommended that a cluster has 3 or 5 nodes. Clusters with more than 7 nodes are afflicted by high control overhead.

All commands introduced by the application, including those received by the followers and redirected to the leader, are appended to the log of the leader. Once new commands have been appended, the leader replicates its log by sending *append requests* in parallel to all followers and including the log in these requests. When the leader receives *append replies* from more than half of the cluster, the commands of the replicated log are deemed *committed* and they are executed by the leader. Then, the leader resends *empty append requests* to the followers without any log, forcing them to execute the already delivered and committed commands. If followers crash or network packets are lost, the leader resends append requests until all followers eventually execute the committed commands and send their append replies.

IV. ML assisted Raft

As described in Section III, an election is triggered when a follower becomes candidate and sends vote requests. This occurs when the follower has not received a heartbeat request from the current leader for the duration of its election timeout.

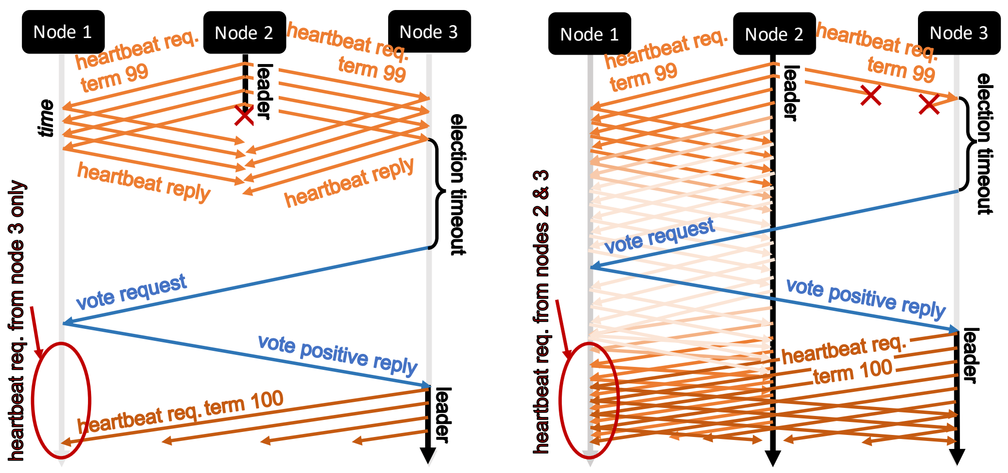

Fig. 2. Possible series of Raft messages exchanged on a 3-node cluster in the event of node 2 (at left) or link $2-3$ (at right) failure.

It is presumed that heartbeat requests have ceased since the current leader has failed. However, the heartbeat requests could be missed if the link between the leader and a follower fails. In the event of link failure as opposed to node failure, the leadership is likely transferred to the follower using this link. If this link moves up and down again, the same phenomenon will occur, with the leadership shifting from one end of the link to the other [15]. Due to an unstable link that repeatedly moves up and down, multiple successional leadership transitions could occur, which could be avoided by assigning leadership to a third node. In addition, the third node would be able to connect to all nodes, whereas the two nodes at the ends of the failed link are unable to connect to each other.

This is the reasoning for utilizing ML to decrease the number of Raft elections and increase the availability of a Raft-operated application. The ML classification of the trigger of a leadership transition as either node or link failure, can be extremely helpful in this regard. In general, once a node receives heartbeat request of a new term, it is aware that a new leader has been elected, but it does not know why. According to our design, each node is able to independently monitor its exchanged messages and uses the most recent ones before the heartbeat request to determine the cause of the leadership transition. Figure 2 illustrates the distinctions between two possible series of Raft messages exchanged on a 3-node cluster in the event of a node or link failure.

Before and after the failure, the leaders in this example are node 2 and node 3. When node 1 receives the first heartbeat request of term 100 from node 3, it is simple for it to determine that a failure has occurred and that node 3 is the new leader. Combining this information with the previously received heartbeat requests of term 99 from node 2, it concludes that the leadership transition is from node 2 to node 3, but does not know the cause of this change. As depicted, the leadership change could be triggered by the failure of node 2 or link $2-3$ (left or right part of Figure 2, respectively). Under the node failure scenario, node 1 only receives heartbeat requests from node 3, whereas under the link failure scenario, node 1 receives heartbeat requests from all other nodes; however, the random series of messages may be considerably altered, necessitating ML for enabling node 1 to classify the failure.

In Section V, it is presented how various ML classifiers are used to create models that classify failures, trained offline with series of Raft messages collected by one node, e.g. node 1. Prior to each heartbeat request for a new term, a series of Raft messages has been exchanged and it is used as training sample. The inference of the produced models enables node 1 to predict the type of the failure that caused the last leadership transition, in order to campaign or not for the next leadership, as the current leader may be unstable in the event of multiple link failures. Similar to node 1, the same procedure is repeated on all other cluster nodes.

In particular, the *Decision Tree* (DT), *Support Vector Machines* (SVM) and *K-Nearest Neighbours* (KNN) ML classifiers are evaluated. The DT classifiers generate tree structures that, beginning at the root of the tree, examine each tree node to determine if one of the sample values is less than a predetermined threshold. This process is repeated until they reach a leaf that is always mapped to a node or link failure. The SVM classifier defines a hyperplane that divides the samples into two spaces, one for each failure, with the utmost distance from the nearest sample on both of its sides. The KNN classifier relies on the majority vote of the k nearest neighbors of each sample.

V. Implementation & Testbed Experimentation

The presented experimentation relies on the NITOS [16] testbed and the open-source implementations of the Raft protocol and ML classifiers, *etcd* [17] (version 3.3.0-rc.0) and *scikit-learn* [18] (version 0.24.1) respectively. The messages are monitored using *pyshark* and a custom script that extracts Raft messages from TCP segments. Time is designated and the duration of each slot is 100 milliseconds. The heartbeat

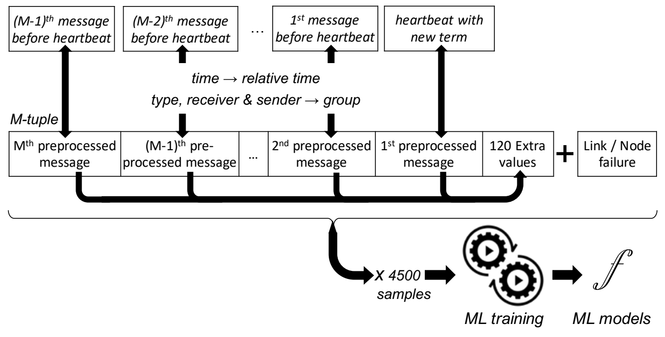

Fig. 3. Supervised training of ML classifiers.

timeout is 1 slot (100 milliseconds), while the election timeout is chosen at random between 10 and 19 slots (1 second and 1.9 seconds). In addition to the Raft messages, which consist of the heartbeat, vote, and append requests/replies, *link-heartbeat* messages are periodically exchanged every 5/3 seconds. There are two TCP connections named *message stream* and *msgApp stream* between each pair of nodes. Almost all messages, including heartbeat and vote requests/responses, empty append requests, and all append replies, use the message stream. In contrast, the msgApp stream is utilized solely for non-empty append requests that include the log. Link-heartbeats are transmitted across both streams.

During the experimentation, an etcd cluster is deployed at NITOS, and each node is connected to the other nodes via links that are independent of one another. At the conclusion of each round, either the leader or a randomly chosen link adjacent to the leader is failed for 5 seconds. The probabilities of selecting each failure are equal. Then, the node or the link recovers, and the procedure is repeated 5 seconds later. This procedure is repeated 5000 times, creating the same number of samples and matching them with the failures that caused them. Since failures are sporadic, we presume that multiple failures cannot occur at the same moment. Each sample consists of the M most recent messages exchanged by node 1, across all its TCP streams prior to the first heartbeat request of a new term (including this heartbeat).

The sample is a M-tuple of preprocessed messages, during which only the *timestamp*, *type*, *receiver* and *sender* of the original message are retained and edited appropriately. The timestamp of each message is replaced with its *relative time*, which is the absolute difference from the timestamp of the most recent message in the tuple. For instance, if timestamp is 10000 microseconds and the timestamp of the most recent message is 10002 microseconds, then relative time is 2 microseconds. In addition, the type, receiver and sender of each message are replaced by a *group* identifier. There are 6 type groups: one for link-heartbeats, one for heartbeat requests, one for heartbeat replies, one for both empty and non-empty append and vote requests, one for append and positive vote replies, and one for negative vote replies. It has been arranged so that messages of the same type group occupy nearly identical positions in the tuple. For example, the append or vote reply of a node typically precedes the heartbeat request of the new leader; therefore, these two message types are grouped together. Each type group is further divided into 6 subgroups based on the message receiver and sender: three for messages received by node 1 and sent by either the previous leader (*PL*), the new leader (*NL*) or another follower (*F*), and three for messages sent from node 1 to either PL, NL or F. In total, there are $6^2 = 36$ message groups.

Unlike the relative times, the groups are not ordered, so they are not included in the tuple as integers. Each message in the tuple consists of its relative time and 36 additional bits, of which only one is set to one to identify its group. This method permits the Euclidean distance to reflect the actual similarity between two samples. In addition to the M messages with their M non-negative real values for their relative times and their $36M$ boolean values for their groups, each tuple also contains the following:

- the average and variance of all message relative times,
- the 36 frequencies of all message groups in the tuple,
- the $2 \times 36 = 72$ positions in the tuple, from 1st to M^{th}, of the newest and oldest messages in each group (messages with a lower position are newer and more recent to the heartbeat with new term),
- the 3 frequencies of all packets sent or received by PL, NL, or F respectively,
- the $2 \times 3 = 6$ positions in the tuple of the newest and oldest messages sent or received by PL, NL, or F respectively,
- the frequency of all heartbeat requests in the tuple.

As follows, each sample contains $M + 120$ non-negative real values and $36M$ boolean values. Figure 3 depicts the structure of the samples and how they are fed to the ML classifiers for supervised training, along with their failure types. Training uses 90% of samples and the rest samples are used for testing. The frequencies of all leadership transitions between each pair of nodes, due to node or link failures, are nearly equal. Below is an evaluation of the three aforementioned classifiers for two cluster sizes, always beginning with the performance of the DT classifier, which is also accompanied by a graphical explanation of its results.

A. 3-*node cluster*

The initial deployment consists of 3 etcd nodes connected by 3 links with equal propagation delay.

1) DT: It is constructed using the CART algorithm. In order to prevent overfitting, the criterion for further dividing a tree node is to minimize its Gini impurity by more than 0.01, whereas the Gini impurity of each node measures the probability that a random sample will be misclassified. The sample size M has been gradually increased from 4 to 20, resulting to various models that effectively classify from 75% to 95% of the testing samples, as it is depicted in Figure 4. Figure 6 depicts the resulting DT model for $M = 20$, which effectively classifies 95% of the testing samples. All possible cases of this model are the following:

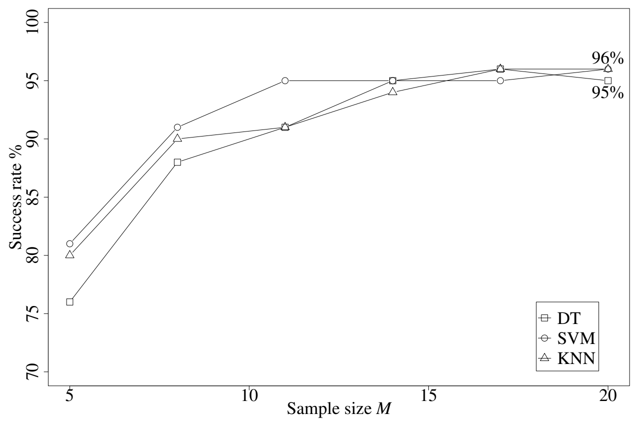

Fig. 4. Success rates of various ML classifiers for various sample sizes M.

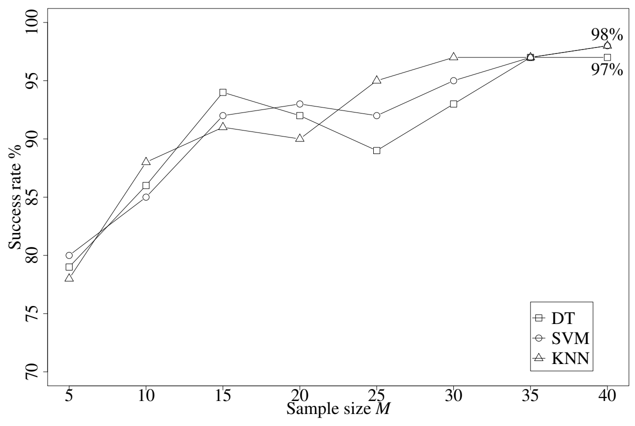

Fig. 5. Success rates of various ML classifiers for various sample sizes M.

Case A. If the relative times of the older messages in the sample are not too high, then it is a link failure, since node failure duration takes longer than link failure duration, as one may anticipate.

Case B. Else, it is a node failure if node 1 requested a vote from F and many other messages followed until the heartbeat request from NL. The high position of this vote request is most likely the result of a split vote, which does not occur in the event of link failure.

Case C. Alternatively, if the variance of relative times is adequately small, it is a link failure for the same reason as *Case A*.

Case D. Otherwise, a link failure occurs if the oldest heartbeat request with previous term to F has high position.

Case E. Otherwise, it is a node failure.

Additionally, it should be noted that nearly all (99%) incorrectly classified testing samples refer to link or node failures that transfer leadership to node 1. This indicates that the DT model of node 1 is nearly 100% effective at classifying failures in which node 1 is not involved as NL. In turn, this means that in the event of a leadership transition due to a link failure that requires actions to move the leadership from NL, the other nodes will be able to recognize the link failure and campaign for replacing this NL.

2) SVM: It uses the linear "kernel" (inner product) as a distance measure, in order to divide the samples into two spaces using a hyperplane that features the utmost distances from the nearest samples of both failures. Figure 4 depicts the success rates of SVM for a variety of sample sizes, clearly reaching 96% for $M = 20$ and above. The CPU time required for training the SVM model is approximately 1.6 times that of the DT model.

3) KNN: It uses the votes of $k = 6$ neighbors to categorize each sample, and the Euclidean metric is used to calculate the distances between the samples. Figure 4 depicts that KNN reaches again 96% for $M = 20$ and above, but the corresponding CPU time is now 3.6 times that of the DT model.

B. 5-node cluster

The classifiers are also assessed in the scenario of a 5-node cluster, connected by 10 links with equal propagation delay. The same parameters are used in all classifiers with the 3-node cluster. The only difference is the sample size, which has to be nearly $M = 40$ for classifiers to achieve their maximum success rates, as it is depicted in Figure 5.

1) DT: Its success rate is 97% for sample size $M = 40$ and above. Figure 7 depicts the DT model for $M = 40$. The cases of this model are the following:

Case A. If the variance of relative times is small enough, then it is a link failure, as the duration of a node failure is lengthier than that of a link failure.

Case B. Else, if the frequency of all heartbeat requests is high, then it is a node failure.

Case C. Otherwise, if there are no vote/append requests from F, then it is a link failure.

Case D. Otherwise, it is a node failure.

As is the case with the 3-node cluster, the model predicts failures where NL is not node 1 with an accuracy of nearly 100%.

2) SVM: Its success rate is 98% for sample size $M = 40$ and above. The CPU time required for its training is approximately 4.8 times that of the DT model.

3) KNN: Its success rate is 98% for sample size $M = 40$ and above. The CPU time required for its training is approximately 1.6 times that of the DT model.

VI. Conclusions & Future Work

This study presents the use of ML classifiers to enhance the operation of Raft, thereby increasing its availability and leadership stability. SVM seems slightly more effective than DT and KNN, with success rates of 96% and 98% for 3-node and 5-node clusters, respectively. However, it appears to require more CPU time, while DT is much faster at the expense of only 1%, and KNN falls somewhere in the middle of both dimensions. Moreover, all failures that are not successfully classified by some nodes are correctly classified by other nodes that use this information to enhance Raft. In

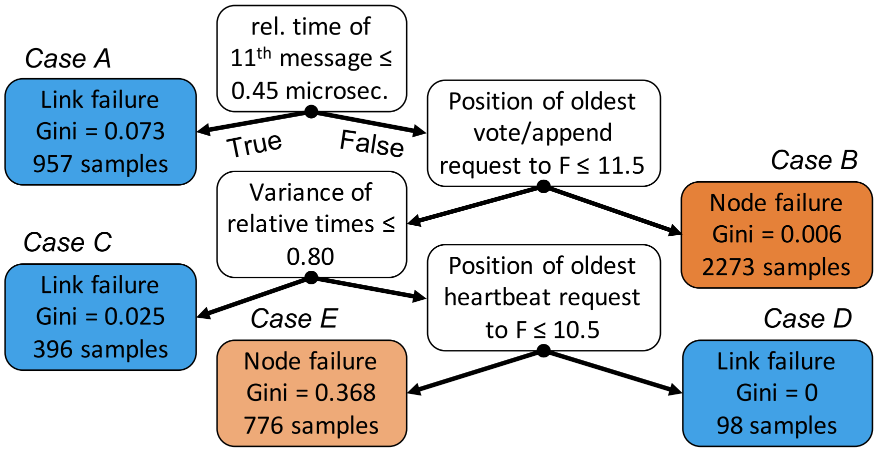
rel. time of 11th message ≤ 0.45 microsec.
True
False
Case A
Link failure Gini = 0.073 957 samples
Position of oldest vote/append request to F ≤ 11.5
Case B
Node failure Gini = 0.006 2273 samples
Variance of relative times ≤ 0.80
Case C
Link failure Gini = 0.025 396 samples
Position of oldest heartbeat request to F ≤ 10.5
Case E
Node failure Gini = 0.368 776 samples
Case D
Link failure Gini = 0 98 samples

Fig. 6. DT model for a 3-node cluster.

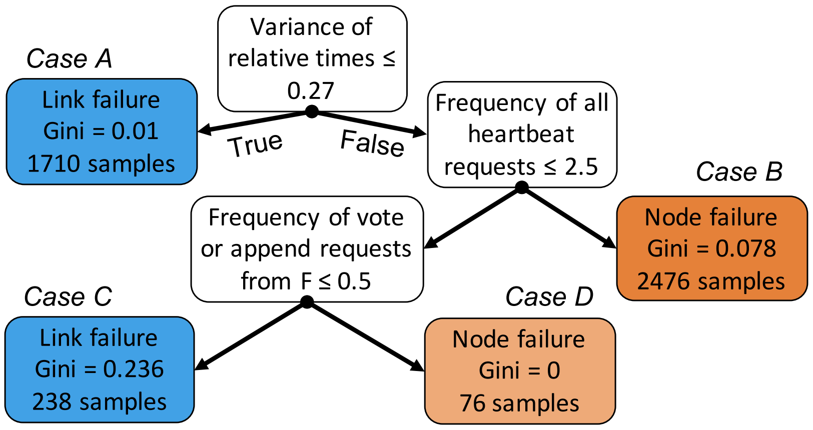
Variance of relative times ≤ 0.27
True
False
Case A
Link failure Gini = 0.01 1710 samples
Frequency of all heartbeat requests ≤ 2.5
Case B
Node failure Gini = 0.078 2476 samples
Frequency of vote or append requests from F ≤ 0.5
Case C
Link failure Gini = 0.236 238 samples
Case D
Node failure Gini = 0 76 samples

Fig. 7. DT model for a 5-node cluster.

the future, additional classifiers and unsupervised learning will be evaluated, even in the case of larger clusters.

References

[1] The Raft Consensus Algorithm. https://raft.github.io/.

[2] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX ATC*, 2014.

[3] K. Choumas, D. Giatsios, P. Flegkas, and T. Korakis. The SDN Control Plane Challenge for Minimum Control traffic: Distributed or Centralized? In *Proc. IEEE CCNC*, 2019.

[4] M. Karatisoglou, K. Choumas, and T. Korakis. Controller Placement for Minimum Control Traffic in OpenDaylight Clustering. In *Proc. IEEE WF-5G*, 2019.

[5] E. Sakic and W. Kellerer. Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane. *IEEE TNSM*, 15(1):304–318, 2018.

[6] K. Choumas and T. Korakis. When Raft Meets SDN: How to Elect a Leader over a Network. In *Proc. IEEE NetSoft*, 2020.

[7] K. Choumas and T. Korakis. On using Raft over Networks: Improving Leader Election. *IEEE TNSM*, 2022.

[8] Y. Zhang, B. Han, Z. Zhang, and V. Gopalakrishnan. Network-Assisted Raft Consensus Algorithm. In *Proc. SIGCOMM Posters and Demos*, 2017.

[9] H. I. Kobo, A. M. Abu-Mahfouz, and G. P. Hancke. Efficient controller placement and reelection mechanism in distributed control system for software defined wireless sensor networks. *Tran. on Emerging Telecommunications Technologies*, 30(6):e3588, 2019.

[10] R. Hanmer, L. Jagadeesan, V. Mendiratta, and H. Zhang. Friend or Foe: Strong Consistency vs. Overload in High-Availability Distributed Systems and SDN. In *Proc. ISSREW*, 2018.

[11] C. Fluri, D. Melnyk, and R. Wattenhofer. Improving Raft When There Are Failures. In *Proc. LADC*, 2018.

[12] Ruowen Gu and Dongyan Huang. A Leadership Transfer Algorithm for the Raft. In *Proc. Blockchain Technology and Application*, 2022.

[13] Donghee Kim, Inshil Doh, and Kijoon Chae. Improved Raft Algorithm exploiting Federated Learning for Private Blockchain performance enhancement. In *Proc. ICOIN*, 2021.

[14] D. Huang, X. Ma, and S. Zhang. Performance Analysis of the Raft Consensus Algorithm for Private Blockchains. *IEEE Tran. on Systems, Man, and Cybernetics: Systems*, 50(1):172–181, 2020.

[15] Y. Zhang et al. When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network. In *Proc. APNet*, 2017.

[16] Network Implementation Testbed using Open Source platforms (NITOS). https://nitlab.inf.uth.gr/NITlab/nitos.

[17] etcd: Distributed, reliable key-value store. https://etcd.io/.

[18] scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/.