STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXV**, Number 2, 2020
DOI: 10.24193/subbi.2020.2.05

LOG REPLICATION IN RAFT VS KAFKA

MANUELA PETRESCU AND RĂZVAN PETRESCU

Abstract. The implementation of a fault-tolerant system requires some type of consensus algorithm for correct operation. From Paxos to Viewstamped Replication and Raft multiple algorithms have been developed to handle this problem. This paper presents and compares the Raft algorithm and Apache Kafka, a distributed messaging system which, although at a higher level, implements many concepts present in Raft (strong leadership, append-only log, log compaction, etc.).

This shows that mechanisms conceived to handle one class of problems (consensus algorithms) are very useful to handle a larger category in the context of distributed systems.

1. INTRODUCTION

Due to the increased volumes and fault tolerance requested by the real-life applications, new methods emerged to implement replicated servicers and coordination client interaction with server replicas. From Paxos algorithm, to Raft and Kafka, many applications use different models in order to provide solutions for these requirements.

This paper is analyzing differences and similarities between Raft consensus algorithm and Kafka methods and consensus. Why Raft and why Kafka? Raft is a consensus algorithm that was created as an alternative to Paxos algorithm, comparable with it in terms of fault tolerance and performance, having a different structure allowing it to be easier to understand and to implement. For both of them (Kafka and Raft), logs play an essential role.

Kafka is a distributed software platform originally developed by Linked, based on log replication that allows client processes to publish and consume data. It is an alternative to the traditional queues-based messages systems (ActiveMQ).

Received by the editors: 25 September 2020.

2010 *Mathematics Subject Classification.* 68U99.

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

Key words and phrases. comparison, Raft, Kafka, consensus algorithm, replicated log.

Previous Work There are a lot of papers in the international scientific databases that are related to RAFT and to KAFTA or ZooKeeper. There are also papers that compare the algorithms, but the last ones are focusing on a comparison between the algorithm's implementation in Hyperledger Fabric [14, 15], a blockchain platform. The current paper focuses on Raft and Kafka as Apache plans to replace ZooKeeper with a RAFT based algorithm. The plans take the form of an approved change request, currently under development.

The paper is structured in four parts. The first part presents the Raft algorithm: the key aspects and the most important features are detailed, focusing on leader election and log replication algorithm. The second part presents Kafka, the leader election and log replication processes. The third part analyzes the differences and the similarities between Kafka and Raft, the last part is for conclusion and future work.

2. RAFT

Raft is an algorithm for managing replicated logs as the consensus is implemented by selecting a leader. The leader has the full responsibility of managing the logs without consulting the other servers. The typical number of servers used in a Raft cluster is five, because having five servers a system can tolerate up to two node failures.

In real life applications appeared the need for a collection of machines (nodes) to work in a coherent manner, to perform operations on reliable data which is constantly updating and to provide methods in case of node failure. One algorithm that tried to provide a solution for these requirements was Paxos algorithm, published in 1998 by Lamport.[1] The approach, based on a state machine replication, ensured that all cases in a distributed, fault-tolerant implementations are handled safely. However, according to Diego Ongaro and John Ousterhout from Stanford University [3], Paxos was neither easy to understand nor to implement. They have conducted a series of experiments with students which concluded the Raft is much easier to understand as Raft separates the key elements of consensus (such as leader election, log replication and contingency measures) and reduces the number of states in the system (for example, the logs are not allowed to have holes). The Paxos implementation in real systems such as Chubby [7], encountered many problems. Chubby implementation had to modify the Paxos algorithm, and the final solution was so different from the original algorithms that the comments from its developers are relevant in this context: "There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system . . . the final system will be based on an unproven protocol" [7].

The key features of Raft comparing with Paxos are:[3]

- The strong leader: the form of leadership is stronger than other forms of leadership in other consensus algorithms as Raft allows one-way updates. The log is replicated only from leader to the nodes.
- Raft allows one-way updates. The log is replicated only from leader to the other nodes called followers.
- Membership changes:the mechanism used for changing the set of servers in the cluster is based on a joint consensus method.

2.1. **Leader Election.** A server in the cluster can be in one of the three states (leader, follower or candidate to be a leader). By default, when the system is started a leader is selected, and a new leader is selected in case the current leader fails. The follower's servers are passive, they just respond to the requests from the leader and the candidates and apply the log changes send by the leader. All the client's requests are handled by the leader, and in the event that a client sends the request to a follower, the request is redirected to the leader.

Raft algorithm introduces in the election process the "term" concept. Raft splits time into terms of variable length, each term is identified by a consecutive integer. The election is started when the followers do not receive messages from the leader in a period of time called election timeout, they enter in candidate state and send messages to other nodes in order to be elected. Only one vote is allowed per server, and the vote uses the "first come, first served" methodology. A condition for a server to be voted - check for stale servers- is that the candidate term is greater or equal than its own (terms act as a logical clock and are sent in in the communication between servers). If a deadlock occurs (two candidates get the same number of votes), the election closes and a new election is started, each candidate initiating the selection process after a randomly timeout has passed (150-300ms)- the random timeout is used to increase the possibility that a leader is elected in the next election process. The following image displays the term representation in Raft[3]:

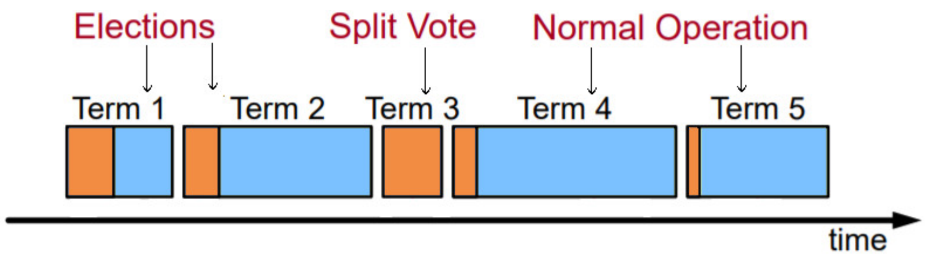

FIGURE 1. Term representation in Raft

2.2. **Log Replication.** Once a new leader is elected, it starts serving client's requests. Each request contains a command with data to be applied to the state machine. The leader appends the command to its log and starts the notification process for the other servers. After the log entry was replicated on the majority of servers, the leader applies /executes the command (the entry is committed), responds to the client and informs the followers. In case of network errors, the leader retries indefinitely, and based on the timers, the follower's logs will contain valid data. Once the follower is informed that a log entry was committed, it updates using the entry its own state machine. The inconsistencies between the leader and the follower's logs are solved by forcing the follower's log to update to the leader's log version. The following image presents the possible status of a replicated log in different nodes.

Due to the log management replication, the size of the logs is increasing in a rapid manner, so the log replicated systems and log replicated algorithms (including Raft) use compacting features. The compacting mechanism is based on the snapshot solution, where the entire system state is copied to a stable storage, and then the logs delete all the information up to that point.

Contingency measures in case of node failure, are different according to the node's role and can be structured as follows:

- if the leader node has failed, then a new election has to be performed.
- if the node's role is candidate or follower, the communication (using RPC) fails and the leader will send messages continuously until the server is back on track. At this point, the log will be updated according to the entries from the leader's log and the commands will be applied on the state machine.

3. KAFKA

Apache Kafka is a distributed streaming platform, based on a commit log that started as an internal LinkedIn project designed to provide a low-latency, high-throughput platform for manipulating real-time data feeds. Kafka was designed to be used for two classes of applications that require building real-time streaming data pipelines used to get data between other systems in a reliable manner and for applications that react or manipulate the streams of data. Kafka is used for publishing and subscribing to streams of data, to store the data in a fault-tolerant way and to guarantee the order of the messages.

Kafka runs as a cluster of servers (same as Raft), and although the size of the cluster can have any size from 1 to (probably) hundreds, Kafka uses an additional software component for metadata management. This component, called Apache Zookeeper employs a quorum-based voting for leader election and always requires an odd number of nodes. In order to survive a single node failure, the minimum number of nodes is 3. To handle a two-node failure, it requires a cluster made of 5 nodes similar to what Raft recommends - number of 5 servers for a two-node failure). Kafka stores the stream of data in topics, each topic can be split in different partitions. In order to avoid duplicate writes from the client, the Kafka client library assigns a unique Id to each client process and a sequential number to every record submitted for processing. Thus, the Kafka leader can perform duplicate checks to avoid multiple writes [6]. Zookeeper is a distributed coordination service that provides synchronization and group services. Zookeeper uses an atomic broadcast protocol called Zab that ensures data to be kept consistent while trying to achieve a high performance primary-backup system. The protocol, its correctness and its performance were analysed in papers [10, 11, 12]; however for distributed systems, in which data accuracy is mandatory, Zookeeper might face some issues. The high efficiency in Kafka is based on the fact that the follower servers serve the read requests, and forward the write requests to the leader. That implies that there is the possibility that a read request can be served by providing stale data [13]. However, Kafka plans to switch from ZooKeeper to a Raft based implementation. The plans take the form of a change request currently under development, to implement a controller quorum based on Raft for leader election [16]. Using a different algorithm, the cases when the system will return stale values for read requests will be solved and fixed and it will also remove the dependency from a single service.

Kafka topics are just a scalability and performance design decision, as the order of operations are guaranteed only for data written in a single partition. The Kafka partition is similar to Raft log. The following image presents the structure of a topic and the method to handle write requests[1].

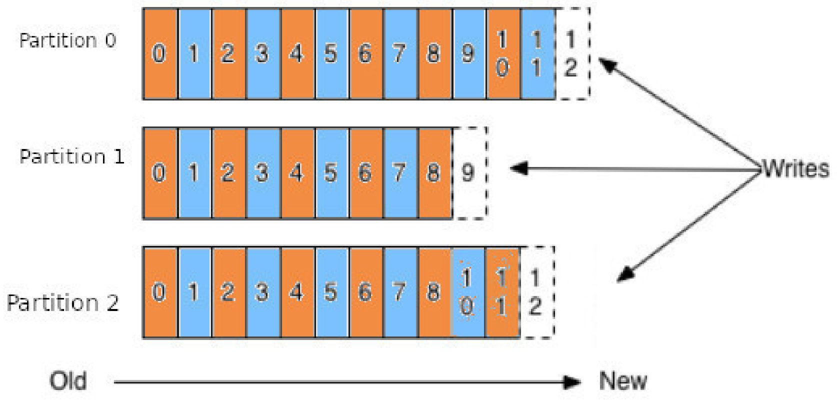

FIGURE 2. The structure of a topic [1]

Published data is written to the disk and then is replicated to ensure fault-tolerance. The producer is informed that the data was acknowledged only after it is replicated across the other server's partitions. The replication factor can be controlled for each topic.

3.1. **Leaders.** Each partition has a leader. This flexibility increases the performance, contributing to low latency and high-throughput. Kafka tries to avoid having the same leader node for a large number of partitions, so it enforces a balance mechanism. The leaders for partitions are distributed among different servers in the cluster. In Kafka, the number of replicas for each partition is configurable (so, it allows to have a different number of replicas based on a topic basis).

A node is considered to be "alive" if it is able to maintain its session with ZooKeeper and to replicate the leader's write in a reasonable time frame. The leader keeps track of all its "alive" or "in sync" nodes also known as the ISR (In-Sync Replicas) list, and when a follower falls behind or dies it is removed from the list.

When a server falls (that is not a leader for a specific partition), the leader will continue to notify and wait for that node. However, when the leader for a specific partition falls, a new server has to be selected from the followers as the leader for that partition. The constraint related to the follower is that it has to be up-to-date, as the crucial guarantee for a log replication algorithm is the following: if the client receives the commitment acknowledgment for a message, and the leader fails, that new selected leader must contain the message. The leader has to wait for a sufficient number of followers to acknowledge they received the message before the leader declares the message is committed.

For cluster coordination tasks, Kafka uses ZooKeeper as a single control broker. It stores information about partition's leaders and is responsible for selecting a new leader. In the unlikely event that all the follower's node die, two methods can be used:

- the system can wait for a follower/replica from the preferred replica set to recover and to choose it as leader (hoping that the data is valid).
- or to use the first replica that is available (even if it is not in the preferred replica set)- this method is a trade-off between not having any data or having some "older" data.

3.2. **Communication and logs.** Kafka architecture consists of a cluster having minimum 3 servers (needed for ZooKeeper), but is scalable to hundreds for Kafka (as a note, Kafka can also run on a single server). On each server there are multiple partitions that could also be split into topics. Each producer can publish to any topic, it can publish to all the partitions of the topic in a round-robin way or it can add an id/key to the message in order to publish to the same partition. Consumers on the other hand are grouped in clusters, Kafka allows multiple consumer clusters. Consumers can be separate processes or separate machines, and each record published for topic is delivered to each subscribing group. Only one customer instance is allowed for a subscribing group. The only metadata used for each consumer is the position of the consumer in the log. This approach offers a higher flexibility compared to queue messaging, as it offers each client the possibility to re-read an older message(s) or to skip others to get to the up-to-date index [4].

Kafka persists the messages for a configurable time period, and manages to perform constantly with respect to the data log size. Storing data for a long time only requires more space on the disks. However, Kafka offers a method to compact the logs - the process is based on copying the file but saving only the last value of a message.

The replication algorithm ensures that all the logs from different servers have the same offsets for the same messages and the same order for the messages. The only exception from this rule occurs in the following situation: the messages in the leader log that are in the process of replication and were not saved in the follow's files.

In order to improve performance, to reduce the number of I/O requests and the overhead of the network round-trip, Kafka implements the "message set" abstraction that groups the messages, using bulk messages whenever is possible. Kafka tries to send larger packets instead of sending smaller messages one by one, as they were published by the producers. Another method to improve performance is appending in the log in bulk mode. The servers append in the log in one go in continuous memory blocks (bulk mode) and the consumer receives a larger linear chunk every time. As a note, Kafka uses ”push” type for producer- allowing them to send as many messages as they can produce, but for customers, it works with a ”pull” method. The pull method implies that the messages are not sent to the consumers (the consumers might get overwhelmed and could not process them in a timely manner), so Kafka waits for the consumer to request for new messages.

4. KEY SIMILARITIES BETWEEN RAFT AND KAFKA

4.1. **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.

4.2. **Log replication.** Again, both Raft and Kafka rely on a consistent, ordered log of operations that is replicated from the leader node to the followers. An operation (in Raft) or a message (in Kafka) is reported to the client as committed only after it is confirmed by the followers. Both behave in the same manner: first they apply the operation/message in their own logs, then try to replicate on the other nodes, and finally wait for the nodes to confirm the writing before sending a confirmation message to the producer.

Both systems provide the same guarantees regarding the replicated log: same positions in the replicated logs will contain the same, identical information.

Both solutions support log compaction by using periodical snapshots to reduce the size of the logs.

4.3. **Failover and leader election.** Both Raft and Kafka support automatic failover by executing an election algorithm among the up-to-date follower nodes that are still alive. The actual failure detection method is different, but the high-level election method is similar. Both solutions only handle non-Byzantine failures.

In case of a follower node failure, there is no impact in operation and the restarted follower or candidate node will simply replay all the operations from the leader node until it becomes up to date.

4.4. **Additional Kafka features. Topics and Partitions** Even if Kafka is based on a replicated log, it's architecture that uses topics enables a higher degree of parallelism and efficiency as requests are split between the servers in the cluster.

The fundamental condition for a log replication-based system is that the messages persist and that are sorted in the log in the same order as they were received. In Kafka, each partition is ordered and it consists in an immutable sequence of records. Each record in the partition is uniquely identified by a sequential id number called offset (the consumer only has to remember the offset of the last record he got in order to "move" in the log and to read records from the past). A consumer is able to deliberately go back to an old offset to re-process the data. This ability, even if is not specific for queues, can bring a lot of benefits to the consumers [8].

Load balancing

A no routing tier is used in Kafka, all the requests are sent directly to the broker that is the leader of the partition, avoiding a possible bottle neck in the routing tier. In order for a producer to know which is the leader of a specific topic, the servers in Kafka must be able to answer which servers are alive and where are located the leader nodes for each partition.

Moreover, the parallelism and load balancing is increased as each topic is split into partitions, and each publisher can write on a partition based on a round-robin method or can write to a specific partition (using the exposed interface for semantic partitioning which allows the publisher to specify a key to the partition) [4]

Consumer Groups

Kafka introduce the notion of consumer groups "a consumer group" is a cluster of consumers or processes in an application. For example, if the messages in a system must be consumed by two modules or processes of the same application, these processes should be configured in separate groups. The configuration is due to the fact that a message will be sent only once for each consumer group.

Highly Configurable (number of replicas, recovery policy, batch parameters)

Designed to be addressed to a large type of applications, Kafka is quite configurable, as it offers support by parameters to configure the number of replicas for each partition, and to modify it (for example: increasing the replication factor of an existing partition). The batch mode parameters are also configurable (time of waiting or number of messages per batch), and the recovery policy for falling nodes.

Batching

Batching is a feature that contributes a lot to the efficiency of the system. Batching in Kafka can be configured, to pile up and to send no more than a specified number of messages or to wait only a specific amount of time. This allows sending out large chunks of data through the network and fewer I/O operations on the server (even if they are larger) increasing performance. As a note, in exprimental validation there was a stong correlation betweend the packet sizes and the time (packet send interval), and the time needed for the packages to fit a phase-type distribution. The performance metrics are influenced by the various configuration and by the network latency [9].

5. Conclusion and Future Work

Kafka and Raft were designed in parallel in relatively the same period (Kafka was open sourced in 2011 and came out from Apache incubator in late 2012 as Raft was the subject of a lecture in march 2013 [5]) and they have a lot of similarities. Kafka can be used as an implementation of consensus distributed algorithms and can be a solution to develop a replicated state machine.

As a note, Kafka plans to remove the ZooKeeper dependency, in order to manage metadata in a more robust and scalable way using a RAFT based algorithm. Using ZooKeeper has some drawbacks because ZooKeeper is a separate system. Unifying the systems and the configurations between Kafka and ZooKeeper would ease Kafka usage and also would improve performance. The plans take the form of a change request, currently under development: it implements a controller quorum based on Raft for leader election [2].

The future work would be to develop a benchmark test to compare the performance between a Raft implementation and Apache Kafka. Both solutions provide a set of configuration parameters which can heavily influence the performance of each implementation. As an example: the replication factor set in Kafka can influence the efficiency. The hardware can also play an important role: does the system run on physical or on virtual machines (physical machines influence network delays as virtual processing introduces other types of delays).

References

[1] Apache Kafka. http://kafka.apache.org

[2] C. McCabe, Kafka improvements proposals" (having status accepted 2020, Retrieved from https://cwiki.apache.org/confluence/display/KAFKA/KIP-500

[3] D. Ongaro, J. Ousterhout (2013), In Search of an Understandable Consensus Algorithm, USENIX ATC 14, Proceedings of the 2014 USENIX Annual Technical Conference, June 2014 Pages 305-320, Retrieved from https://web.stanford.edu/ ouster/cgi-bin/papers/raft-atc14

[4] J. Kreps, Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines), Retrieved from https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines, Retrieved

[5] J. Ousterhout, Lecture for the Raft User Study, March 2013, Retrieved from https://raft.github.io/slides/raftuserstudy2013.pdf

[6] L. Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems, 16 (2): 133-169. DOI:10.1145/279227.279229.

[7] T. D Chandra, R. Griesemer, J.Redstone, Paxos made live: an engineering perspective. In Proc. PODC'07, ACM Symposium on Principles of Distributed Computing (2007), ACM, pp. 398-407.

[8] W. Guozhang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, J. Stei, Building a Replicated Logging System with Apache Kafka, Proceedings of the VLDB Endowment, Retrieved from http://www.vldb.org/pvldb/vol8/p1654-wang.pdf, DOI:10.14778/2824032.2824063

[9] W. Han Z. Shang, K. Wolter, Performance Prediction for the Apache Kafka Messaging System, 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City, DOI: 10.1109/HPCC/SmartCity/DSS.2019.00036

[10] B. Reed, F Junqueira, A simple totally ordered broadcast protocol, Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, September 2008, Article No. 2, Pages 1-6, doi:10.1145/1529974.1529978,

[11] A. Medeiros, ZooKeeper's atomic broadcast protocol: Theory and practice, Helsinki University of Technology, 2012, Retrieved from https://www.semanticscholar.org/paper/ZooKeeper-

[12] F. P. Junqueira, B. C. Reed and M. Serafini, Zab: High-performance broadcast for primary-backup systems, 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), Hong Kong, 2011, pp. 245-256, DOI: 10.1109/DSN.2011.5958223

[13] P. Hunt, M. Konar, F. Junqueira, B. Reed, ZooKeeper: wait-free coordination for internet-scale systems, USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference, June 2010, Pages 11

[14] H. Yusuf1, I Surjandari. (2020). Comparison of Performance Between Kafka and Raft as Ordering Service Nodes Implementation in Hyperledger Fabric. International Journal of Advanced Science and Technology, 29(7s), 3549-3554. Retrieved from http://sersc.org/journals/index.php/IJAST/article/view/17652

[15] H. Yusuf, I Surjandari, Comparison of Performance Between Kafka and Raft as Ordering Service Nodes Implementation in Hyperledger Fabric, International Journal of Advanced Science and Technology Vol. 29, No. 7s, (2020), pp. 3549-3554, ISSN: 2005-4238 IJAST

[16] C. Wang, X. Chu, Performance Characterization and Bottleneck Analysis of Hyperledger Fabric, arXiv:2008.05946v1 [cs.DC]

[17] C McCabe, Kafka improvements proposals (having status accepted), 2020, https://cwiki.apache.org/confluence/display/KAFKA/KIP500

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1 Kogălniceanu St., 400084 Cluj-Napoca, Romania
Email address: `mpetrescu@cs.ubbcluj.ro`

Montran Corporation, Romania
Email address: `rpetrescu@montran.com`