European Journal of Computer Science and Information Technology
Vol.7, No.3, pp.41-52, June 2019
Published by European Centre for Research Training and Development UK (www.eajournals.org)

CONSENSUS MODELS FOR PERMISSIONED DISTRIBUTED LEDGERS

D. D. Amarasekara
Research and Development, Typefi Systems Pty Ltd, Maroochydore, Australia

D. N. Ranasinghe
University of Colombo School of Computing, Colombo, Sri Lanka

ABSTRACT: *The present study builds a taxonomy of State Machine Replication (SMR) and Transactional Replication (TR) models based on three algorithmic descriptors: the existence or the absence of a leader, synchronicity and the type of fault tolerance. As per FLP theorem, termination cannot be guaranteed in deterministic algorithms in the presence of even a single faulty process. However, the taxonomy shows that this impossibility has been overcome through weak asynchrony assumptions to ensure termination or using randomized methods to ensure termination deterministically. Increased resource usage and low throughput are major problems associated with current Byzantine Fault Tolerance (BFT) systems. Cross Fault Tolerance (XFT) is a new approach which solves practical BFT cases with minimal resources. The article further concludes with a short review of selected open source practical frameworks that implement SMR like Hyperledger, Apache Kafka, Zookeeper, Zab, etcd, zetcd and piChain.*

KEYWORDS: BFT, Paxos, Permissioned Ledgers, RAFT, SMR, TR

INTRODUCTION

A distributed ledger is a replicated data structure among many different nodes or replicas in a peer-to-peer network. A fundamental feature of a distributed ledger is that it is not maintained by any central authority. Any updates to the ledger are independently made by each node. The nodes then vote on these updates to ensure that the majority agrees with the conclusion made. Once the consensus has been reached, the most agreed upon version of the ledger is saved on each node separately.

Thus, consensus algorithms are central to the functioning of any distributed ledger as it ensures the replication across nodes. Consensus may be implemented in different ways such as through the use of proof-based algorithms including Proof of Elapsed Time (PoET) and Proof of Work (PoW) for permission-less ledgers or through the use of voting-based methods which include State Machine Replication and Transactional Replication (Hyperledger Architecture, 2017) for permissioned ledgers.

This paper aims to provide a methodical view of consensus based distributed ledger mechanisms available in the research literature and then build a taxonomy of state of the art based on three algorithmic descriptors: existence (or not) of a leader, the synchrony model and the failure model. It is clearly evident from the taxonomy i.e. Figure 1 that the majority of consensus methods which support both CFT and BFT are leader-based and they are for replicated state machines. Consensus protocols designed for Transaction Replication is quite low and they support CFT. As per FLP (Fisher, Lynch & Peterson, 1985) theorem liveness property i.e. termination cannot be guaranteed in deterministic algorithms in the presence of even one faulty process. This impossibility has been

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

overcome through weak asynchrony assumptions or by using randomized methods. Hashgraph (Baird, 2016) and Aleph (Gągol & Świętek, 2018) are such randomized methods proposed recently for replicated state machines. Both are leaderless, asynchronous and Byzantine fault tolerant. Leader-Free Byzantine (Borran & Schiper, 2010) and Leaderless Byzantine Paxos (Lamport, 2011) are such non-randomized i.e. deterministic and Byzantine fault tolerant methods developed using weak asynchrony assumptions. Further, Leaderless Byzantine Paxos uses a virtual leader. Having a consensus algorithm which is completely leaderless, asynchronous and byzantine fault tolerant seems to be another impossibility under deterministic condition. Increased resource usage and low throughput are major problems associated with current BFT systems. XFT i.e. Cross Fault Tolerance is an approach where it tries to solve practical BFT with minimal resources. XFT needs 2f+1 replicas as in CFT and supports both CFT and BFT. We conclude the article with a short review of practical open source frameworks for SMR such as Hyperledger, Apache Kafka, Zookeeper, Zab, etcd, zetcd and piChain.

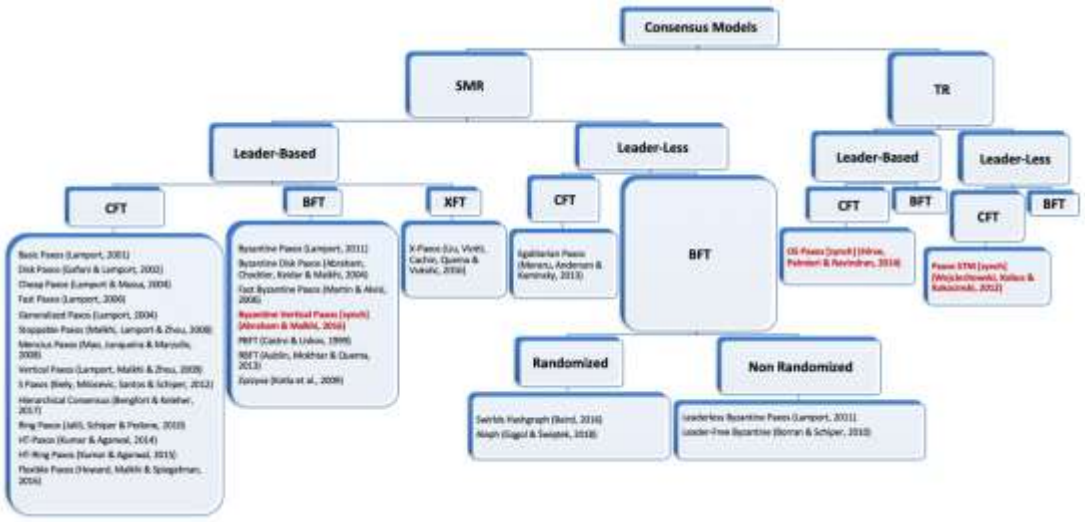

Figure 1. A Taxonomy of Applicable Consensus Models

THE ROLE OF LEADER

In leader-based protocols, the most responsibility for the tasks i.e. Request dissemination (receiving requests from clients and distributing them to all replicas), Establishing order (reaching agreement on the order of requests), Request execution (executing requests in the determined order and sending replies to clients) is with the leader while the followers are only responsible for acknowledging the order proposed by the leader and executing requests (Biely et al., 2012).

Therefore, a bottleneck exists at the leader and the maximum update throughput is limited by the leader's resources (Biely et al., 2012). For instance, with n replicas, for each command, the leader handles O(n) messages and non-leader replicas handle only O(1). Also when the leader fails the state machine becomes unavailable until a new leader is elected (Moraru, Andersen & Kaminsky, 2013). The scalability of such protocols is also poor as the number of replicas on the system increases, the leader quickly reaches a CPU limits. Leader election is a difficult task specially in an asynchronous distributed system, being similar to the consensus problem as to elect a leader as all replicas must reach consensus on the leader (De Prisco, Lampson & Lynch, 1997).

There are a few Paxos variants available in the research literature where the above shortcomings have been addressed. Ring Paxos (Jalili, Schiper & Pedone, 2010) offloads the tasks of leader by ordering

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

of identifiers of client requests (i.e. instead of client requests and dissemination of requests) through ip-multicasting and a ring of acceptors and batching of requests at the leader and use of pipelining. Still, if the leader fails, service will be interrupted until a new leader is elected. S-Paxos (Biely et al., 2012) offloads work from the leader by delegating it to the other replicas. The only remaining leader-based task is ordering. HT-Paxos (Kumar & Agarwal, 2014) offloads the leader's the work of handling client communications and client request dissemination and instead it only receives the batch identifiers (or request identifiers) and order them. HT-Paxos gives high throughput compared to S-Paxos. HT-Ring Paxos (Kumar & Agarwal, 2015) is another Paxos variant which uses the iornic concepts of Ring Paxos, S-Paxos and HT-Paxos to offload the leader's tasks further. Thus, it has higher throughput compared to all other Paxos variants. Further, many Paxos-based systems use read leases where one or several replicas can satisfy read requests locally without having to commit a more expensive Paxos protocol (Moraru, Andersen & Kaminsky, 2014) which improves read throughput and latency.

In leader-based consensus algorithms, all read and write requests are handled by the leader. This can be improved if we allow followers to serve read requests. This is because all write requests are done using majority vote. So, reading from any majority gives the correct result. Thus, the Quorum reads is also another technique used to offload the load of the leader. Hence the better performance of the followers is assured under failure-free and read-heavy workloads (Arora et al., 2017).

The other leaderless Paxos variants are leaderless Byzantine Paxos (Lamport, 2011), Egalitarian Paxos (Moraru, Andersen & Kaminsky, 2013) and Paxos STM (Wojciechowski, Kobus & Kokocinski, 2012). Egalitarian Paxos has no designated leader process. Instead, clients can choose at every step which replica to submit a command to, and in most cases the command is committed without interfering with other concurrent commands. Further, no need for leader election, as long as more than half of the replicas are available. Paxos STM is the first replicated Distributed Transactional Memory system to provide support for recovered operations within transactions. It replicates all transactional objects (objects shared by transactions) and maintains strong consistency of object replicas. Leaderless Byzantine Paxos has a virtual leader which performs the role of a leader. For Byzantine agreement, relying on a leader is problematic. Leaderless Byzantine Paxos was developed to overcome such problems whereby replacing the leader by a virtual leader using a synchronous Byzantine agreement. If the system does not behave synchronously, then the synchronous Byzantine agreement algorithm may fail, causing different servers to choose different virtual-leader messages (Lamport, 2011).

Safety and liveness of Paxos is guaranteed by the fact that any two quorums will intersect. Thus, the quorums are typically composed of any majority of replicas. However, in Flexible Paxos (Howard, Malkhi & Spiegelman, 2016), use of disjoint quorums as opposed to majority quorums is safe too. For example, in a system of 10 nodes, we can safely allow any set of only 3 nodes to participate in replication, provided that we require 8 nodes to participate when recovering from leader failure. This way, it reduces the latency as leaders will no longer be required to wait for a majority quorum to accept proposals. Hence, we will have the high throughput as the result. However, it reduces the availability as the system can tolerate fewer failures.

Hierarchical Consensus (Bengfort & Keleher, 2017) is a leader-oriented protocol which has multiple sub-quorums where each sub-quorum elects a leader to coordinate decisions. All the nodes are organized into a tiers of quorums such that parent quorums manage the decision space and leaf quorums manage access ordering. At the beginning all nodes are independent and they must self-

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

organize into a hierarchical structure. This has more leaders but less global messages exchanged hence it prevents bottlenecks and increases the throughput.

All deterministic (non-randomized) Byzantine algorithms for partially synchronous systems have so far been leader-based. However, in (Borran & Schiper, 2010) a deterministic (non-randomized), leader-free Byzantine consensus algorithm has been introduced in a partially synchronous setting. As per FLP (Fisher, Lynch & Peterson, 1985) theorem which states that no deterministic and asynchronous algorithm can satisfy liveness in the presence of at least one faulty process, the researchers had replaced determinism by introducing randomization in a form of coin tossing. Two new algorithms, called the Swirlds Hashgraph consensus algorithm (Baird, 2016) and ALEPH (Gągol & Świętek, 2018) were introduced based on this idea. Both are randomized, leaderless, asynchronous, byzantine fault tolerant replicated state machines.

Hashgraph uses a gossip protocol where the participants gossip about gossip and build a data structure which records who gossip to whom in which order. In Hashgraph it is assumed that up to n/3 of the nodes can behave arbitrarily, where n is the total number of nodes in the system. Any two honest nodes can eventually communicate after some unknown but bounded amount of time. Nodes gossip with each other constantly by selecting a peer randomly. These gossip interactions grow the Hashgraph in a consistent way. Hashgraph uses a round number which puts a given node's events in a monotonically increasing order and a binary value called "witness" which is true whenever an event is the first created by the particular node in a round. An event is famous if it is a witness and was received by most members soon after being created. Every participant has its own copy of the hashgraph. Thus, it uses virtual voting i.e. every member can reach Byzantine agreement without a single vote ever being sent. In other words, a participant can calculate what another participant's vote would have been sent to it. Hence, the number of messages exchanged when making an agreement is zero compared to a traditional Byzantine agreement protocol involved in sending votes. The algorithm guarantees that it makes progress eventually when over 2/3 of nodes continue gossiping forever but no analysis is available to see how long it takes. For example, during network partitions, random coin flips are required for progress. Hashgraph is a local coin protocol and requires expected exponential number of rounds. Hence in those situations the latency could be very high.

This issue has been addressed by Aleph which guarantees liveness with an expected constant number of rounds needed to terminate. In Aleph, it is assumed that 1/3n-1 processes are controlled by a malicious adversary, all remaining 2/3n+1 processes are correct and processes exchange information by creating and sharing with each other units which can contain arbitrary data. The total ordering of units is computed locally, based only on the partial order structure, and is guaranteed to be the same for all processes. This is very similar to Hashgraph's virtual voting. The Aleph protocol for agreement on total ordering of units satisfies both safety and liveness.

In both Hashgraph and Aleph, the consensus for total ordering is done locally and need potentially expensive computation by all nodes. Since there are no released performance metrics available, no conclusions can be made about the throughput, latency and storage requirements of these protocols. Other issue is how well the protocol responds to fail-stop and fail-restart crashes.

SYNCHRONY MODEL

Consensus algorithms based on synchronous timing are much simpler. However, they are unrealistic in practice.

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

Literature shows that the existence of both time bounds i.e. the time required for a message to be sent from one processor to another or the relative speeds of different processors is necessary to achieve any resiliency, even under the weakest type of faults. It has been proven that if a fixed upper bound on message delivery time does not exist or a fixed upper bound on relative processor speeds does not exist then there is no consensus protocol that satisfies liveness under even one fail-stop fault (Fischer et al., 1985).

The assumption that messages are delivered asynchronously is made in almost all the Paxos variants except for Leaderless Byzantine Paxos (Lamport, 2011) and Paxos STM (Wojciechowski, Kobus & Kokocinski, 2012).

There are other models available in the literature as well with regard to synchronicity. i.e. partially synchronous model and eventually synchronous model. Partially synchrony lies between complete synchronicity and complete asynchronicity. In one version of partial synchrony, fixed time bounds exist, but they are not known a priori. In another version of partial synchrony, the time bounds are known, but are only guaranteed to hold starting at some unknown time T, and protocols must be designed to work correctly regardless of when time T occurs (Dwork, Lynch & Stockmeyer, 1988). The eventual-synchrony model considers an asynchronous network that may delay messages among correct nodes arbitrarily, but eventually behaves synchronously and delivers all messages within a fixed (but unknown) time bound (Dwork, Lynch & Stockmeyer, 1988). Protocols in this model do not violate the safety property during asynchronous periods as long as the assumptions on the type and number of faulty nodes are met. When the network stabilizes and behaves synchronously, then the nodes are guaranteed to terminate. Replication protocols need to tolerate network partitions and node failures. Protocol designers today prefer eventual synchrony assumptions because of its simplicity and practitioners observe that it captures the broader coverage of actual network behaviour, especially when compared to the partially synchronous models that assume probabilistic network behaviour over time (Cachin & Vukolic, 2017).

PROCESS FAILURE MODEL

Fault tolerance is a key property expected of any distributed system. It is all about the system resiliency in terms of achieving safety and liveness in the face of node failures. There are two kinds of possible faults in a SMR-based system i.e. Crash Fault Tolerance (CFT) and Byzantine Fault Tolerance (BFT).

Crash Fault Tolerance can be assumed either as fail stop or fail restart. In the fail-stop model, processes can fail by stopping, i.e. a faulty process eventually stops executing the algorithm permanently whereas in the fail-restart model a process can resume execution after crashing. In this model, any state stored in volatile storage is lost upon a crash, while the state in stable storage is preserved (Wojciechowski & Poznan, 1983). In most of the SMR based Paxos variants, process crash by stop is assumed. However, Paxos variants like Disk Paxos (Gafani & Lamport, 2002), Fast Paxos (Lamport, 2006) and Byzantine Disk Paxos (Abraham et al., 2004) support both fail-stop and fail-restart. Paxos requires 2f + 1 replicas where f is the number of crash faults. This is because, the system should be able to execute a request without waiting for f replicas but receives a quorum of f+1 responses as they may be crashed and they do not send responses. The protocol must ensure that there will be at least one replica that processed the most recent request and which will participate on other executions there by guarantees safety.

BFT state machine replication is an efficient and effective way to address arbitrary software and

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

hardware faults. Practical Byzantine Fault Tolerant (PBFT) is such a BFT protocol for state machine replication. It uses a combination of primary and backup replicas to order and execute requests. The primary is the leader and assigns sequence numbers to requests. Backups check these numbers for consistency and monitor the leader to detect crashes or misbehaves. PBFT which requires 3f+1 replicas where f is the number of Byzantine faults. Consequently, each step of the protocol must be processed by at least 2f+1 replicas. The protocol works by client first sending a request to all replicas, second by the primary multicasts the request to the backups, third by replicas execute the request and send a reply to the client, and finally by the client waits for f+1 matching responses for the request and completes the operation (Castro & Liskov, 1999).

All the Byzantine Paxos variants have been derived from their corresponding Paxos variants. For instance, Byzantine Paxos (Lamport, 2011) was developed by byzantizing a variant of the ordinary Paxos algorithm and BVP (Abraham & Malkhi, 2016) is built from the foundations of Vertical Paxos (VP) (Lamport, Malkhi & Zhou, 2009). BFT consensus protocols assure the key properties of blockchain such as immutability and auditability (Baliga, 2017).

Comparing PBFT with Paxos, it is clear that besides using f additional replicas, PBFT requires one extra communication step where the PREPARE phase is executed for the non-leader replicas to verify if the leader has made a consistent proposal in its PRE-PREPARE message. But it is clear that the PBFT refines the Byzantine Paxos. i.e. the leader in Byzantine Paxos is the replica called primary in PBFT and the Byzantine acceptors in Byzantine Paxos are the backups of PBFT (Lamport, 2011).

Zyzzyva (Kotla et al., 2009) is another BFT state machine replication protocol executed by 3f + 1 replicas and it is based on three sub protocols i.e. agreement, view change and checkpoint. Agreement operates within a sequence of views. In each view a single replica called primary is responsible for leading the agreement sub protocol. If the current primary is faulty or the system is running slowly, then the view change sub protocol coordinates the election of a new primary or leader. The checkpoint sub protocol limits the state that must be stored by replicas and minimises the cost of performing view changes. Like other BFT replication protocols, in Zyzzyva, an elected primary server proposes an order on client requests to the other server replicas. However, the main difference in Zyzzyva is, its replicas immediately execute requests speculatively, without running an expensive agreement protocol to establish the order. If the primary is faulty then correct replicas' states may diverge. Hence, they may send different responses to a client. However, Zyzzyva preserves correctness because a correct client detects such divergence and avoids acting on a reply until the reply and the sequence of preceding requests are stable and guaranteed to eventually be adopted by all correct replicas.

Existing BFT protocols have not been able to provide high throughput when faults occur. BFT protocols which target high throughput typically use a master server. The master tells the other replicas the order in which requests should be processed. If the master is malicious, it can degrade the performance of whole system without being detected by correct servers. This problem has been addressed by Redundant Byzantine Fault Tolerant protocol (RBFT) which gives more robustness to BFT. But RBFT uses more hardware resources as the underlying BFT-SMR protocol is executed f + 1 times in parallel and it adds the PROPAGATE phase, increasing therefore latency (even during failure-free runs) and protocol complexity of the underlying BFT-SMR protocol (Milosevic, 2009). Further, the replicas of each protocol instance execute a three-phase commit protocol to order the request. Each time a node receives an ordered request from a replica of the master instance, the request operation is executed and After the operation has been executed and the node sends a REPLY message

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

to the client that is authenticated with a MAC address. When the client receives f +1 valid and matching REPLY from different nodes, it accepts the result of the execution of the request (Aublin, Mokhtar & Quema, 2013).

However, a straightforward implementation of Byzantine state machine replication on top of Fast Byzantine Paxos (FaB) requires only four rounds of communication. FaB Paxos requires $a \geq 5f+1$ acceptors, $p \geq 3f+1$ proposers, and $l \geq 3f+1$ learners; as in Paxos, each process in FaB Paxos can play one or more of these three roles. Further, the number of rounds is made to three using tentative execution, i.e. learners tentatively execute clients' requests as provided by the leader before consensus is reached. Acceptors send to clients and learners the information required to determine the consensus value, so that they can at the same time determine whether their trust in the leader was well placed (Martin, 2006).
BFT systems have additional cost in terms of resources and performance compared with CFT. An alternative approach, Cross Fault Tolerant (XFT) (Liu et al., 2016) gives the reliability of CFT SMR protocol such as Paxos and Raft. It supports BFT even with the network asynchrony as long as majority of replicas are correct and communicate synchronously.

OPTIMISTIC REPLICATION

There are two approaches to fault-tolerant replication of services (or objects) (Wojciechowski, Kobus & Kokocinski, 2012). The first is the replicated state machine (SMR) in which a client request is executed on every server. The services must be deterministic: any service replica being in the same state always produces the same effect upon the same request. SMR coordinates all servers, so that all requests are delivered and processed by every replica in the same order.

The second is transactional replication (TR) based on deferred update (also known as multi-primary passive replication). Atomic transactions are used to access critical objects; such objects are replicated on every server. The transaction's atomicity and serializability guarantee that the concurrent modifications of object replicas are propagated consistently on every server.

We can identify four phases in both SMR and TR protocols: 1) Sending a client request, 2) Replica coordination (in SMR only) 3) Request CPU processing (i.e. local execution), 4) Agreement coordination (in TR only) 5) Sending a system response (or answer) (Wojciechowski, Kobus & Kokocinski, 2012).

Execution dominated workloads are handled much better when using TR since it can (inherently) execute multiple requests concurrently, in contrast to classical SMR. In particular, TR allows higher throughput than SMR for read-write requests with a majority of read operations that do not cause conflicts (Wojciechowski, Kobus & Kokocinski, 2012).

OPEN SOURCE FRAMEWORKS

JPaxos

JPaxos (Konczak et al., 2011) is an open source project available at https://github.com/JPaxos/JPaxos and is a Java library and runtime system, implementing Paxos for replica coordination. It supports the fail-restart model of failure and does not support byzantine faults. Further, many research projects have re-used JPaxos, including PaxosSTM for agreement coordination among the replicas.

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

Raft

Raft (Ongaro & Ousterhout, 2014) is more comprehensible SMR and a leader based protocol than Paxos and has a variant BFTRaft (Copeland & Zhong, 2014) which supports Byzantine Fault Tolerance as well. The software is in the Haskell and is available at https://github.com/chrisnc/tangaroa. In Raft, there is only one leader and all the other servers are followers. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). Raft servers communicate using remote procedure calls (RPCs), it is synchronous and it uses timeouts when communicating. In Raft, nodes can accommodate fail re-start.

BFTRaft (Copeland & Zhong, 2014) holds safety and liveness properties of Raft in the presence of Byzantine failures, while also aiming towards to Raft's main goal of simplicity and understandability. It is written in Haskell and is available at https://github.com/chrisnc/tangaroa.

Hyperledger

Hyperledger (Hyperledger Architecture, 2017) is a fast-growing open source project hosted by the Linux Foundation, to create enterprise grade and distributed ledger frameworks code bases. Hyperledger Fabric (Hyperledger Architecture, 2017) is a foundation for developing applications or solutions with a modular architecture as it allows components, such as consensus and membership services, to be plug-and-play. In Fabric, Apache Kafka (Apache Kafka, 2017) has been used as a reference implementation. Hyperledger Indy (Hyperledger Architecture, 2017) uses RBFT (Aublin, Mokhtar & Quema, 2013) as the consensus model and Hyperledger Iroha (Hyperledger Architecture, 2017) uses a voting-based approach to consensus that provides fault tolerance i.e. CFT and BFT, and finality within seconds. Both Hyperledger Indy and Hyperledger Iroha provide Byzantine fault tolerance and take more time to reach consensus when there are more nodes exist on the network. PoET in Hyperledger Sawtooth (Hyperledger Architecture, 2017) uses a proof-based approach for consensus. The advantage of proof-based algorithms is that they can scale to a large number of nodes since the winner of the proof proposes a block and transmits it to the rest of the network for validation. This provides both scalability and Byzantine fault tolerance. Further, all consensus mechanisms used in Hyperledger framework i.e. Indy, Iroha and Sawtooth are leader-based.

Apache Kafka

Apache Kafka is a distributed streaming platform (Apache Kafka, 2017). The Kafka cluster stores streams of records in categories called topics. For each topic, the Kafka cluster maintains a partitioned log. A Kafka partition represents a distributed replicated log. i.e. The log for each topic's partition is replicated among a configurable number of servers. This is how Kafka ensures the CFT. With a majority vote for both the commit decision and the leader election and with 2f+1 replicas, if f+1 replicas are to receive a message prior to a commit being declared by the leader, and if we elect a new leader by electing the follower with the most complete log from at least f+1 replicas, then, with no more than f failures, the leader is guaranteed to have all committed messages. Apache Kafka uses a different approach to choosing its quorum set. Instead of simple majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are eligible to be the leader. A write to a Kafka partition is not considered committed until all in-sync replicas have received the write. In practice, to tolerate f failures, both the majority vote and the ISR approach will wait for the same number of replicas to acknowledge before committing a message. Apache Kafka uses Zookeeper for leader election of Kafka Brokers and Topic Partition pairs, to see which brokers are alive, and for topic configuration.

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

Kafka has replication between brokers, but by default it is asynchronous. Thus, it suffers from the problems endemic to achieving consistency on top of an asynchronous replication system (Rodeo, 2016).

Kafka is also not flexible. Kafka suffers significant bottlenecks as the number of queues increase: leader reassignment latency increases dramatically and memory requirements increase significantly (Rodeo, 2016). The ability to commit without the slowest servers in ISR is impossible. This is because, any write to a Kafka partition is not considered committed until all in-sync replicas have received the write (Apache Kafka, 2017).

Zookeeper and Zab

Zookeeper is a primary-backup system where the primary process which executes clients requests. It uses Zookeeper atomic broadcast (Zab) to trigger the state changes to backup processes. Zab is a crash-recovery atomic broadcast protocol used by Zookeeper. When primary crashes, all backup processes execute a recovery protocol to agree on a common consistent state and to elect a new primary. To be elected as the new primary, it needs the support of a quorum of processes. Compared to Paxos, Zab has an additional phase called synchronization. Any elected leader tries to become established by executing a read phase as known as discovery phase which is followed by a synchronization phase. Once synchronization phase is done, Zab executes a write phase which is similar to the Paxos (Junqueira, Reed & Serafini, 2011).

etcd

etcd is a distributed coordination framework like Zookeeper whose main focus is to provide distributed consensus. It is a distributed and reliable key-value store which uses the Raft consensus algorithm to manage a highly-available replicated log. The etcd's data model and client protocol is incompatible with Zookeeper applications. This problem has been addressed by zetcd which is a proxy which sits in front of an etcd cluster and under the hood, zetcd translates Zookeeper's data model to fit etcd APIs. This allows unmodified Zookeeper applications to run on top of etcd. The source code for both etcd and zetcd can be found at https://github.com/etcd-io/etcd and https://github.com/etcd-io/zetcd respectively.

piChain

piChain (Burchert & Wattenhofer, 2018) is an open source RSM library which supports CFT only and which is for replicated state machines. It inherits features of Paxos and features of a blockchain to provide both consistency and simplicity. piChain does not have an explicit leader election subroutine as in Raft and it provides scalability and availability.

CONCLUSIONS

Distributed consensus can be implemented in two ways, through the use of proof-based schemes or through the use of voting-based methods such as State Machine Replication (SMR) and Transactional Replication (TR). Voting based consensus is applicable to closed systems and is expected to tolerate both Byzantine faults and Crash faults. Paxos is a well-known core consensus protocol used in Replicated State Machines and its liveness is guaranteed under weak synchrony assumptions, i.e. it requires a majority of non- faulty processes to ensure progress. A majority of Paxos variants which support either CFT or BFT or both are leader based. As the designated leader is a single point of failure and a bottleneck, the system throughput is limited by the leader's resources. The message overhead is significant in voting-based algorithms and it becomes worse when it tries to support Byzantine fault tolerance which results in poor scalability. As per the FLP theorem, termination

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

cannot be guaranteed in asynchronous deterministic algorithms in the presence of even one faulty process. This impossibility has been overcome through weak asynchrony assumptions to ensure the termination deterministically or by using randomized methods. Achieving all three features, i.e. being leaderless, asynchronous and BFT seemed to be an impossible task deterministically.

References

Abraham, I., Chockler, V. G., Keidar, I. & Malkhi, D. (2004), Byzantine disk paxos: optimal resilience with byzantine shared memory, PODC 2004: 23rd annual ACM symposium on Principles of distributed computing, St. John's, Newfoundland, Canada, (pp. 226-235).

Abraham, I. & Malkhi, D. (2016), BVP: Byzantine Vertical Paxos.

Apache Kafka, (2017). Retrieved August 25, 2018, from https://kafka.apache.org/documentation/#replication

Arora, V., Mittal, T., Agrawal, D., Abbadi, A., Xue, X., Zhiyanan & Zhujianfeng (2017), Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols, 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17).

Aublin, P., Mokhtar, S. B. & Quema V. C. (2013), RBFT: Redundant Byzantine Fault Tolerance. Paper presented at the meeting of the 2013 IEEE 33rd International Conference on Distributed Computing Systems, Philadelphia, PA, USA.

Baird, L. (2016), The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance. Retrieved September 13, 2019, from https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

Baliga, A. (2017), Understanding Blockchain Consensus Models, Persistent Systems Ltd.

Bengfort, B. & Keleher, P. (2017), Brief Announcement: Hierarchical Consensus.

Biely, M., Milosevic, Z., Santos, N. & Schiper, A. C. (2012), S-Paxos: Offloading the Leader for High Throughput State Machine Replication. Paper presented at the meeting of the 2012 IEEE 31st Symposium on Reliable Distributed Systems, Irvine, CA, USA.

Borran, F. & Schiper, A. (2010), A Leader-Free Byzantine Consensus Algorithm.

Burchert, C. & Wattenhofer, R. (2018), piChain: When a Blockchain meets Paxos. Retrieved February 17, 2019, from https://doi.org/10.4230/lipics.opodis.2017.2

Cachin, C. & Vukolic, M. (2017), Blockchains Consensus Protocols in the Wild. Retrieved August 11, 2018, from https://arxiv.org/pdf/1707.01873.pdf

Castro, M & Liskov, B. (1999), Third Symposium on Operating Systems Design and Implementation, New Orleans, USA.

Copeland, C. & Zhong, H. (2014), Retrieved August 11, 2018, from http://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf

De Prisco R., Lampson B., Lynch N. (1997), Revisiting the Paxos algorithm. In: Mavronicolas M., Tsigas P. (eds) Distributed Algorithms. WDAG 1997. Lecture Notes in Computer Science, vol 1320. Springer, Berlin, Heidelberg

Dutta, P., Guerraoui, R. & Lamport, L. (2005), How Fast Can Eventual Synchrony Lead to Consensus, Retrieved October 25, 2018, from https://doi.org/10.1109/DSN.2005.54

Dwork, C., Lynch, N. & Stockmeyer, L. (1988), Consensus in the Presence of Partial Synchrony, Journal of the Association for Computing Machinery, vol. 35, no. 2, (pp. 288-323).

Fisher, M., Lynch, N. & Paterson, M. Impossibility of distributed consensus with one faulty process, Journal of ACM, Volume 32 Issue 2, 1985, 374-382.

Gafani, E. & Lamport, L. (2002), Disk Paxos. Paper presented at the meeting of the 14th International Conference on Distributed Computing, Springer-Verlag London, UK, (pp. 330-344).

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

Gągol, A. & Świętek, M. (2018), Aleph: A Leaderless, Asynchronous, Byzantine Fault Tolerant Consensus Protocol.

Gupta, M. (2017), Blockchain For Dummies, IBM Limited Edition, John Wiley & Sons, Hoboken, NJ, (pp. 25-30).

Hire, S., Palmieri, R. & Ravindran B. (2014), HiperTM: High Performance, Fault-Tolerant Transactional Memory, ICDCN 2014:15th International Conference on Distributed Computing and Networking, Springer-Verlag New York, Inc. New York, NY, USA, vol. 8314, (pp. 181-196).

Howard, H., Malkhi, D. & Spiegelman, A. (2016), Flexible Paxos: Quorum intersection revisited.

Hunt, P., Konar, M, Junqueira, F., & Reed, B. (2010), ZooKeeper: Wait-free Coordination for Internet-scale Systems.

Hyperledger Architecture (2017), vol. 1. Retrieved August 11, 2018, from https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf

Jalili, P., & Schiper, N. & Pedone, F. (2010), Ring Paxos: A high-throughput atomic broadcast protocol. Proceedings of the International Conference on Dependable Systems and Networks.

Junqueira, F. P., Reed, B. C. & Serafini, M. (2011), Zab: High-performance broadcast for primary-backup systems.

Konczak, J., Santos, N., Zurkowski, T., Wojciechowski, P. & Schiper, A. (2011), JPaxos: State machine replication based on the Paxos protocol. Retrieved August 11, 2018, from http://www.cs.put.poznan.pl/pawelw/pub/EPFL-REPORT-167765.pdf

Kotla, R., Alvisi, L., Dahlin, M., Clement, A. & Wong, E. (2009), Zyzzyva: Speculative Byzantine Fault Tolerance.

Kumar, V. & Agarwal, A. (2014), HT-Paxos: High Throughput State-Machine Replication Protocol for Large Clustered Data Centers

— (2015), HT-Ring Paxos: Theory of High Throughput State-Machine Replication for Clustered Data Centers.

Lamport, L. (2001), Paxos Made Simple. Retrieved August 11, 2018, from https://www.cs.columbia.edu/~du/ds/assets/papers/paxos-simple.pdf

— (2004), Generalized Consensus and Paxos. Retrieved August 11, 2018, from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf

— (2006), Fast Paxos. Retrieved August 11, 2018, from https://doi.org/10.1007/s00446-006-0005-x

— (2011), Byzantizing Paxos by Refinement, In: Peleg D. (eds) Distributed Computing, DISC 2011, Lecture Notes in Computer Science, vol. 6950, Springer, Berlin, Heidelberg.

— (2011), Leaderless Byzantine Paxos, Springer, DISC 2011: 25th International Symposium, (pp. 141-142).

Lamport, L., Malkhi, D. & Zhou, L. (2009), Vertical paxos and primary-backup replication, PODC 2009: 28th ACM symposium on Principles of distributed computing, Calgary, AB, Canada, (pp. 312-313).

Lamport, L. & Massa, M. (2004), Cheap Paxos, DSN 2004: International Conference on Dependable Systems and Networks, IEEE Computer Society Washington, DC, USA, (pp. 307).

Lamport, L., Shostak, R. & Pease, M. (1982), The Byzantine Generals Problem ACM Trans. Program. Lang. Syst., vol. 4, no. 3.

Liu, S., Viotti, P., Cachin, C., Quema, V. & Vukolic, M. (2016), XFT: Practical Fault Tolerance beyond Crashes. Paper presented at the meeting of the OSDI 2016: 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA.

Malkhi, D., Lamport, L. & Zhou, L. (2008), Stoppable Paxos. Retrieved August 11, 2018, from https://www.microsoft.com/en-us/research/wp-content/uploads/2008/04/stoppableV9.pdf

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)

Mao, Y., Junqueira, P. & Marzullo, K. (2008), Mencius: building efficient replicated state machines for WANs, OSDI 2008: 8th USENIX conference on Operating systems design and implementation, San Diego, California, (pp. 369-384).

Martin, J. & Alvisi, L. (2006), Fast Byzantine Consensus, IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 3, (pp. 202-215).

Milosevic, Z. (2009), Abstractions for Solving Consensus and Related Problems with Byzantine Faults. Retrieved September 1, 2018, from https://infoscience.epfl.ch/record/196604/files/EPFL_TH5975.pdf

Moraru, I., Andersen, D. G. & Kaminsky, M. (2013), There Is More Consensus in Egalitarian Parliaments, SOSP, Farmington, Pennsylvania, USA.

— (2014), Paxos Quorum Leases: Fast Reads Without Sacrificing Writes.

Ongaro, D. & Ousterhout, J. (2014), In Search of an Understandable Consensus Algorithm. Paper presented at the meeting of the ATC 2014: 2014 USENIX Annual Technical Conference, Philadelphia, PA.

Rodeo (2016), Apache Kudu as a More Flexible and Reliable Kafka-style Queue, blog post, 24 January. Retrieved August 25, 2018, from http://blog.rodeo.io/2016/01/24/kudu-as-a-more-flexible-kafka.html.

Todd, S. (2018), SBFT: VMware's Blockchain Consensus Algorithm. Retrieved November 11, 2018, from https://stevetodd.typepad.com/my_weblog/2018/04/vmwares-blockchain-consensus-algorithms.html.

Wojciechowski, P., Poznan, J. K. (1983), A Formal Model of Crash Recovery in a Distributed System, IEEE Transactions on Software Engineering, vol. 9, no. 3, (pp. 219-228).

Wojciechowski, P., Kobus, T. & Kokocinski, M. C. (2012), Model-Driven Comparison of State-Machine-based and Deferred-Update Replication Schemes, 2012 31st International Symposium on Reliable Distributed, Poznan, University of Technology, Poland.

— (2016), State-Machine and Deferred-Update Replication: Analysis and Comparison.

Print ISSN: 2054-0957 (Print), Online ISSN: 2054-0965 (Online)