Threshold Logical Clocks

Manuel Vidigueira

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

June 2019

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Ceyhun Alp
EPFL / DEDIS

Contents

Abstract

In this work, we analyze a novel *threshold logical clock* (TLC) abstraction that seeks to close the gap between synchronous and asynchronous protocols, keeping the simplicity of the former while providing the stronger guarantees of the latter. TLC provides an illusion of synchronicity by allowing subsets of nodes to make progress as a group, despite operating in a fully asynchronous and/or byzantine network. We focus on the particular application of TLC for the problem of asynchronous consensus, and give an overview of how it can be implemented in a simple and efficient way with a modular architecture. Lastly, we compare the performance of different implementations of TLC and present the results of our experiments.

1 Introduction

We introduce Threshold Logical Clocks (TLC), a novel abstraction whose purpose is to decouple communication logic from asynchronous protocols, and provide the illusion of a synchronous network. TLC is based on lamport timestamps, as well as its more powerful generalization, vector clocks. However, both lamport timestamps and vector clocks fail to capture the idea of group progress. They can be used to create causal history of messages, but nodes can still remain out of sync, with vast discrepancies in their local logical clocks. Furthermore, they completely break down in the presence of malicious nodes. On the other hand, TLC implements logical time in a Byzantine setting, and forces nodes to advance time in synchronized *groups*.

We find that TLC's illusion of synchronicity makes it simpler to build many useful asynchronous protocols, such as threshold signing, threshold randomness, and consensus. Consensus protocols in particular are often delicate and complex despite numerous attempts to simplify or reformulate them [1, 2, 3] especially in the presence of Byzantine node failures [4, 5, 6, 7] and asynchronous network conditions [8, 9, 10]. We give an overview of a modular approach to consensus where each participant shares their proposal through TLC (e.g., a block in a blockchain), and then repeatedly gossips what they receive for a number of TLC rounds. We find this approach to be simpler, while remaining fairly efficient under the strongest network and adversarial models possible for consensus.

The author's semester project was to explore concepts around simplifying design and analysis of consensus algorithms by assuming the existence of a logical clock (TLC). The outline of the report is as follows: Section 2 introduces the main primitives we build on and related work in the area. Section 3 gives an overview of consensus, defines the model and theoretical concepts, and presents a solution on top of a generalized clock abstraction. Section 4 details and analyzes the Threshold Logical Clock abstraction and Section 5 evaluates its performance. The report concludes with future work.

2 Background and Related Work

In this section, we introduce the basic systems and notions we build upon and present a brief overview of the related work on consensus.

Logical clocks. Lamport timestamps [11] are a way to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, they are used to provide a partial ordering of events with minimal overhead. Vector clocks [12, 13] build upon this notion to detect causality violations. This is one of the main inspirations of TLC.

Threshold cryptographic signatures. Threshold signing [14] is a (t,n) type protocol where each party holds a share of the signing key and k cooperating parties together can generate a signature. In a non-interactive threshold signature scheme, each party outputs a signature share upon request and there is an algorithm to combine k valid signature shares to constitute a valid signature. This scheme can be used by honest nodes to validate that a message has been witnessed by t nodes in constant time complexity.

Threshold cryptographic randomness. A common coin is a random source observable by all participants but unpredictable for an adversary [9]. This abstraction is widely used in many distributed systems, particularly in Byzantine settings. RandHound [15] is a scalable, secure multi-party computation protocol that provides unbiasable, decentralized randomness in a Byzantine setting, and which can effectively implement a common coin.

Accountable state machines. Accountable virtual machines [16] can record non-repudiable information that allows auditors to subsequently check whether the software behaved as intended. PeerReview [17] more generally ensures that Byzantine faults whose effects are observed by a correct node are eventually detected and irrefutably linked to a faulty node. This makes it impossible for malicious nodes to pretend to not have received a message without it being detected.

Consensus. Asynchronous and Byzantine consensus has been the subject of research for a long time [18, 19, 8, 9]. Perhaps the most fundamental result in the field is known as the FLP impossibility [20], which proves that deterministic asynchronous consensus is impossible. Since then, research has either been focused on weaker network models [4, 1, 21], or on randomized algorithms [9, 10, 22]. We explore the latter.

3 Threshold Logical Clocks

In this section we present Threshold Logical Clocks (TLC), a synchronization primitive used to simplify round based asynchronous behaviour. It builds on the ideas of Lamport timestamps and vector clocks, but has the crucial difference that time only advances if *multiple* nodes in a group are in sync. It was designed as a primitive for asynchronous protocols in general, and can work even in the strongest adversarial settings (i.e. Byzantine).

3.1 Design goals

Many asynchronous algorithms operate with a notion of *rounds*. In a round t, each node broadcasts a message and waits until it has received some threshold T_m of messages from other nodes, before moving on to the next round and repeating this process. This threshold is usually set to guarantee desired liveness and safety properties, and is dependent on the threat model and assumptions made for the system.

The main design goal of TLC for round based communication is to provide predictable, semi-synchronized behaviour, even in the presence of failures. It preserves the notion of round as expressed above by appending a numerical value to every message broadcast or delivered, and restricts nodes by allowing (and expecting) only one message from them every round.

This model should fit fault-tolerant decentralized systems particularly well since the protocol and message communication pattern for each node is typically symmetric, as the goal is to tolerate as many failures as possible.

We now discuss an early set of potential properties for TLC that approximate these design goals. The properties presented here are simplified for exposition.

3.1.1 Properties

Define a (T_m, T_a) *threshold set* to be any set that includes T_m messages broadcast in the same round, where each has at least T_a different acknowledgements also from the same round. Under a TLC instance with parameters (T_m, T_a), we refer to it simply as a *threshold set*.

A potential set of properties for TLC is:

- **TLC1: Validity:** If every correct process p broadcasts a message m at round t, then every correct process q will eventually deliver a threshold set for round t.
- **TLC2: No round duplication:** No correct process delivers more than one message for the same process and round.

- **TLC3: Limited round broadcast:** No correct process broadcasts more than one message per round.
- **TLC4: Authenticity:** If a correct process p delivers a message m with source q and broadcast round t, then m was previously broadcast by q with round t.
- **TLC5: Forced round increase:** If a correct process p delivers (or broadcasts) a message m with round t, and process p has previously delivered a threshold set with round t', then $t > t'$.
- **TLC6: Pacing:** If a correct process p delivers (or broadcasts) a message m with round t, then p has delivered a threshold set in every previous round.

In order words, validity captures the notion of *liveness*, even in the presence of failures. If all correct nodes act in a symmetric way (as TLC is supposed to be used) and the adversarial assumptions hold, then TLC ensures that progress will be made (the delivery of threshold sets).

The second and third properties formalize the idea that *correct* nodes are only supposed to send one message per round. The fourth property, authenticity, is a naturally desirable property for broadcast primitives.

The last two properties are what make TLC a group synchronization primitive. TLC5 intuitively forces nodes to advance their clock time in order to communicate more information. This can be used directly to imply a causal ordering between messages: for any two messages delivered or broadcast by the same correct node, the one with the lowest round time is the oldest one. TLC6, on the other hand, keeps nodes from advancing time *too* rapidly. As honest nodes are have to participate and receive messages from every previous round, some types of byzantine behaviour become inexcusable, like claiming to not have seen certain messages. They can't just "bury their heads in the sand" and pretend to not have heard the news.

3.2 Protocol definitions

There are two main parameters that define any given TLC instance: the message threshold, T_m, and the acknowledgement threshold, T_a. The first one represents the minimum number of messages a node has to collect before moving on to the next round. The acknowledgement threshold, on the other hand, represents the minimum number of acknowledgements from other nodes each of those messages must have. We now present two potential implementations of TLC: a simpler "naive" version, and another leveraging threshold signing with reduced communication complexity.

3.2.1 Simple TLC

Each TLC node $i \in \mathrm{N}$ maintains a local integer variable that acts as a logical counter (or clock), initially zero. The value stored in that clock at any time represents the current round that node is in. During a normal execution of the protocol, each (honest) node is assumed to broadcast exactly one message per round, for however many rounds the protocol is supposed to operate.

For any given round, a node executes the following steps:

1. **Broadcast**

   1) Wait until there is a new message, m, to broadcast.
   2) Append the current round value, b, and broadcast the message (m, b) to all peers, properly signed.
   3) Start receiving messages (including any that were buffered).

2. **Collect**

   Whenever receiving a normal message m, for that message:

   1) Check that its content and format are correct, and that it is correctly signed. If validation fails, the message is ignored.
   2) Check that no other message from the same peer and for the same broadcast round has been received before. Since correct nodes are only supposed to broadcast one message per round, this constitutes proof of malicious behaviour, and can be recorded.
   3) Check that the broadcast round is equal to the current round. If it is greater, the message is buffered until this condition applies. If it is smaller, the message is discarded.
   4) Broadcast a special ACK message to all peers, properly signed, containing the signature of m and the current round (i.e. $(m, b)_{sig}$).

   These steps are executed strictly in this order.

   Once the node holds a threshold $\mathrm{T_m}$ of messages where each, simultaneously:

   - passed all verification tests (made it past step 4) .
   - has broadcast round equal to the current round.
   - has at least $\mathrm{T_a}$ acknowledgements from other peers.

   then it stops processing messages (buffering them instead), and moves on to the next phase.

3. **Deliver**

    1) Delivers to the upper layer all the messages that passed all tests (step 4 of the Collect phase), appending to each:
        - the current round, b.
        - the corresponding set of ACK messages.
    2) Increments its logical clock by 1 (moving on to the next round).

The whole process then repeats for the next round.

Note that it is up to the upper layer to provide the TLC abstraction with the means to correctly validate a message's content and format for a specific round. This could be done by passing a function as a parameter, for example, or directly coding it into the TLC implementation. In our specific implementation we opted for the first, as it preserves generality.

3.2.2 Threshold Witnessed TLC

Threshold Witnessed TLC differs from Simple TLC in one key aspect: instead of nodes broadcasting and collecting acknowledgments for every message, it is up to the original sender of the message to collect and relay these acknowledgements. After a node broadcasts its message, other nodes send acknowledgements directly to him (each carrying a signature of the message). The sender then relays the now certified message (collected signatures appended) to the other nodes. By itself, this change reduces the number of messages from $O(n^3)$ to $O(n^2)$. Furthermore, by utilizing previously mentioned collective signature schemes, we can reduce the size of this final message from $O(n)$ ($O(n)$ different signatures) to a constant factor. As we show in the complexity analysis, this reduces the communication complexity of TLC from $O(n^3)$ to $O(n^2)$.

Most of the algorithm remains identical to the previous one. We focus here on the changes instead, omitting most identical steps.

For any given round, a node executes the following steps:

1. **Broadcast** (identical)
2. **Collect**

    Whenever receiving a normal message m, for that message:

    1) 2) and 3) are identical.
    4) **Send a special ACK message only to the sender of the message**, properly signed, containing the signature of m and the current round (i.e. $(m, b)_{sig}$).

Upon receiving T_a different acknowledgements with valid signatures for the message m:

1) Produce an aggregate signature, using the individual signatures and threshold signing.
2) Rebroadcast the (certified) message, appending the aggregate signature.

Once the node holds a threshold T_m of *certified* messages where each, simultaneously:

- has broadcast round equal to the current round.

then it stops processing messages (buffering them instead), and moves on to the next phase.

3. Deliver

1) Delivers to the upper layer all the messages that passed all tests (step 4 of the Collect phase), appending to each:
   - the current round, b.
   - **the aggregate signature, if the message is certified.**
2) Increments its logical clock by 1 (moving on to the next round).

The whole process then repeats for the next round.

3.2.3 Complexity analysis

We analyze the two previous algorithms with respect to message complexity and communication complexity. We assume that message payloads, single signatures and the collective signature (aggregate) are of constant size $O(1)$, which is reasonable using the provided signing schemes.

Simple TLC:

- Broadcast phase: n messages of O(1) size for each of n nodes. $O(n^2)$ message and communication cost.
- Collect phase. n acknowledgements of size O(1) for each of the n^2 messages. $O(n^3)$ message and communication cost.

Threshold Witnessed TLC:

- Broadcast phase: $O(n^2)$ message and communication cost (identical).
- Collect phase. 1 acknowledgement of size O(1) for each of the n^2 messages. n messages of O(1) size (original message and aggregate signature) for each of n nodes. $O(n^2)$ message and communication cost.

For the second protocol, the use of aggregate signatures is crucial to achieve communication complexity of $O(n^2)$. Otherwise, the size of the certified messages in the collect phase would be $O(n)$, resulting in $O(n^3)$ communication complexity (even if message complexity remains $O(n^2)$).

4 Evaluation

This section experimentally evaluates our prototype implementations of both Simple and Threshold Witnessed TLC. Our goal is to assess the performance of the two versions and confirm our complexity analysis, in order to get an intuition on what the performance of our consensus algorithm is. We start with some details about the implementation itself, followed by our experimental setup and results.

4.1 Implementation

We implemented TLC in Go based on the EPFL DEDIS lab's Cothority framework. The Threshold Witnessed version also uses the kyber cryptographic library for the collective signing. The implementation resulted in around 420 lines of Go code for the simple version, and around 575 lines for the threshold witnessed version.

4.2 Experiments

4.2.1 Experimental Setup

We ran our experiments on DeterLab using 10 physical machines, each equipped with Intel Xeon processors (dual, 3 GHz) and 2 GB of RAM. We set network bandwidth to 100 Mbps and the round-trip time to 100ms, and nodes were evenly distributed through the machines. We experimented with different system sizes ranging from 11 nodes to 101. We set the size of shared messages to 1 KB. All values are averages measured over 10 consecutive rounds of TLC.

Bandwidth and Round Time. For our simulations we are mostly interested in bandwidth usage and average round time.

Figures 1 and 2 show these statistics over varying system sizes set for the non-Byzantine case ($\lceil \frac{n+1}{2} \rceil$ thresholds), while Figures 3 and 4 cover the Byzantine case ($\lceil \frac{2n+1}{3} \rceil$ thresholds).

The results confirm our expectations: approximately, the total bandwidth usage for Simple TLC is cubic in the number of nodes, while Threshold Witnessed TLC increases with the square of the number of nodes. We can see this become increasingly relevant with the larger system sizes. Simple TLC's performance starts quickly starts degrading around 51-61 nodes,

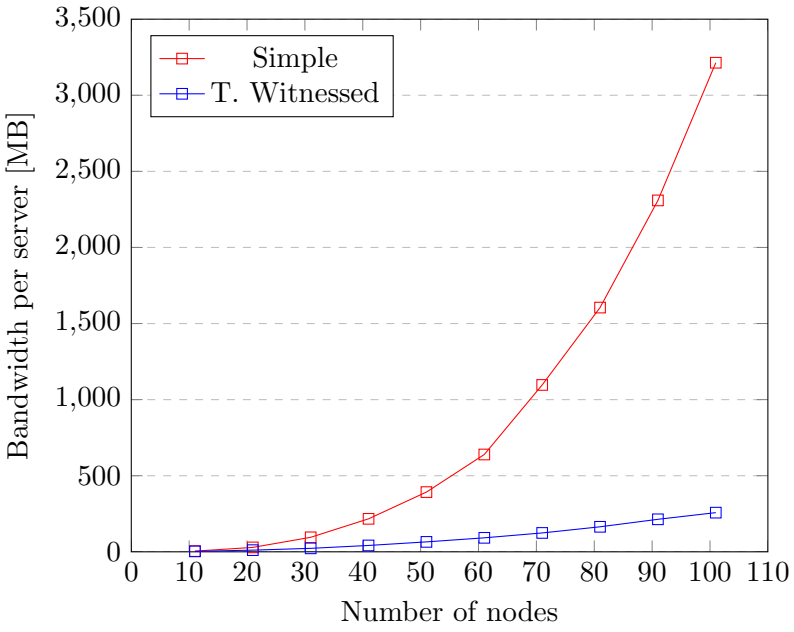

Figure 1: Bandwidth cost per number of nodes, $T_m = T_a = \lceil \frac{n+1}{2} \rceil$

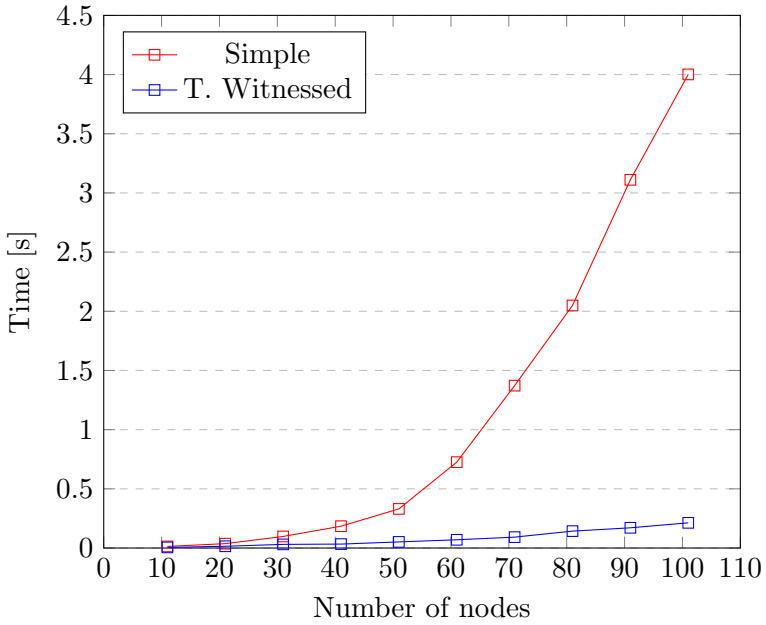

Figure 2: Average round time per number of nodes, $T_m = T_a = \lceil \frac{n+1}{2} \rceil$

reaching 1 second on average per round at 61 nodes, and 4 seconds at 101 nodes. On the other hand, Threshold Witnessed TLC stays in the low hun-

dreds of milliseconds per round: even at 101 nodes its average round time is around 210 ms.

The different thresholds don't seem to significantly impact the performance of either version, which is expected as the communication patterns remain the same.

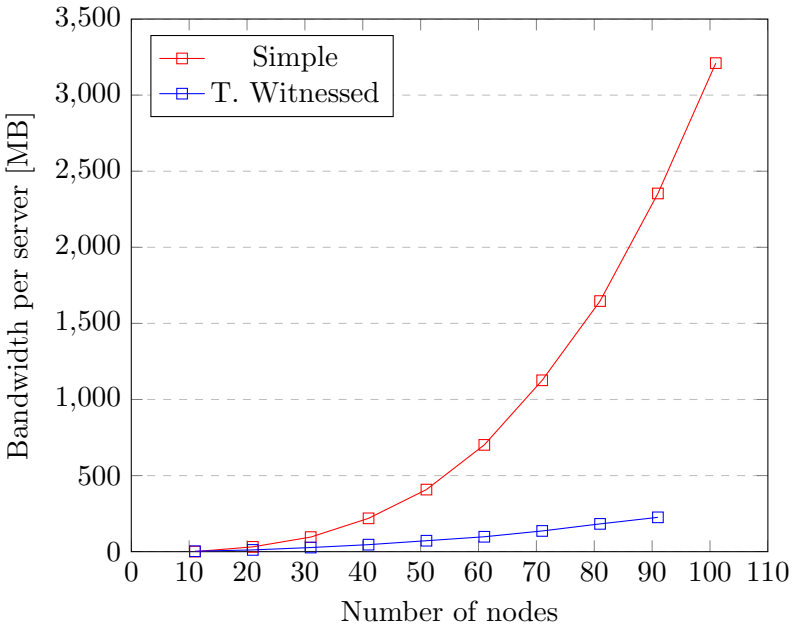

Figure 3: Bandwidth cost per number of nodes, $T_m = T_a = \lceil \frac{2n+1}{3} \rceil$

5 Application to Consensus

One of the main problems that motivate our work is finding a simple yet efficient solution to the problem of Asynchronous Byzantine Consensus (ABC). ABC consists of multiple nodes each proposing their own value until, eventually, all honest nodes decide and agree on the same value. As simple as the problem might sound, fault tolerant consensus has been subject of research for over 40 years. Deterministic consensus in an asynchronous network, for example, is impossible with even one faulty node [20].

The advent of new and more scalable solutions like Proof of Work [21] might lead to believe that consensus has, in practice, been solved. However most of these systems either suffer from security issues [23], relax their network and/or threat models, or offer only a partial solution to classical consensus. Moreover, despite the scaling issues of ABC due to its $\Omega(n^2)$ bound on communication complexity [24], it is still very useful as a primi-

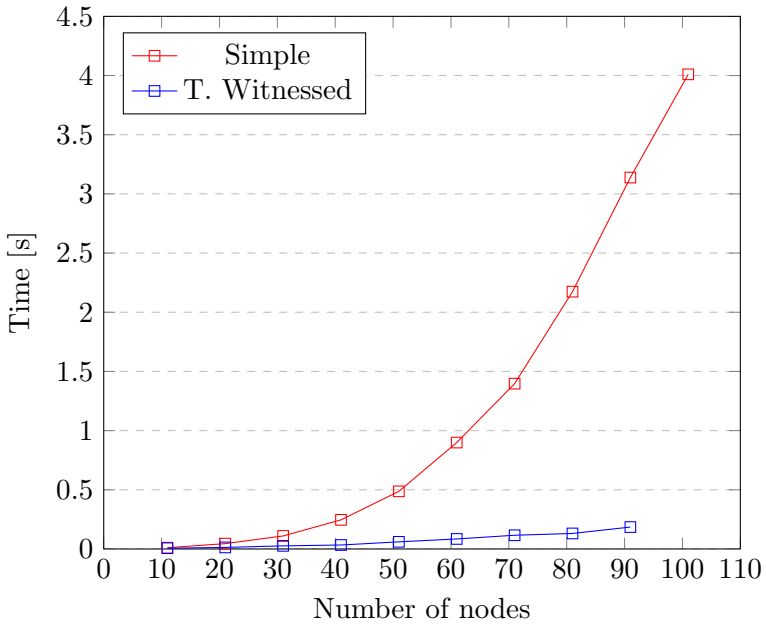

Figure 4: Average round time per number of nodes, $T_m = T_a = \lceil \frac{2n+1}{3} \rceil$

tive in a wide variety of smaller applications, or even in large scale systems that use committees [25].

In this section we give our system model and problem definition. We present a strawman argument, followed by an overview of our proposed solution using TLC. We show that TLC can significantly simplify consensus.

5.1 Model

In the classical consensus model, nodes operate in an asynchronous network. A node is called correct if it does not crash and is not malicious, i.e. does not deviate from the protocol, for the entire duration of the protocol. In a system of n nodes, the adversary may corrupt up to f of them.

In our previous brief explanation of consensus, nodes were allowed to propose any value at any time. However, we are interested in a more general definition of consensus, one where nodes can validate proposals according to a globally defined function or set of rules, and only valid proposals are accepted. For example, a proposal that says that Alice transfers Bob 1000 coins is not valid if Alice's balance is insufficient. Borrowing from Cachin et al [9][26], we define the following set of properties for validated consensus:

- **External Validity:** If an honest node decides a value v, then v is an externally valid value.
- **Termination:** Every honest node eventually decides.

- **Agreement:** If an honest node decides on a value v, then any other honest node that decides also decides v.
- **Integrity:** If an honest node decides v, then some node proposed v.

This typical definition of validated consensus is focused on deciding only one value at a time. Although this approach is sound, there is room for improvement: we can look at consensus as continuously proposing and deciding on *multiple* compatible values, instead of just one.

Take the analogy between a consensus proposal, and a block proposal in a typical blockchain with no forks: there are cases where nodes can "decide" different values without necessarily disagreeing. Imagine a case where Alice "decides" a valid block A, and Bob later "decides" a valid block B building on top of A. Even though Alice and Bob decided different blocks, they are not in disagreement. Bob's block implicitly includes Alice's block, as it builds on the same chain and is valid. He just hadn't seen Alice's block before. We can say that Bob's block is *compatible* with Alice's block, and vice-versa.

Trying to express it formally would amount to something like:

- **Consistent Agreement:** If an honest node decides on a value v, then every other honest node that decides decides a value compatible with v.

We refrain from presenting a full formal redefinition of validated consensus here. However, the notion of compatible values and how they fit our original goals will be useful later.

Another property worth mentioning is *fairness*, or the idea that nodes should get fair representation in the values picked over multiple consensus rounds. To achieve this, it is enough that values from honest nodes are decided in the consensus with good probability. Honest nodes can simply share values between them before proposing them to the consensus to make sure no node is left out. We will not focus on it in this report, but it is possible to show that our solution also holds this property.

5.2 System Architecture

We present a brief explanation of our solution to consensus. The purpose of this section is to give an intuition of how the solution works and the appeal of our modular approach.

5.2.1 Certification levels

If we look at TLC from a very basic perspective, we can summarize its behaviour to the following:

- we provide it a message m
- it returns a set of messages S, and T of those messages come with cryptographic proof that they were *witnessed* by A nodes this round.

Specifically, if a message is *witnessed* by a node in a round then it will be present in that node's S set for that round. The idea that a message is witnessed by a group of nodes is closely related to the idea of *quorums* [27].

We say that a message m is *certified* in a round t if there is cryptographic evidence that m was witnessed by A nodes in that round. We call this piece of cryptographic evidence the *certificate* of m, or $cert(m)$. Basically, TLC is guaranteeing that T of the messages it returns are *certified*, and so are delivered by A nodes.

If we were to now now give this certificate to TLC as the message for the next round, nodes can also obtain a certificate of this certificate, i.e. $cert(cert(m))$.

We generalize this notion to that of a *certificate level*. For example, we say that a message has certificate level 2 (is *double certified*) from the point of view of a node n, if node n has received $cert(cert(m))$, and so on.

We now present a strawman solution to asynchronous consensus with crash-stop failures (non-Byzantine) that tolerates up to a minority of node failures.

5.2.2 Strawman

Assume that nodes have access to a source of shared randomness. In a non-Byzantine setting, a shared seed value can work. They also have access to TLC, with the parameters T_m and T_a set to a strict majority in order to tolerate a minority of failures (optimal).

Nodes start by giving their proposed values TLC. They order the messages they receive according to a random permutation of the senders' ids, using their shared random value to get the same order. Each node then picks the first message they received, according to this order.

The problem with this approach is that nodes do not necessarily share the same sets of messages. For example, in a system of 3 nodes, each node might end up with a different pair of messages, one for each possible pair of nodes. If nodes were to pick the first *certified* message instead, the same problem would happen. What if we add *more* rounds, and only decide on the best certified proposal later? Doing this still doesn't fix our problem: nodes will have to decide at some point, and there is no guarantee that they all see the same message at the same *certification level*.

5.2.3 Randomized Asynchronous Consensus

Even if adding more rounds does not immediately solve the problem, it does head in the right direction. Our solution to consensus builds on top of the strawman: instead of nodes using only one round, nodes use two extra rounds to communicate. In the second round they share everything they saw in the first, including the message certificates, and collect *double certificates*. Likewise, in the final third round nodes share what they've seen in the previous round, but this time just to propagate information (the triple certificates are not necessary).

After the final propagation round is over and the proposals are ordered like in the strawman solution, nodes do two things:

1. take the lowest *single certified* proposal they have received. This is the *top* proposal.
2. check if the top proposal is also *double certified*, and there is no other proposal lower than it (certified or not).

If a node's second step is successful, it decides/commits the top proposal.

Because nodes might receive different sets of certified messages, it might be the case that some nodes end up deciding in a round while others don't, which violates *Termination*. This can be fixed by continuously repeating the above algorithm until all nodes have decided. However, doing this naively and simply reusing the same proposals could lead to issues in the long run, namely violations of *Agreement*. Alice might decide on a proposal A in the first run of the algorithm, while Bob might decide on a *different* proposal in the second run, simply because he was unaware that proposal A was double certified.

This is where the notion of compatible values becomes useful: in subsequent repeats of the above algorithm, nodes only propose values *compatible with their top proposal*. If we think of proposals as blocks in a blockchain, this would be equivalent to only proposing blocks that build *on the top proposal's chain*. We can prove that this is sufficient to maintain agreement. The general intuition is the following:

If honest nodes use TLC in the way described at the start (sharing messages of the previous round), and have the TLC parameters set to strict majority ($\lceil \frac{n+1}{2} \rceil$), we get the following lemma:

Lemma 5.1 *If any node sees a message m as K-certified by round R, all honest nodes will see m as (K-1)-certified by round R+1.*

Layer	Description
consensus	single globally-consistent historical timeline
randomness	unpredictable, unbiasable public randomness
time release	withholds information until designated time
threshold time	communication-driven global notion of time
witnessing	threshold certification that nodes saw messages
logging	logging of nondeterministic events nodes observe
real time	labeling events with approximate wall-clock time
authentication	message signing and source authentication
messaging	direct messaging between pairs of nodes

Figure 5: Layered architecture for threshold logical time and consensus. TLC incorporates the bottom 6 layers.

If a message is double certified for Alice after the second round, then it will be *at least* single certified for Bob after the third (propagation) round. In the previous example, if Alice decides a value after the first run of the algorithm, then Bob or any other node that didn't decide will *necessarily* propose a value compatible with Alice's: they would all have seen Alice's value as *single certified* at the end of the first run. In fact, any value that is decided later is compatible with every previous value, and so *Agreement* (or more specifically *Consistent Agreement*) is ensured.

Probability of Success and Termination

Because the order of messages done uniformly at random, every proposal has essentially the same chance of being the lowest global proposal. By the end of the third round, every node obtains a set of roughly $\frac{n}{2}$ double certified messages (strictly greater in fact), and so the probability that the lowest global proposal is double certified from any node's point of view is at least $\frac{1}{2}$.

Repeating the entire algorithm over multiple *consensus* rounds (sets of 3 TLC rounds), we can upper bound the probability p that a specific node n has not decided by the end of K consensus rounds:

$$P(\text{node n not decided}) < (\frac{1}{2})^K$$

This means that, with a sufficiently large but constant number of rounds, every node will eventually decide except with negligible probability, and so *Termination* is achieved.

Tolerating Byzantine failures

The solution presented here is only for asynchronous consensus with crash-stop failures. However, it is surprisingly easy to adapt it to tolerate

Byzantine failures. The overall idea is to:

1. Use an accountability system like PeerReview [17] to keep nodes honest when it comes to having witnessed messages.

2. Employ publicly verifiable shared randomness, and only *after* the TLC communication steps. This stops a strong adversary from knowing the outcome in advance and manipulating the network.

3. Change TLC's parameters to the typical Byzantine threshold of $\lceil \frac{2n+1}{3} \rceil$ (theoretical optimum [28]).

For the sake of simplification we omit here several other important primitives (e.g. secure authenticated links). Figure 5 represents the layered view we have in mind. This design by layers allows us to come up with simple solutions to otherwise complex problems, ideally at only a moderate cost to performance.

6 Conclusion

In this work we introduced TLC, a synchronization abstraction that makes it simpler to implement and reason about asynchronous protocols. We analyzed the performance of two of its possible implementations. Finally, we showcased its usefulness by building a simple asynchronous consensus algorithm on top of it. Based on our findings and results, we believe TLC is a useful and practical primitive that makes big steps towards unifying synchronous and asynchronous protocols.

7 Future Work

Future work working on formal theoretical proofs for the consensus algorithm, implementing both its Byzantine and non-Byzantine versions, and evaluating their performance. It would also be interesting to explore gossip based alternatives to consensus and TLC under different threat model definitions, as well as other potential applications of TLC.

References

[1] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing paxos," *ACM Sigact News*, vol. 34, no. 1, pp. 47–67, 2003.

[3] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft refloated: Do we have consensus?" *Operating Systems Review*, vol. 49, no. 1, pp. 12–21, 2015.

[4] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, p. 7, 2009.

[6] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.

[7] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, p. 12, 2015.

[8] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[9] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[10] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] C. J. Fidge, *Timestamps in message-passing systems that preserve the partial ordering*.

[13] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Citeseer.

[14] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[15] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2017, pp. 444–460.

[16] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines."

[17] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: Practical accountability for distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 175–188, 2007.

[18] P. Berman and J. A. Garay, "Randomized distributed agreement revisited," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 1993, pp. 412–419.

[19] B. Chor and C. Dwork, "Randomization in byzantine agreement," *Advances in Computing Research*, vol. 5, pp. 443–497, 1989.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process." MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, Tech. Rep., 1982.

[21] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[22] I. Abraham, D. Malkhi, and A. Spiegelman, "Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication." *CoRR, abs/1811.01332*, 2018.

[23] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 375–392.

[24] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience," Cryptology ePrint Archive, Report 2018/1028, 2018, https://eprint.iacr.org/2018/1028.

[25] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[26] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[27] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[28] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.