Received: 31 October 2018 | Revised: 18 February 2019 | Accepted: 6 March 2019
DOI: 10.1002/cpe.5276

WILEY

SPECIAL ISSUE PAPER

Non-blocking two-phase commit using blockchain

Paul Ezhilchelvan | Amjad Aldweesh | Aad van Moorsel

School of Computing, Newcastle University, Newcastle upon Tyne, UK

Correspondence
Paul Ezhilchelvan, Newcastle University, Newcastle upon Tyne, NE4 5TG, UK.
Email: paul.ezhilchelvan@ncl.ac.uk

Amjad Aldweeseh, Newcastle University, Newcastle upon Tyne, NE4 5TG, UK.
Email: a.y.a.aldweesh2@ncl.ac.uk

Present Address
Paul Ezhilchelvan, Amjad Aldweesh, Aad van Moorsel, School of Computing, Newcastle University, Newcastle upon Tyne, NE4 5TG, UK

Summary

The two-phase commit (2PC) protocol has long been known to have a provably inevitable vulnerability to blocking or non-progress amidst server crashes, even when the distributed database system guarantees the most demanding timing-related or "synchrony" requirements. Our aim here is to eliminate this vulnerability by using a blockchain for coordinating 2PC execution. We present the impossibilities, the possibilities, the cost, and the trade-offs in this blockchain-based approach to blocking-free management of distributed transactions. We prove that a non-blocking and blockchain-coordinated 2PC protocol can exist only if both the blockchain and distributed database systems meet synchrony requirements; otherwise, although blocking remains eliminated, transactions can unnecessarily abort. We present a blockchain-coordinated 2PC protocol and provide rigorous arguments for its correctness under the synchrony requirements. We then implement this protocol on the Ethereum Testnet and demonstrate, through our experiments, that the monetary cost of executing smart contracts is quite small, that the protocol performance slows down when using a public blockchain like Ethereum, and that even major violations of synchrony requirements lead only to relatively small increases in unnecessary aborts. We thus identify a trade-off between improving protocol performance and admitting a risk that transactions could occasionally abort unnecessarily.

KEYWORDS

atomic commit, blockchain, blocking protocols, delay bounds, smart contract, synchronous systems, two-phase commit

1 | INTRODUCTION

Since the advent of bitcoin as introduced by Satoshi,[1] cryptocurrencies have gained considerable interest. This is then followed by an even larger interest being accorded to bitcoin's underlying technology, the blockchain, and to Ethereum's development of smart contracts that empower users to execute custom-made programs on a blockchain. A variety of applications outside the cryptocurrency domain, such as finance,[2] banking, and energy trade,[3] have been leveraging blockchain and smart contract technologies to enhance accountability, auditability, and trust in their core processes.

This paper investigates the use of these technologies in enhancing the availability of distributed database management systems[4,5] and the associated cost. Precisely, we revisit a well-known impossibility result[6,7] related to *blocking* in atomically committing database transactions and demonstrate that these new technologies, under certain conditions, help accomplish what would otherwise be impossible.

When a database transaction is executed by multiple processes in a distributed system, an atomic commit protocol ensures the essential requirement that all processes either commit the transaction or abort it—a requirement that is commonly known as *Atomicity* or *Agreement*. The two-phase commit (2PC, for short) protocol is widely used as an atomic commit protocol due to its conceptual simplicity, ease of implementation, and low message cost. It is, however, vulnerable to periods of non-progress or *blocking*. This vulnerability is proven[6] to be inevitable even in *synchronous* distributed systems, where bounds on delays (eg, message transfer delays) can be reliably estimated and the only type of undesirable events that can occur is process crash.

The definition of a "synchronous" distributed system has long been established in the literature.[8] In our earlier work,[9] we extended this definition for a blockchain system and developed a protocol in which the blockchain plays specific roles in the execution of 2PC. This protocol was shown to eliminate blocking when both the distributed system and the blockchain used are synchronous. Its design, however, required that

the timestamps of blocks in a blockchain be increasing in value and that they emulate "ticks" of a global clock to database servers. While the Ethereum blockchain meets this requirement, other blockchain systems do not, and newly emerging ones may not. Hence, in this paper, we remove this requirement and present a new protocol together with correctness arguments. This new version also eliminates blocking under synchronous constraints and retains the native structure of 2PC for database processes, which makes it easily adoptable in legacy systems.

To the best of our knowledge, our earlier paper[9] is the first in the literature to demonstrate that the impossibility result in the work of Skeen[6] can be circumvented in synchronous distributed systems by using a synchronous blockchain. This revised and extended version not only improves on the earlier protocol but also addresses two significantly pertinent questions: can blocking be eliminated if the blockchain or the distributed system is *not* synchronous, and, if the answer is no, what are the practical implications if the blockchain and the distributed system can be synchronous most of the time, but not always?

Some blockchain systems, typically the public ones with miners having the freedom of choice in composing their blocks, may cease to be synchronous if it becomes harder to accurately estimate delay bounds. Similarly, a *cluster* hosting distributed database servers becomes asynchronous if accurate delay-bound estimation within the cluster is not guaranteed.

We are thus faced with four possible combinations: (1) the blockchain is synchronous and the database cluster is asynchronous, (2) the blockchain is asynchronous and the cluster is synchronous, (3) both are asynchronous, and (4) both are synchronous. 2PC blocking is eliminated for case (4) as our protocol would demonstrate. Still to be addressed, therefore, is the question of whether 2PC blocking can be eliminated for the other three cases.

We argued in our earlier work[9] that the elimination of 2PC blocking cannot be guaranteed for (3). We prove here that the same impossibility holds for more restricted cases of (1) and (2) as well. Thus, the impossibility results presented here are stronger than those shown in our earlier paper[9] and point to quite a fundamental result: a non-blocking 2PC using a blockchain is possible *if and only if* both the blockchain and the database cluster are synchronous. That is, many desirable features that a blockchain system has, such as reliability, immutability, etc, are not, by themselves, sufficient to eliminate 2PC blocking, and synchrony is required additionally.

Finally, when the blockchain and the distributed system are considered to be synchronous, even carefully computed delay-bound estimates are at risk of being violated, eg, due to bursts in network traffic. We argue that such violations can cause some *commit*-worthy database transactions to *abort* unnecessarily but cannot undermine the core *Atomicity* requirement that all servers either *commit* or *abort*. We investigate the relation between the number of *unwarranted* aborts and the degree of violations in the synchronous assumption and observe that the former is small even when the latter is large.

In summary, this paper explores and exposes the impossibilities, the possibilities, the cost, and the trade-offs involved in using a blockchain to implement non-blocking atomic commit. Its structure and contributions are as follows. The next section presents the atomic commit problem that 2PC solves, the notion of blocking, and the distinction between synchronous *versus* asynchronous distributed systems. Assuming a synchronous system, Section 3 describes the traditional version of 2PC and explains the causes of 2PC blocking. It thus provides the essential background for Section 4, which describes in detail our first contribution that is in the domain of *protocol design*: a non-blocking 2PC with a synchronous blockchain, together with pseudocode for smart contracts and correctness arguments. Section 5 presents our second *conceptual* contribution: the impossibility results that prove that non-blocking 2PC is not possible when either the blockchain or the distributed system is asynchronous and the observation that synchrony violations in a blockchain-coordinated 2PC have no impact on non-blocking atomic commit except for potential to cause unwarranted aborts. Our *practical* contributions are detailed in Section 6, which describes an Ethereum Testnet–based implementation of the protocol of Section 5 and discusses the results of our experiments. The discussions present the cost of smart contract execution, report both the estimated and observed worst-case 2PC execution latency values, quantify the probability of occurrence of unwarranted aborts caused by synchrony violations, and point out the scope for trade-off between improving performance and minimizing wasteful aborts. Finally, Section 7 concludes this paper.

2 | THE ATOMIC COMMIT PROBLEM

The problem is specified in the context of a set of distributed processes $\Pi = \{P_1, P_2, \ldots, P_n\}$, where $n > 1$ is known. A process P_i, $1 \leq i \leq n$, can crash at any time and recover after some arbitrary amount of time. Information *logged* in the disk prior to crash survives the crash. At any given instance, there are two complementary subsets of Π: the *crashed* and the *operative*. For discussions, we would assume that the former is small and a strict subset of Π.

Each operative process autonomously evaluates a *vote* that can be either *yes* or *no*. The problem is to have processes *decide* either on *commit* or *abort*, subject to the following four requirements.[10]

- *Agreement*: No two processes decide differently.
- *Termination*: All operative processes decide.
- *Abort-Validity*: Abort is the only possible decision if some process votes *no* or does not vote at all.
- *Commit-Validity*: Commit is the only possible decision if every process is operative and votes yes.

Agreement requires any two decided processes, currently crashed or operative, to have decided identically. For instance, P_k decides on *commit* and immediately crashes; then, no other process can decide on *abort* even if all but P_k are operative and deduce P_k to have crashed. *Termination*

ensures that the decision be available to all working processes; in particular, if a process crashes undecided, it should be able to decide when it becomes operative again, post-recovery.

Abort-Validity permits a process with *no* vote not to exercise its vote at all. *Commit-Validity* rules out trivial solutions such as all processes perforce decide on *abort* irrespective of their votes. This last requirement, as we shall see in Section 5, is impossible to guarantee even in blockchain-based solutions when the worst-case delay estimates being used are not guaranteed to hold.

Observe that any nontrivial solution to atomic commit requires operative processes of Π to interact among themselves—either directly leading to *decentralized* protocols or via a protocol *coordinator C* leading to *centralized* versions. The former extracts a huge message cost. The widely used 2PC protocol is a centralized one and is highly message efficient. It would be our focus here. (In practice, the role of *C* is typically played by a designated process in Π.)

Definition 1. An atomic commit protocol is said to be *blocking* if there *can* exist executions in which operative processes cannot decide until some nonempty subset of crashed processes ought to recover.[6,11] Blocking is thus undesirable as the progress of operative processes, normally larger in number, is dictated by the recovery times of crashed ones. A protocol is *non-blocking* if operative processes are *guaranteed* to decide even if each crashed process is never to recover. Whether one can have a non-blocking atomic commit protocol or not depends on if the distributed system is *synchronous* or *asynchronous*.[7,10]

2.1 | Synchronous vs asynchronous systems

Definition 2. A distributed system is said to be *synchronous* if bounds on processing delays and inter-process communication delays can be reliably estimated; otherwise, it is said to be *asynchronous*.[7,10]

Note that the bound estimates in a synchronous system can be large (typically, worst-case estimates) but must be finite and hold reliably. Typically, distributed systems where delays can fluctuate arbitrarily and, therefore, reliable bound estimations are not possible are classed as asynchronous.

It is known that non-blocking atomic commit is not possible when the distributed database system is asynchronous,[7] unless the system obliges *every* execution by behaving in certain desirable ways.[10] It is, however, possible to have a non-blocking atomic commit in a synchronous system by using the message-expensive, decentralized approach.[12,13] Intuitively, the design rationale in this approach is as follows. Reliable bound estimates in a synchronous system are used to implement *perfect* crash detection using timeouts: a crash is always detected and an operative process is never mis-detected (no false positive/negative). In addition, protocol performance is speeded up by assuming a bound on the maximum number of processes that can crash.[13]

Nevertheless, the centralized 2PC is a blocking protocol *even* in a synchronous system,[6] ie, even when a cluster hosting Π supports delay bounds to be estimated reliably and can thereby facilitate perfect crash detection.

2.2 | Synchronous vs asynchronous blockchains

We observe that this synchronous vs asynchronous classification holds for blockchain-based systems[9] as much as for traditional distributed systems. (Earlier definitions[9] will be restated in Section 4.2 for completeness.) In public blockchain systems, such as Ethereum, the time taken for a *valid* transaction to be *confirmed* or irreversibly placed in the blockchain is determined by a variety of delay-prone factors—both human as well as system related; for instance, a miner being (un)willing to include a transaction in their block[14] falls under the former category, and factors such as the required number of follow-up blocks to assure blockchain linearity and incoming transaction rate fall under the latter.

Ethereum blockchain confirmation time for a transaction can be unbounded with a significant probability,[14] suggesting large variances in end-to-end processing delays within the blockchain infrastructure. On the other hand, permissioned blockchain systems (eg, HyperLedger[15]) with their hardened modular implementation of consensus protocols[16] over dedicated machines appear to promise that the delays for transaction confirmation have small mean (in the order of milliseconds) and small variance and can, therefore, be reliably bounded, thus making such systems candidates for a synchronous blockchain.

3 | 2PC IN SYNCHRONOUS SYSTEMS

The 2PC protocol is explained below in the context of database transactions.[5] Shards of a database are distributed over processes in Π. We assume that a crash-prone process, called the *coordinator* and denoted as *C*, launches a multi-shard transaction that requires every process in Π to execute a set of serializable operations on their respective shards. We refer to this launching by *C* as each process in Π *getting_work* from *C*.

Let ω and δ denote upper-bound estimates on the time any operative $P_i \in \Pi$ takes to complete its work and on message transfer delays between any two operative processes, respectively. Since the system is assumed to be synchronous, ω and δ always hold.

C disseminates the work and awaits on a timeout of $(\omega + \delta)$ duration, which is sufficient for any operative P_i to receive and complete the work given to it. At the expiry of the timeout, it initiates an execution of 2PC by broadcasting *cast_vote* to all processes, as shown in line 1 of phase 1 for *Coordinator C* in Figure 1. This is then followed by setting a timer for $\Delta = 2\delta$ and proceeding to phase 2. (*Note: C* waiting for $(\omega + \delta)$ time before broadcasting *cast_vote* is not shown in Figure 1.)

PHASE 1

Coordinator C:

1. *Broadcast cast_vote* to all $P_1 \ldots P_n$
2. *Set Timeout* $\Delta = 2\delta$; *go to Phase 2*

P_i:

1. IF (*cast_vote* not received until T_i or $V_i = 0$) THEN *quit* ELSE {Log $V_i = 1$; *send* V_i to C; *Set timer*; *go to Phase 2*}

PHASE 2

C on timeout Δ:

1. IF any absent V_i THEN *verdict = abort* ELSE *verdict = commit*
2. Log *verdict*; *Broadcast verdict* to all $P_1 \ldots P_n$

P_i:

1. *Repeat* on *timer*: IF *verdict* arrived THEN Log *verdict* ELSE {request C; reset *timer*}
2. *Until verdict* logged

FIGURE 1 Two-phase commit protocol

When P_i receives work from C, it computes T_i as the local time when a duration $(\omega + 2\delta)$ would elapse after the receipt of the work. While doing the work, P_i will either complete it and set its vote $V_i = 1$ or decide that work cannot be completed in a serializable manner and set $V_i = 0$. In the latter case, by the *Abort-Validity* property, P_i can deduce that the decision or *verdict* is abort, ie, the transaction would be aborted system-wide; hence, P_i quits executing 2PC, as shown in line 1 of phase 1 for P_i in Figure 1.

Note that it is possible to have a 2PC implementation that makes P_i send $V_i = 0$ to C; we consider such an implementation only where relevant (Section 5.2); otherwise, we will assume the common (and message-optimal) case of P_i with $V_i = 0$ simply halting the execution with an *abort* decision.

If P_i has set $V_i = 1$, it waits to receive *cast_vote*. If the *cast_vote* message is not received until T_i, P_i assumes that C has crashed, decides *abort*, and quits its execution of 2PC. If, on the other hand, *cast_vote* arrives by T_i, P_i continues executing 2PC by logging its vote $V_i = 1$, sending V_i to C, and proceeding to phase 2. That is, the "ELSE" part in line 1 of phase 1 for P_i in Figure 1 is executed when (*cast_vote* not received until T_i or $V_i = 0$) is false, which is equivalent to (*cast_vote* received before T_i and $V_i = 1$) becoming true.

Note that while a given P_i may or may not enter phase 2, C always does. When its Δ-timeout expires, C counts an absent vote from any P_k as $V_k = 0$; it decides on *commit verdict*, if $V_i = 1, \forall i : 1 \leq i \leq n$; on *abort* verdict, otherwise. The *verdict* decided is logged and broadcast to all P_i (see phase 2 of Figure 1).

Any P_i that executes phase 2 awaits *verdict* from C and requests C periodically (as per some timer value), if *verdict* is not forthcoming. This periodic request will prompt a crashed C to respond after its recovery by referring to the *verdict* it logged prior to the crash. If no *verdict* has been logged, C must have crashed prior to computing the *verdict*; in that case, C's response would be *abort*.

Similarly, if P_i crashes after sending $V_i = 1$ to C, it will observe, after recovery, the log entry of $V_i = 1$ and request C to send the *verdict*. Thus, all operative processes, including those that crash during execution and recover, decide, ensuring *Termination*. It is easy to see that the other three requirements of atomic commit are also met in 2PC.

Figure 2 depicts the state transition diagram for any P_i, where a circle denotes a state and a double circle denotes a terminal state; a state transition is indicated by a unidirectional arrow with a label $\frac{I}{O}$, where I indicates the input received by P_i, which causes the transition, and O indicates any output produced by P_i after the transition. ("–" indicates null output.) WG, W_1, and W_2 represent states where P_i is doing the work given, waiting for *cast_vote* (see line 1, phase 1 in Figure 1) and for *verdict* (line 1, phase 2 in Figure 1), respectively; *a* and *c* denote the *terminal* states where P_i *aborts* and *commits*, respectively.

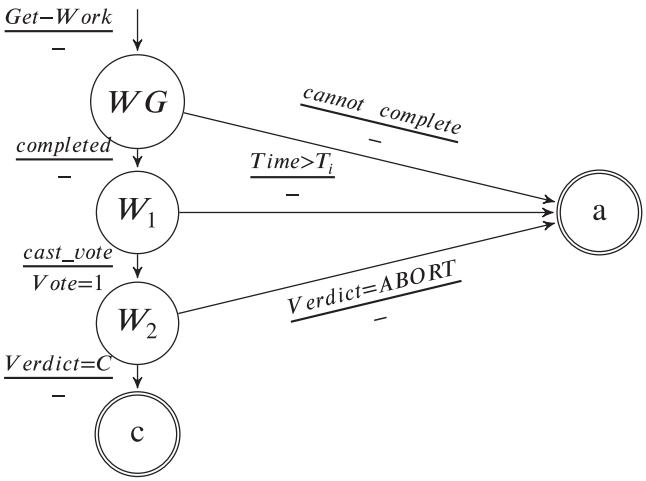

FIGURE 2 2PC state transition diagram for process P_i

3.1 | Inevitability of blocking in 2PC

While Skeen[6] formally proves this inevitability, we offer here, for completeness, an intuitive understanding of the reasons for it. By the definition of blocking (see Section 2), in every execution of a non-blocking 2PC protocol, operative processes decide despite some processes crashing and staying crashed, ie, operative processes reach a *verdict* that satisfies the atomic commit requirements without having to wait for any crashed process to recover.

We present three distinct execution *scenarios* of 2PC and show that no mechanism can possibly exist that avoids blocking in *all* scenarios and all meets all atomic commit requirements.

Scenario 1. In this execution of 2PC, every $P_i \in \Pi$ votes $V_i = 1$, and C crashes just before it is to broadcast its *verdict*. C remains crashed, ie, does not recover, for a long time.

Each P_i is blocked until C recovers. Suppose that blocking is avoided by using some *repair* sub-protocol $\mathcal{R}$ that enables operative processes to decide on a *verdict* (here, *commit*) without waiting for the crashed C to recover. For example, $\mathcal{R}$ may require operative processes to interact among themselves on how they voted and to arrive at a *verdict* that C would have broadcast had it not crashed. Next, two scenarios prove that $\mathcal{R}$ cannot exist.

Scenario 2. It is identical to *scenario* 1, except that one $P_k \in \Pi$ could not complete its work, decides on *abort*, and, then, crashes. P_k also remains crashed for a long time.

$\mathcal{R}$ must now enable all operative P_i, $i \neq k$, to decide on *abort* without waiting for P_k or C to recover.

Scenario 3. It is also identical to *scenario* 1, except that C crashes after sending *verdict* = *commit* only to P_k, which crashes soon after logging the received *verdict*. P_k, as in *scenario* 2, remains crashed for a long time.

$\mathcal{R}$ must now lead all operative P_i, $i \neq k$, to decide on *commit* without waiting for P_k or C to recover.

We observe that the execution environments of scenarios 2 and 3 are identical for all operative P_i, $i \neq k$: first, both C and P_k remain crashed until all P_i decide on *verdict*; second, there is no interaction between P_k and C in **Scenario 2** after C broadcast *cast-vote* and P_i cannot deduce any of the precrash interactions between P_k and C in **Scenario 3** until one of the crashed ones recovers. Thus, $\mathcal{R}$ is expected to make all operative P_i decide differently in identical execution environments. Such an $\mathcal{R}$ cannot be designed, and hence, 2PC blocking is inevitable.

Remark 1. As per Skeen,[6] the root causes for the inevitability of 2PC blocking are 2-fold: (1) both terminal states, *c* and *a*, are one-step reachable from W_2, as can be seen in Figure 2, and (2) it is possible to have an operative P_i waiting in W_2 and a crashed P_k either in *a* (see scenario 2) or in *c* (see scenario 3). In Skeen's terminology, (2) is referred to as the terminal states, *c* and *a*, being in the *concurrency* set of W_2. Designing $\mathcal{R}$ involves modifying 2PC itself and introducing new preterminal "*buffer*" states so that both terminal states are not in the concurrency set of W_2. This 2PC modification leads to three-phase commit, and details are given in the work of Skeen.[6]

4 | NON-BLOCKING WITH BLOCKCHAIN

4.1 | Approach

We can observe that if C *were* never to crash *during* 2PC execution, then blocking cannot happen. We build on this observation by having C initiate a transaction by delegating work to all P_i and then entrust the 2PC coordination responsibilities to a blockchain infrastructure (BC, for short), which, being a replicated state machine, must coordinate 2PC execution in a crash-free manner. To accomplish this, several aspects of BC will be made use of, and they are listed below.

Event ordering. Events directed at a BC are also called *transactions*. BC puts a total order on these events and records them in that order; event recording is immutable, and recorded events are permanently visible to all concerned parties. Event ordering in BC can also be used to ensure *exactly once* execution of an action, for instance, A when multiple sources, eg, processes in Π, can request A's execution: BC can be programmed (see smart contract below) to accept only the first request for A in the total order and ignore the duplicates.

Smart contract. A smart contract is a computer program stored within, and run by, BC in response to a *function call* embedded within an ordered transaction. Execution is guaranteed to be correct and is publicly verifiable. A smart contract has a unique address, and its code is stored within BC. The latter is structured as a collection of deterministic *functions* that can only be invoked by transactions admitted into BC. In Ethereum (see next item), contract code is written in languages like Serpent, LLL, or Solidity.[17] Irrespective of the language used, the code is compiled into bytecode and interpreted by a BC component, such as an Ethereum virtual machine.

Ethereum.[18] Ethereum is a popular platform that supports smart contract technology and is used in our implementation described in Section 6. A user process, such as C, can deploy a smart contract in BC by launching a transaction whose *data* field contains the bytecode of the smart contract with parameters appropriately initialized. Once this transaction is accepted in BC, any named process, such as P_i, can invoke a contract function by submitting a transaction. The invoking transaction is constructed with (1) the receiver address pointing to the contract address and (2) the parameter values for the function call. In addition, in Ethereum, a transaction includes two more fields: GAS and GAS PRICE.[18] The miner

who adds a block to BC will use the GAS PRICE to convert the amount of GAS consumed into the Ethereum's native currency called Ether. Thus, the sender of an invoking transaction is charged for executing the contract.

4.2 | Synchronous blockchain

Similar to definitions of ω and δ, let β be the *block construction* bound on the delay that can elapse between the instant when a user process U launches a valid (blockchain) transaction TX_U and the instant when a block containing TX_U is (irreversibly) added in BC; let α be the *awareness* bound on the delay that can elapse between the instant when TX_U enters BC irreversibly and the instant when any interested party gets aware of TX_U in BC. A BC infrastructure (together with miner/consensus nodes) is said to be[9] *synchronous* if it supports reliable estimation of bounds β and α; otherwise, it is said to be *asynchronous*.

The assumption of a synchronous BC implies that several requirements have been met: a valid transaction submitted to BC is never lost but is always considered for entry into the BC in a timely manner, a party interested in a given TX_U is periodically scanning BC, etc. This is just like the validity of δ bound requiring that no message be lost but every message be queued, transmitted, received, and delivered, all in a timely manner.

4.3 | 2PC with synchronous blockchain

We explain here (1) how C hands over the coordination responsibilities for 2PC execution to the BC infrastructure and (2) how P_i interacts with BC to execute 2PC in two phases. Informally, P_i uses phase 1 to register its *vote* in BC and phase 2 to receive the *verdict*, very similar to the traditional 2PC execution. We also assume that the cluster hosting database processes Π is synchronous as well. We do not, however, require processes of Π to directly detect each other's crash (eg, by operating a failure detector). This is also the case in the traditional 2PC version.

4.3.1 | Coordinator C

C disseminates the work to each $P_i \in \Pi$, and, immediately after that dissemination, it enters phase 1 to hand over the coordination to BC infrastructure. On entering phase 1, C launches a blockchain transaction TX_C that sets up the 2PC coordination smart contract in BC with initial *state* = *VOTING*.

Phase 1 for C ends with the launch of TX_C, and there is no phase 2. Another major difference from the traditional 2PC is that C does not wait on any timeout between disseminating its work to Π and entering phase 1. Note that C may crash during work dissemination or after dissemination and before launching TX_C.

Although Section 4.4 is devoted to explaining the smart contract in detail, the roles of two of its functions are briefly explained here for ease of understanding: function *VOTER* enables P_i to enter its vote in BC and computes the *verdict* once all $P_i \in \Pi$ have voted, and function *VERDICT* allows a P_i to explicitly request for the *verdict* to be computed. Moreover, once the smart contract computes the *verdict*, it changes the initial *state* to display the computed *verdict*, ie, to *COMMIT* or *ABORT*.

4.3.2 | Get-Work by P_i

When P_i receives work from C, it records its current local clock time as T_i and enters the "working" state WG (see Figure 3). If C has indeed launched TX_C, then TX_C must enter BC no later than the local time $T_i + \delta + \beta$ and P_i must observe TX_C in BC no later than its local time $T_i + \delta + \beta + \alpha$.

If P_i cannot complete the work due to serializability constraints, it unilaterally decides on *abort* and terminates the execution. This is shown by the state transition from WG to a in Figure 3.

If, on the other hand, P_i completes the work from C, it enters phase 1 by transiting from WG to the first *wait* state W_1 in Figure 3.

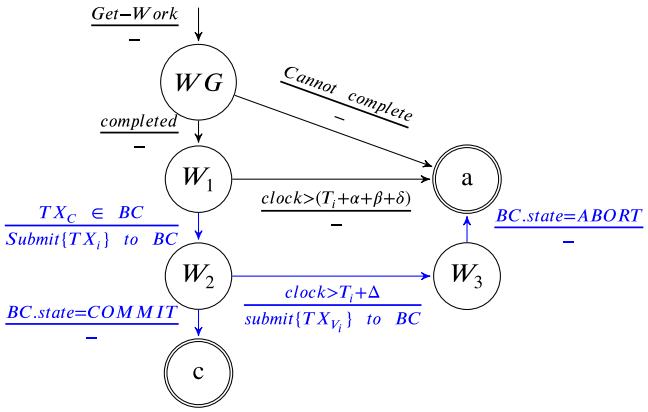

FIGURE 3 State diagram for 2PC with blockchain

4.3.3 | Phase 1 by P_i

P_i starts phase 1 by looking for TX_C in BC. If it does not observe TX_C in BC until its clock has exceeded $T_i + \alpha + \beta + \delta$, it deduces that C crashed before launching TX_C and subsequently *aborts*, as shown by the transition from W_1 to a in Figure 3. P_i awaiting TX_C to appear in BC is similar to its waiting for *cast_vote* in Figure 1. Moreover, the transitions from state *WG* in Figure 3 are identical to the traditional 2PC execution shown in Figure 2. (Transitions from *WG* are also called "off-chain" activities.[19])

If P_i gets aware of TX_C by local time $T_i + \alpha + \beta + \delta$, it logs T_i first, followed by logging of $V_i = 1$ (the latter as in phase 1 of Figure 1). The logging order of T_i and then V_i is important for post-recovery execution, by which P_i can decide if it crashed undecided after this point in 2PC execution (description is given in Section 4.3.5).

After logging T_i and V_i, P_i launches transaction TX_i with its vote $V_i = 1$. It then enters phase 2, with its state transiting from W_1 to a second *wait* state W_2 in Figure 3. Note that P_i launching its TX_i must happen by its clock time $T_i + \max\{\alpha + \beta + \delta, \omega\}$, where $\max\{\alpha + \beta + \delta, \omega\}$ is the larger of $(\alpha + \beta + \delta)$ and ω: P_i must observe TX_C in BC by clock time $T_i + \alpha + \beta + \delta$ and complete its work by $T_i + \omega$.

4.3.4 | Phase 2 by P_i

When TX_i is accepted in BC, it invokes the *VOTER* function of the smart contract with $V_i = 1$ as input. Moreover, if all $P_j \in \Pi$ launch TX_j, ie, vote $V_j = 1$, then the *VOTER* function would compute *verdict* = *commit* and display *state* = *COMMIT* when the last $V = 1$ is counted; otherwise, the *state* of BC will remain at the initial *state* = *VOTING* (details are given in Section 4.4).

Let $\Delta = \max\{\alpha + \beta + \delta, \omega\} + \alpha + \beta + \delta$. P_i in phase 2 waits for the BC state to change to *state* = *COMMIT* until its clock time $T_i + \Delta$. If P_i observes BC *state* = *COMMIT* by then, it decides *verdict* = *commit*.

If P_i, on the other hand, still observes *state* = *VOTING* until its clock exceeds $T_i + \Delta$, this means that some P_k, $k \neq i$, did not launch TX_k. Hence, *verdict* must be *abort*. Although P_i can now safely decide *abort*, our description here assumes that P_i decides on *verdict* = *abort* in response to such an indication from BC, just as in the traditional 2PC description where a P_i that voted $V_i = 1$ decides on *abort* by receiving *verdict* from C.

When BC *state* = *VOTING* and clock exceeds $T_i + \Delta$, P_i launches TX_{V_i} to invoke the *VERDICT* function of the smart contract so that *verdict* is computed in BC and displayed. In Figure 3, P_i does $W_2 \rightarrow W_3$ after launching TX_{V_i}, waits in W_3 until BC indicates *state* = *ABORT*, and, then, decides *verdict* = *abort*.

Waiting by P_i in W_3 must terminate as BC is reliable. It is likely that several other P_j launch their own TX_{V_j} around about the same time when P_i launches TX_{V_i}. If so, only one will be effective in executing *VERDICT* (like A in Section 4.1). Once BC indicates *state* = *ABORT*, P_i decides on *abort* and terminates the execution (W_3 to a in Figure 3).

4.3.5 | Post-recovery execution

It is possible that some $P_k \in \Pi$ crashes during the protocol execution. When it recovers, there are two possible cases: log of P_k has or does not have entry $V_k = 1$.

The absence of entry $V_k = 1$ means that TX_k was never launched and any work done by P_k has been erased from its (volatile) memory during the crash. Hence, the recovered P_k has no knowledge of the database transaction that triggered the 2PC execution. P_k could, and hence would, do nothing regarding that database transaction; in other words, P_k indirectly decides on *abort*. Furthermore, any $P_i, i \neq k$, that logged $V_i = 1$ can also decide only on *abort*.

Suppose that the log of P_k has the entry $V_k = 1$. This means that P_k, prior to its crash, must have observed TX_C in BC during its precrash execution of phase 1 and also logged the local time T_k (see Section 4.3.3). P_k will resume executing 2PC starting from phase 2 (with its state in W_2) and get the *verdict* from BC.

Since T_k is logged prior to logging $V_k = 1$, the log that contains $V_k = 1$ must have T_k as well. If P_k had crashed after logging T_k but before V_k (hence, before launching TX_k), then V_k would not be found in the postcrash execution and the entry T_k without a matching V_k is simply deleted.

Note that the post-recovery execution enables P_k to decide even if P_k is the only process in Π to have logged $V_k = 1$ and crashed before launching TX_k, while all others transited from *WG* to a: the recovered P_k would then launch TX_{V_k} when its clock $> T_k + \Delta$, and BC would subsequently change its *state* from *VOTING* to *ABORT*. Note also that there is no assumption on how long a crashed P_k can take to recover.

4.4 | Smart contract pseudocode

Figure 4 presents the pseudocode of 2PC coordination, and the description here assumes that the contract is already deployed on the blockchain with a unique address. The deployed contract is in the initial *state INIT* and has two set variables, namely, Σ and Σ_V, which are the set of participants eligible to vote and the set of those who actually voted, respectively; both sets are initially empty (when BC *state* = *INIT*). The smart contract has three functions, as follows:

- *REQUEST*() invoked by TX_C to initialize the contract,
- *VOTER*() invoked by TX_i to register the vote of P_i and to compute *verdict* once all $P_i \in \Pi$ voted, and
- *VERDICT*() invoked by TX_{V_i} to request the *verdict* to be computed, if not already done.

***INIT*:** Set $state := INIT$; $\Sigma := [0x000, \ldots, 0x000]$; $\Sigma_V := \Sigma$;

***REQUEST()*:** Upon C submitting $TX_C(\Pi)$: Assert ($state == INIT$ and $credentials\ of\ C$) Set $\Sigma := \Pi$; Set $state := VOTING$;

***VOTER()*:** Upon P_i submitting TX_i ($Vote$): Assert ($state == VOTING$ and $P_i \in \Sigma$); Assert ($P_i \notin \Sigma_V$); Assert ($Vote == 1$); Set $\Sigma_V := \Sigma_V \cup \{P_i\}$; if ($\Sigma_V == \Sigma$) then Set $state := COMMIT$;

***VERDICT()*:** Upon P_i submitting TX_{V_i}: Assert ($state == VOTING$ and $P_i \in \Sigma$); Set $state := ABORT$;

FIGURE 4 Pseudocode for 2PC coordination smart contract

TX_C submitted by C contains Π and invokes the *REQUEST* function. This invocation succeeds only if C is *asserted* to have ownership rights to invoke this function and the code is in the initial state *INIT*, as indicated in the *Assert* statement. If this assertion succeeds, TX_C is accepted, and the *state* of the contract is changed to *VOTING* and Σ to Π; otherwise, TX_C is ignored.

Note that it is the feature of any blockchain that a transaction, such as TX_C, is rejected if any of the pre-invocation assertions fails. Throughout this description here, assertions are assumed to succeed, except for those TX_V that seek to invoke the *VERDICT* function not for the first time.

Having observed TX_C in BC, a $P_i \in \Pi$ with vote $V_i = 1$ launches its TX_i. After asserting that *state* = *VOTING*, $V_i = 1$, and $P_i \in \Sigma = \Pi$, the contract records P_i to have voted by adding it in Σ_V. The BC *state* is changed to *COMMIT* when $\Sigma_V = \Sigma$.

Any P_i in W_2 that finds *state* = *VOTING* even after its clock has read $T_i + \Delta$ invokes the *VERDICT* function by submitting TX_{V_i}. The invocation succeeds only if $P_i \in \Sigma = \Pi$ and *state* = *VOTING*. If it succeeds, it sets *state* = *ABORT*. An attempt to redundantly invoke *VERDICT* when *state* = *ABORT* will not meet the latter condition and not succeed.

4.5 | Correctness arguments

They are based on the assumption that BC and the cluster hosting Π are both synchronous, ie, the bound estimates α, β, δ, and ω—as defined in Sections 3 and 4.2—are reliable and are never violated at any point during an execution.

Lemma 1. *If any P_i that received work from C at local clock time T_i does not observe TX_C in BC until local clock exceeds $T_i + \alpha + \beta + \delta$, then TX_C was never launched and would never enter BC.*

Proof. C is to launch TX_C immediately after it disseminates work to Π. By the definition of δ, work dissemination by C must complete within δ time, and the subsequent launching of TX_C must occur at or before P_i's clock time $T_i + \delta$ even if the work message had taken near-zero time to reach P_i, ie, even if C started its dissemination just before P_i's clock read T_i.

By the definitions of bound estimates β and α, P_i must observe TX_C in BC at or before its clock time $T_i + \alpha + \beta + \delta$. If P_i cannot observe TX_C until its clock exceeds that time, then either C did not launch TX_C or C launched TX_C *and* some bound estimate(s) got violated. When bound estimates are reliable and inviolable, the former ought to be the only underlying cause, and hence, the lemma ought to be correct. □

Lemma 2. *If any P_i that launched TX_i does not observe BC state = COMMIT until its local clock = $T_i + \Delta$, then there must be some P_j that did not launch TX_j, where $\Delta = \max\{\alpha + \beta + \delta, \omega\} + \alpha + \beta + \delta$ as defined in Section 4.3.4.*

Proof. Recall that P_i launches its TX_i no later than its clock time $T_i + \max\{\alpha + \beta + \delta, \omega\}$, as noted in Section 4.3.3. Suppose that another $P_j \in \Pi, j \neq i$, launches its TX_j at its clock time $T_j + \max\{\alpha + \beta + \delta, \omega\}$.

P_j receives its work from C at local time T_j, which can be as late as P_i's clock time $T_i + \delta$ because it is possible that C's work message to P_i took near-zero transmission delay whereas that to P_j suffered a maximum delay of δ. Thus, P_i could expect P_j to launch its TX_j no later than its clock time $T_i + \max\{\alpha + \beta + \delta, \omega\} + \delta$. This means that TX_j must enter BC and that its vote $V_j = 1$ must be counted no later than P_i's clock time $T_i + \max\{\alpha + \beta + \delta, \omega\} + \delta + \beta$.

Therefore, if every other $P_j \in \Pi$ had launched its TX_j, BC must have *state* = *COMMIT* no later than P_i's clock time $T_i + \max\{\alpha + \beta + \delta, \omega\} + \delta + \beta$, and P_i must get aware of this new BC *state* no later than its clock time $T_i + \max\{\alpha + \beta + \delta, \omega\} + \delta + \beta + \alpha$. If P_i observes BC in its initial *state* = *voting* when its clock exceeds $T_i + \Delta$, then some P_j did not launch TX_j. Hence, the lemma is proved. □

Corollary 1. *If P_i launches TX_{V_i} when its clock exceeds $T_i + \Delta$, there cannot be any TX_j from some $P_j \in \Pi$ that is yet to enter BC.*

Follows from Lemma 2.

4.5.1 | Agreement

Lemma 3. *In any execution, the Agreement requirement is met: no two processes decide differently.*

Proof. Consider $P_i, P_j \in \Pi$ and $j \neq i$. Suppose that they both decide. Without loss of generality, we will choose P_i to consider how it could decide on some *verdict* $\in \{commit, abort\}$ and argue that P_j cannot decide differently. P_i can decide in four ways, as follows.

1. P_i decides by observing BC *state* $\neq$ *VOTING*: Once the BC state changes to *COMMIT* from *VOTING*, no TX_V, if ever any launched, can reset *state* = *ABORT* because the assertion *state* = *VOTING* in the *VERDICT* function is not true. Similarly, once the BC state changes to *ABORT* from *VOTING*, no TX_i, if any launched, can reset *state* = *COMMIT* because the assertion *state* = *VOTING* in the *VOTER* function is not true. Thus, if P_j also decides on a *verdict* by observing BC *state* $\neq$ *VOTING*, it cannot decide differently to P_i.
2. P_i decides by transiting *WG* to *a*: P_i decides *verdict* = *abort* without ever launching TX_i. Assuming that *C* launches TX_C, the Boolean condition $(\Sigma_V == \Sigma)$ in the *VOTER* function will not become true, and BC *state* = *COMMIT* cannot happen. If P_j does not take the transition *WG* to *a* (as P_i) but goes on to launch TX_j, it cannot decide differently as *verdict* = *commit* when BC *state* $\neq$ *VOTING*.
3. P_i decides by observing $TX_C \notin$ BC when clock $> T_i + \alpha + \beta + \delta$: By Lemma 1, TX_C was never launched, and hence, P_j would also observe $TX_C \notin$ BC. Both identically decide on *abort*.
4. Crashed P_i executes post-recovery: For instance, its log has no entry for V_i. Hence, prior to crash, P_i may have decided on *verdict* = *abort* by modes (2) and (3) above; otherwise, it decides *indirectly* on *abort* during its post-recovery execution, as explained in Section 4.3.5. From P_j's perspective, P_i deciding indirectly is the same as P_i deciding by mode (2) in its crash-free execution if P_j observes TX_C in BC or by mode (3) otherwise. If the recovered P_i finds a log entry $V_i = 1$, then it decides by mode (1). Thus, P_j cannot decide differently to P_i.

Thus, given that P_i and P_j, $i \neq j$, decide, they cannot decide differently. □

4.5.2 | Termination and non-blocking

Termination requires that all operative processes decide. An operative process also refers to the one that recovers after a crash.

This requirement is met for any P_i that executes the protocol without crashing: it decides either (1) at the expiry of timeout $(\alpha + \beta + \delta)$ based on its local clock or (2) when BC changes from *state* = *VOTING*. Since BC is reliable, when P_i launches TX_{V_i} after its clock time $T_i + \Delta$, the BC *state* is guaranteed to change to *state* $\neq$ *VOTING* if it has not already.

Consider a P_k that crashes without deciding. After recovery, it either decides indirectly on *abort* or decide by (2) above. Thus, every operative process in Π decides.

Furthermore, neither (1) nor (2) above requires an operative process to wait until another crashed process in Π or crashed *C* to recover. Hence, the protocol is non-blocking.

4.5.3 | Commit-Validity

Lemma 4. *In any execution, the Commit-Validity requirement is met: commit is the only possible decision if every process is operative and votes yes.*

Proof. By given, every P_i logs $V_i = 1$ and votes by launching TX_i. Hence, *C* must have launched TX_C, and the smart contract must have been initialized. The only way the BC *state* can be changed from *VOTING* to *ABORT* is to have some TX_{V_i} enter BC and execute the *VOTER* function before some TX_j can enter BC. By the corollary above, this cannot happen. Thus, the BC *state* can change only to *COMMIT*. Any P_i that launches TX_i can decide only by observing BC *state* $\neq$ *VOTING*. Thus, commit is the only possible decision. □

4.5.4 | Abort-Validity

Abort-Validity requires that *abort* be the only possible decision if some process votes *no* or does not vote at all.

We have shown that all operative processes in Π decide in an execution, including those that crash and recover. In our protocol, a process $P_j \in \Pi$ either votes *yes* by launching TX_j with $V_j = 1$ or does not vote at all by never launching TX_j. When P_j does not launch TX_j, the *Boolean* condition $(\Sigma_V == \Sigma)$ in the *VOTER* function cannot become true, and BC *state* = *COMMIT* cannot happen. Furthermore, in our protocol, a decision can be either *commit* or *abort*, and an operative P_i can decide *commit* only by observing BC *state* = *COMMIT*. Hence, every operative P_i can decide only *abort* when some P_j does not vote at all.

Putting these arguments together, we can claim that our 2PC protocol with BC meets all four requirements of the atomic commit problem (Section 2) and is also non-blocking.

5 | ASYNCHRONY AND IMPOSSIBILITIES

When bounds α and β cannot be reliably estimated, BC becomes asynchronous (see Section 4.2); similarly, when estimates of bounds δ and ω are not guaranteed to hold, the cluster hosting Π becomes asynchronous (Section 2.1).

Note that a public BC can be asynchronous even if the underlying distributed system is synchronous. For example, if miners, at the time of TX_C launch, also encounter several other transactions that are more financially attractive to work on compared to TX_C, then TX_C could take longer

to enter BC, if at all, than any β estimated in more favorable environments.[14] Similarly, BC can be synchronous while the underlying distributed system is asynchronous. Thus, from the synchrony requirements perspective, our system is made up of two distinct subsystems: BC and database cluster. This leads to three pertinent questions: can we have a non-blocking 2PC in which coordinator C offloads its coordinating responsibilities to a BC, when:

1. the BC being used is synchronous and the cluster hosting Π is asynchronous?
2. the BC is asynchronous and the cluster is synchronous?
3. both the BC and the cluster are asynchronous?

Our earlier paper[9] answered question (3) in the negative but left (1) and (2) open. Furthermore, we also hinted in our earlier work[9] that it may be possible to have a non-blocking 2PC for (2) because processes of Π in (2) are endowed with an advantage of being able to accurately detect their crashed counterparts (ie, perfect failure detection).

We formally answer these open questions here and show that non-blocking 2PC is *not* possible in cases (1) and (2) as well. It turns out that the perfect failure detection capability within Π when the cluster is synchronous is not enough to construct a non-blocking 2PC if BC is asynchronous; our optimistic hint expressed in our earlier work[9] for case (2) is misplaced.

We next present the impossibility proofs. Our approach is to prove, by contradiction, which involves three steps: we will (1) hypothesize the opposite of the impossibility, ie, suppose the existence of some correct non-blocking 2PC protocol, for example, $\mathcal{P}$ that meets all four requirements of atomic commit in *every* possible execution scenario; (2) construct two execution scenarios that are indistinguishable from the perspective of any operative $P_i \in \Pi$; and (3) show that if $\mathcal{P}$ is correct in one scenario, it cannot be so in the other. This contradiction will demonstrate that no such $\mathcal{P}$ can exist and, thus, prove the impossibility.

The two execution scenarios we construct will have the following features in common.

- C never crashes, and the bound estimates used in $\mathcal{P}$ hold for all messages/requests sent by C.
- Every process $P_i \in \Pi - \{P_k\}$ is operative and wishes to commit by submitting a "*yes*" vote, $V_i = 1$.
- T_i denotes the local clock time when an operative P_i receives "work" from C.

Note that asynchrony in BC or cluster does not mean that the bound estimates used in $\mathcal{P}$ are *always* violated; they can be met on many an occasion. Hence, the first feature is a possibility that is assumed to hold in the chosen execution scenarios. It ensures that both executions have C offloading its coordination responsibilities in a timely manner and every operative P_i observing TX_C in BC also in a timely manner. The second feature leads to P_i launching TX_i with $V_i = 1$, when it observes TX_C in BC.

5.1 | Synchronous BC, asynchronous cluster

Let us first observe that the cluster is asynchronous, ie, with δ and ω being likely to be violated. Crash detection is typically done by periodically querying another process with "are you alive" pings and awaiting responses to be received within a set timeout duration. It cannot, therefore, be guaranteed to be perfect: an operative process may be seen, at least temporarily, to have crashed, and it may take several nonresponsive pings, and hence a long time, to affirmatively conclude that a process has indeed crashed.

Impossibility 1. It is not possible to have a non-blocking 2PC protocol where coordinator C offloads its coordinating responsibilities to a BC when that BC is synchronous and the cluster hosting $\Pi = \{P_1, P_2, \ldots, P_n\}$ is asynchronous.

Proof. Let us hypothesize that Impossibility 1 is wrong and that there exists a non-blocking 2PC protocol $\mathcal{P}$. Consider two executions of $\mathcal{P}$ that have the common features mentioned earlier. □

Execution 1. P_k does $WG \rightarrow a$ and then crashes. All other P_i's are operative and wish to commit and launch TX_i. Since $\mathcal{P}$ is presumed to solve atomic commit, each P_i, $i \neq k$, must decide eventually, in this case on *abort*; for instance, P_i decides at its local time $T_i + \mathcal{D}_i$ in this execution, for some (finite) $\mathcal{D}_i$. Furthermore, P_k does not recover in this execution until after every operative process has decided, ie, until the local time of every P_i reads or exceeds $T_i + \mathcal{D}_i$.

Execution 2. Every process of Π and C starts the execution at the same time as in *Execution* 1. Moreover, every P_i, $i \neq k$, sends and receives the same set of messages (including ping and ping-response messages) from each other until a decision as in *Execution* 1 and each such message are sent or received at the same local clock time as well. That is, the behavior of every undecided P_i toward every other undecided P_j, $j \neq k$, is identical in both executions.

P_k does not crash but observes TX_C by its clock time $T_k + \alpha + \beta + \delta$ (due to the first common feature) and completes its work. However, the bound estimate ω is violated so much, and the launching of its TX_k (with $V_k = 1$) is so delayed that TX_k does not enter BC until after the clock of every P_i, $i \neq k$, reads or exceeds $T_i + \mathcal{D}_i$. Moreover, every message sent by P_k (including P_k's response to ping) is delayed arbitrarily such that it does not reach its destination until after the clock of every P_i reads or exceeds $T_i + \mathcal{D}_i$. This is possible because the cluster, of which P_k is a part, is asynchronous.

Execution 2 is indistinguishable from *Execution* 1 for any $P_i, i \neq k$, until its clock time $T_i + \mathcal{D}_i$. In the former, P_k appears nonresponsive to any P_i until $T_i + \mathcal{D}_i$, whereas it remains crashed until $T_i + \mathcal{D}_i$ in the latter. Hence, as in *Execution* 1, P_i must decide on *abort* at $T_i + \mathcal{D}_i$. This violates the *Commit-Validity* requirement: if no process crashes and all vote "yes," the decision ought to be *commit* (see Section 2). Thus, the hypothesis is contradicted, and the impossibility is proved.

Remark 2. The proof makes no assumption on whether transmission delays of messages exchanged between any $P_i, P_j \in \Pi - \{P_k\}$, adhered to or violated the bound estimate δ. It is only assumed that the delay experienced by a given message is identical in both executions, which is a possibility that cannot be ruled out. Since $\mathcal{P}$ is supposed to work for an asynchronous cluster, there must be a finite $\mathcal{D}_i$ for every $P_i, i \neq k$, in *Execution* 1. Messages from P_k taking longer than $\mathcal{D}_i$ to reach P_i in *Execution* 2 is another possibility in an asynchronous cluster, which is also assumed. Thus, *Execution* 2 is a feasible execution scenario for $\mathcal{P}$.

5.2 | Asynchronous BC, synchronous cluster

Since the cluster is synchronous, bound estimates δ and ω remain inviolable. Therefore, a pinging process can affirm that a pinged process is operative or crashed if the latter does or does not respond within 2δ time, respectively; neither false positives nor false negatives are possible. The availability of this perfect failure detection capability is taken into consideration in constructing the impossibility proof below.

Impossibility 2. It is not possible to have a non-blocking 2PC protocol where coordinator *C* offloads its coordinating responsibilities to a BC when that BC is asynchronous and the cluster hosting $\Pi = \{P_1, P_2, \ldots, P_n\}$ is synchronous.

Proof. Let us hypothesize that Impossibility 2 is wrong and that there exists a non-blocking 2PC protocol $\mathcal{P}$ for asynchronous BC and synchronous cluster. Consider two executions of $\mathcal{P}$ that have all the common features mentioned earlier. □

Execution 1. P_k remains operative, does $WG \rightarrow a$, decides on *abort*, and quits the execution without ever submitting TX_k to BC. All other P_i's also remain operative but wish to commit and launch TX_i. Since $\mathcal{P}$ is presumed to solve atomic commit, each $P_i, i \neq k$, must also decide on *abort* eventually; for instance, every P_i decides on *abort* at its local time $T_i + \mathcal{D}_i$ in this execution, for some (finite) $\mathcal{D}_i$.

Execution 2. Every process of Π and *C* starts the execution at the same time as in *Execution* 1. Moreover, every $P_i, i \neq k$, sends and receives the same set of messages (including ping and ping-response messages) from each other until a decision as in *Execution* 1 and each such message are sent or received at the same local clock time as well. That is, the behavior of every undecided P_i toward every other undecided $P_j, j \neq k$, is identical in both executions.

P_k does not crash, and, like every other operative $P_i, i \neq k$, launches its TX_k. However, while every TX_i enters BC taking the same delay as in *Execution* 1, TX_k enters BC after a delay that is the maximum in $\{\mathcal{D}_i : \forall P_i \in \Pi - \{P_k\}\}$. Consequently, TX_k does not enter BC until after the local clock of every P_i reads $T_i + \mathcal{D}_i$. Note that TX_k taking longer than β to enter BC is possible since BC is asynchronous.

Execution 2 is indistinguishable for any $P_i, i \neq k$, from *Execution* 1 until its clock time $T_i + \mathcal{D}_i$. In the former, P_k never submits TX_k, whereas, in the latter, TX_k does not appear in BC until after every P_i decides. Moreover, in both executions, perfect failure detectors of P_i will report P_k as an operative process. Hence, as in *Execution* 1, P_i must decide on *abort* at $T_i + \mathcal{D}_i$. This violates the *Commit-Validity* requirement: no process crashed, and all voted *yes* (see Section 2). Thus, the hypothesis about $\mathcal{P}$ is contradicted, and the impossibility is proved.

Remark 3. We noted in Section 3 that some 2PC implementations force a process with a "*no*" vote to explicitly cast its vote. In such an implementation, P_k would launch a transaction TX_k with "*no*" in *Execution* 1. In that case, this TX_k should be considered to behave exactly like the TX_k in *Execution* 2: taking a delay that is the maximum in $\{\mathcal{D}_i : \forall P_i \in \Pi - \{P_k\}\}$ and not entering BC until after the local clock of every P_i reads $T_i + \mathcal{D}_i$. *Executions* 1 and 2 are now indistinguishable for any $P_i, i \neq k$, until its clock time $T_i + \mathcal{D}_i$. Thus, Impossibility 2 holds even in such uncommon implementations.

5.3 | Implications of synchrony violations

A closer look at the impossibility proofs reveals that asynchrony in BC or in the cluster prevents only *Commit-Validity* from being guaranteed, ie, *abort* could be decided when all processes of Π are operative and vote *yes*. This is also confirmed by the correctness arguments in Section 4.5, which show that our 2PC protocol operating with BC solves the atomic commit problem when both BC and cluster are synchronous. More precisely, these arguments indicate that if (1) *C* crashes without launching TX_C, (2) some P_k crashes, or (3) some P_i votes *no*, the other three requirements are guaranteed to be met even when the delay-bound estimates are violated: arguments for *Termination* (Section 4.5.2) and *Abort-Validity* (Section 4.5.4) do not refer to synchrony assumptions at all; moreover, in cases (1)-(3) above, *verdict* = *abort* is the correct outcome, and *verdict* = *commit* cannot ever be reached. Hence, the *Agreement* requirement is also met. In summary, synchrony is needed only to guarantee *Commit-Validity*.

Thus, when a bound estimate $b \in \{\alpha, \beta, \delta, \omega\}$ is violated, the only requirement that risks being compromised is *Commit-Validity*, leading to unwarranted *aborts* of database transactions. Violations of b can occur due to transient surges in computational loads or network traffic or the traffic and/or loads having increased since the bound estimates were last computed.

At any given time, let b_a be the *actual* prevailing value for an estimate $b \in \{\alpha, \beta, \delta, \omega\}$. Synchrony is violated if $b < b_a$ for *any* b. This does not necessarily mean that the two timeouts used in the protocol would be violated. (Recall that $(\alpha + \beta + \delta)$ is the phase-1 timeout defined in Section 4.3.3 for deciding whether TX_C would ever appear in BC, and $\Delta = \max\{(\alpha + \beta + \delta), \omega\} + (\alpha + \beta + \delta)$ is the phase-2 timeout defined in Section 4.3.4 before launching TX_V.)

For example, if only $\alpha < \alpha_a$ and $b > b_a$ for every other b, we can still have $\alpha + \beta + \delta \geq \alpha_a + \beta_a + \delta_a$ and $\Delta \geq \max\{(\alpha_a + \beta_a + \delta_a), \omega_a\} + (\alpha_a + \beta_a + \delta_a)$. Denoting $\Delta_a = \max\{(\alpha_a + \beta_a + \delta_a), \omega_a\} + (\alpha_a + \beta_a + \delta_a)$, let us define

$$m_1 = \frac{\alpha + \beta + \delta}{\alpha_a + \beta_a + \delta_a} \text{ and } m_2 = \frac{\Delta}{\Delta_a}. \quad (1)$$

Only when $m_1 < 1$ or $m_2 < 1$ are phase-1 and phase-2 timeouts at risk of becoming "too small," respectively, leading to the possibility of a transaction being unnecessarily aborted and *Commit-Validity* not being upheld. As noted, $(m_1 \geq 1 \wedge m_2 \geq 1)$ can still hold when only *some* bound estimates suffer minor violations. Using our protocol implementation described next, we evaluate the likelihood of unwarranted *abort* occurrences when phase-1 and phase-2 timeouts are made small by varying amounts.

6 | IMPLEMENTATION AND EVALUATION

We implemented the 2PC blockchain contract from Figure 4 in Solidity 0.40.11[17] and tested its operation on the Ethereum Testnet network,[20] using Ethereum Wallet and Ethereum Mist.[21] Four different machines are used: (a) a MacBook Pro with a 2.8-GHz Intel i5 CPU and 8-GB RAM and (b) three desktop PCs with a 3.20-GHz Intel i7 CPU and 8-GB RAM running on Windows 10. The MacBook is the coordinator C, and the three desktop PCs constitute the "cluster" hosting P_1, P_2, and P_3. Each PC is connected to the Ethereum Testnet as a full node, thus having a full copy of the blockchain stored within it. The PCs do not play the role of miners themselves and operate as non-mining database hosts connected to the blockchain. They are also connected to each other and to switches by a standard switched Ethernet local area network, which connects through a standard Transmission Control Protocol/Internet Protocol with the Ethereum Testnet. The smart contract (see Figure 4) is also registered with the Ethereum Testnet.

6.1 | Delay-bound estimation

In all our experiments, the database transaction is kept null because our main objective is to assess the cost and performance of coordination activities within and around the blockchain. Consequently, a "get-work" message from C contains no work for P_i but simply initiates the latter to execute 2PC, which votes *yes* or *no* as per the purpose of a given experiment; thus, the bound estimate $\omega = 0$. Other bounds α, β, and δ are established as follows.

The awareness delay (bounded by α) is calculated by taking the difference between the confirmation time of a given transaction of interest (such as TX_C or TX_i) entering a block in BC and the time of receiving this block by each P_i. The confirmation time is obtained from the Ethereum Wallet, which shows the time that the block was added. The timestamps at the three P_i nodes give us three data points, and the maximum of these three results is taken as one data point for estimating α. At the end of 30 experiments, in which only C launched TX_C, the maximum of the 30 data points obtained is taken as α.

The block entry delay (bounded by β) is calculated as the difference between the timestamp given to TX_C at the coordinator node when TX_C is sent and the confirmation time of the block that contains TX_C within the blockchain. Similar to α, we take the maximum of all data points obtained as β.

To obtain α and β, each individual experiment consists of C submitting one single transaction TX_C and ends once we have collected all the data points. Each experiment takes several minutes, as we will see, and is repeated 30 times.

To measure data points for transmission delays (bounded by δ), no P_i needs to interact with the blockchain. We measure these data points by letting C send a 1 KB Ethernet packet to each processor P_i, which then sends it back to C. We take the round-trip time and halve it to get one-way delays. The maximum of all data points collected is taken as δ: we collected 30 round-trip times for each P_i; hence, δ is the maximum over 90 one-way delay estimates.

The results for α, β, and δ are shown in Figures 5, 6, and 7. In all three Figures, the x-axis gives the experiment number (from 1 to 30), and the y-axis gives the point estimate of α, β, and δ (the max of the results in the three nodes, as explained above).

In estimating α, all experiments return values within the 2-minute range. The highest observed value is for experiment 4, at 115.734 seconds. Figure 5 shows only the maximum of the values for the three P_i values, and we note that the difference between the three obtained values in each of the 30 experiments is minimal, less than 1 second. For information, the average and the median of the block awareness delays depicted in Figure 5 are 30.461 and 13.455 seconds, respectively.

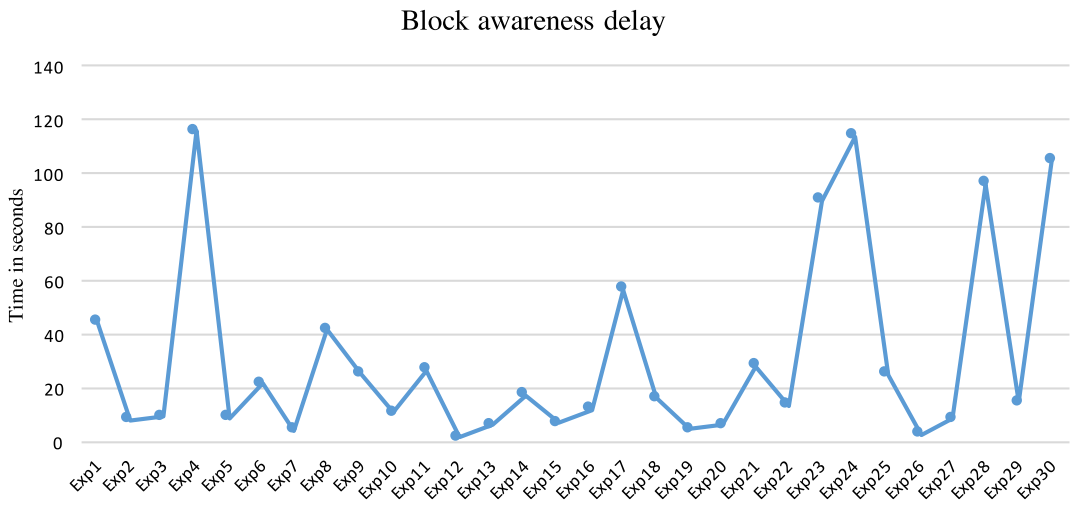

FIGURE 5 Block awareness delay

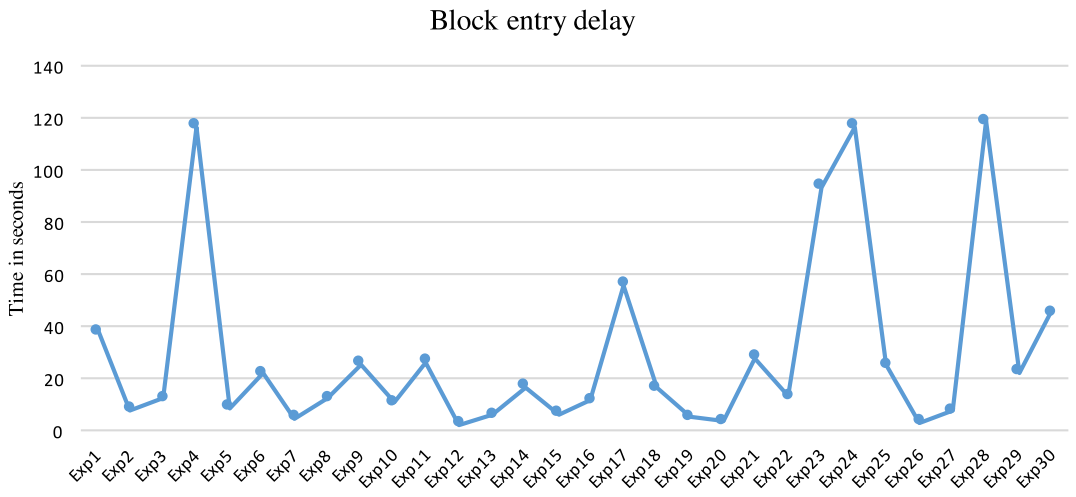

FIGURE 6 Block entry delay

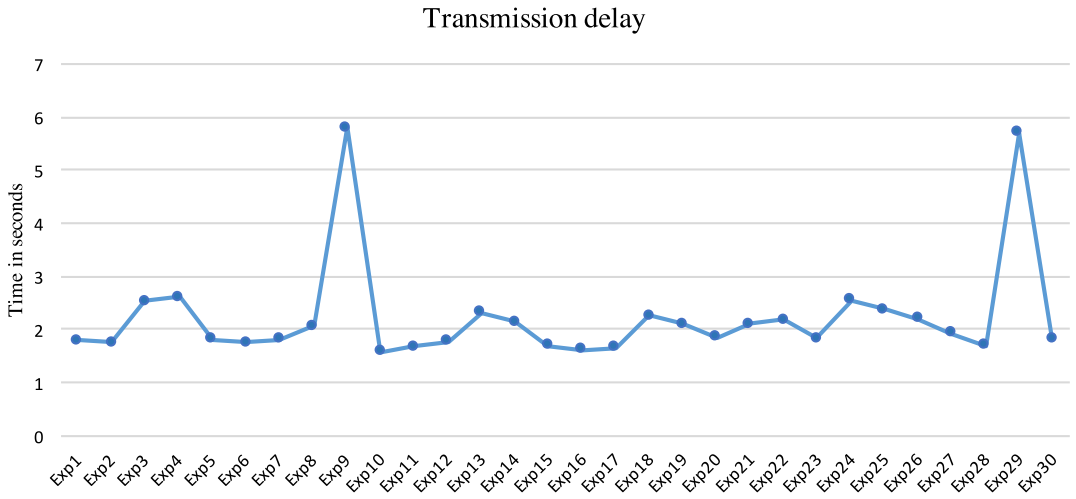

FIGURE 7 Transmission delay

In the experiments for β, the maximum is found in experiment 28, at a value of 118.800 seconds. Note that for some experiments, the transaction finds its way into a block in a matter of seconds, the minimum observed delay was 2.355 seconds. The block entry delay is influenced by factors such as the transaction's gas price, which, in turn, influences miners' decisions of which transactions to include into the blocks they work on.

Figure 7 shows the results of our experiments for estimating δ. They range from 1.590 to 5.790 seconds.

TABLE 1 Cost of executing 2PC blockchain contracts

Transaction	Reason	GAS Used	Cost in ETH
TX_c	By C to request voting	232 736	0.000232736
TX_i	By P_i to vote	84 625	0.000084625
TX_v	By P_i to seek verdict	55 102	0.000055102

TABLE 2 Total cost in various voting scenarios

Scenarios	GAS Used	Cost in ETH
Three vote no	232 736	0.000232736
Two vote no	372 463	0.000372463
One votes no	457 088	0.000457088
All vote yes	486 611	0.000486611

6.2 | Cost of 2PC coordination

As noted in Section 4.1, the initiator of a blockchain transaction that involves executing one or more functions of a smart contract ought to pay the miner in the cryptocurrency *ether* that is commonly abbreviated as ETH. The payment is in proportion to the amount of "gas" (often written as GAS) consumed by the executions of functions a transaction invokes.

Furthermore, a transaction initiator can quote in the transaction the gas price they are willing to pay for executing the smart contract functions. A higher gas price quoted can act as an incentive to miners in giving preferential treatment over those that quote a lower gas price. In our experiments, the gas price quoted was the lowest possible, for example, the coordinator quotes the gas price of 0.001 ETH/million for executing the *REQUEST* function. By quoting only the lowest gas price, the cost in ETH we report here would indicate the lower bound.

When a smart contract function involves repetitive executions conditional on Boolean statements (eg, a *while* loop), the gas cost can vary with the inputs supplied at invocations. As we can see from Figure 4, the 2PC coordination code does not involve aspects that lead to input-dependent execution cost variations, except when the last $P_i \in \Pi$ casts its vote, the Boolean $\Sigma_V == \Sigma$ (which is checked on every invocation of *VOTER()*) comes true, and "Set *state* = *COMMIT*" is additionally executed. This additional execution of a simple "Set" statement does not incur any extra gas, and it is confirmed in all our experiments.

The amount of gas that a miner uses when executing a given contract function is calculated by the Ethereum virtual machine itself and is displayed in the Ethereum Wallet at the initiator end. Thus, it is safer to assume that the reports on the amount of gas expended for executing a given contract function are quite reliable. Table 1 provides the cost of executing each of three smart contract functions: *REQUEST()*, *VOTER()*, and *VERDICT()*. As per the prevailing exchange rates for ETH, the cost is on the order of a few US cents or British pence.

Table 2 presents the total cost for 2PC coordination in four possible voting scenarios when the number of P_i in Π is 3.

When a P_i votes *no*, it knows that the *verdict* = *abort* and terminates. Thus, when all three P_i values vote *no*, none will launch TX_i or TX_{V_i}. Hence, only the *REQUEST()* function is executed and its gas price the total cost, as shown in row 1 of Table 2.

In considering the remaining rows of Table 2, let us assume that neither a process crash nor any violation of the bound estimates occurs during 2PC execution. If n' processes, $n' = 1$ or 2, vote *no*, $(3 - n')$ processes launch TX_i and, at the expiry of Δ timeout, also TX_{V_i}, of which only one will end up invoking the *VERDICT()* function. Thus, the total cost incurred is as follows: the cost of row 1 + $(3 - n^{'})$ × the cost of executing *VOTER()* function once + the gas cost of executing *VERDICT()* function once.

When all three processes vote *yes*, none will launch TX_{V_i}, and the total cost is as follows: the cost of row 1 + 3 × the cost of executing *VOTER()* function once. Generalizing, when y processes, $0 \leq y \leq |\Pi|$, vote *yes*, the total gas cost for 2PC coordination is as follows: gas cost of executing *REQUEST()* function once + y × the gas cost of executing *VOTER()* function once + c × the gas cost of executing *VERDICT()* function once, where $c = 0$ if $y = 0 \vee y = |\Pi|$, and $c = 1$ otherwise (ie, $0 < y < |\Pi|$).

6.3 | 2PC execution latencies

2PC execution latency for an operative P_i can be defined as the duration that can elapse from the moment P_i receives "work" from coordinator C until the moment P_i decides either to *commit* or *abort* the transaction. Let the moments of P_i receiving work and deciding be denoted as T_i and $T_i + E_i$, respectively, and be observed as per P_i's local clock. Thus, E_i is the 2PC execution latency for P_i. We will discuss E_i by first estimating the maximum value it can (theoretically) take and then reporting the actual maximum it took in our experiments, along with an explanation for any wide discrepancy between the two. Our estimation of latency bound will assume that the delay-bound estimates used were *conservatively* arrived at by assigning them to the largest data points observed (as described in Section 6.1) and, hence, are *safe*, ie, never violated.

TABLE 3 Minimum (Min), maximum (Max), and average (Avg) latency in minutes (Mn) and seconds (Ss) expressed as Mn:Ss

	D1	D2	D3	E1	E2	E3	E1-D1	E2-D2	E3-D3
Min	00:09.421	00:42.872	00:20.412	00:08.332	00:42.335	00:20.203	00:07.964	00:41.904	00:20.328
Max	02:56.276	04:37.990	02:40.783	02:55.178	04:36.880	02:40.806	02:55.112	04:36.842	02:40.742
Avg	00:30.336	01:19.295	00:48.959	00:36.843	01:26.309	00:49.466	00:36.728	01:26.217	00:49.489

6.3.1 | Estimated latency bound

All possible execution scenarios need to be considered before arriving at the upper bound for E_i. To start with, let us consider the simplest case where P_i takes the transition $WG \rightarrow a$ (see Figure 3); here, E_i cannot exceed ω.

Alternatively, P_i can vote *yes* instead of doing $WG \rightarrow a$. In this execution scenario, two cases need to be considered: TX_C does not or does enter BC. When TX_C does not enter BC due to C crashing subsequent to disseminating the "work," P_i will affirm the absence of TX_C at the expiry of phase-1 timeout and decide *abort*; thus, E_i = phase-1 timeout = $\alpha + \beta + \delta$. In the second case where C does not crash and TX_C does enter BC, E_i will depend on the number, y, of processes in Π that vote *yes*.

Let $y = |\Pi|$. Measuring time as per P_i's clock, we note that P_i would commence two parallel activities at T_i: doing the work given to it and looking for TX_C to appear in BC. The former must complete by $T_i + \omega$ and TX_C in BC would be known to P_i by T_i + phase-1 timeout = $T_i + \delta + \beta + \alpha$, at the latest. Thus, at or before $T_i + \max\{\omega, (\alpha + \beta + \delta)\}$, P_i must launch its TX_i, and all other P_j's must do so by P_i's clock time $T_i + \max\{\omega, (\alpha + \beta + \delta)\} + \delta$. Thus, the *verdict* computed at BC would be known to P_i no later than its clock time $T_i + \max\{\omega, (\alpha + \beta + \delta)\} + (\alpha + \beta + \delta)$. Therefore, $E_i \leq \max\{\omega, (\alpha + \beta + \delta)\} + (\alpha + \beta + \delta)$. Typically, ω is very small compared to $(\alpha + \beta + \delta)$, and thus, $E_i \leq 2(\alpha + \beta + \delta)$ when $y = |\Pi|$.

Let $y < |\Pi|$. (Since P_i votes *yes*, $y > 0$.) P_i would launch TX_{V_i} at its clock time $T_i + \Delta$ and would observe BC *state* = *ABORT* no later than its clock time $T_i + \Delta + \beta + \alpha$. Thus, $E_i \leq \Delta + \alpha + \beta$. Given that $\Delta = \max\{(\alpha + \beta + \delta), \omega\} + (\alpha + \beta + \delta)$ (defined in Section 4.3.4), $E_i \leq 2(\alpha + \beta + \delta) + (\alpha + \beta)$ when ω is considered small compared to $(\alpha + \beta + \delta)$.

Summarizing, E_i cannot exceed $\Delta + (\alpha + \beta) = 2(\alpha + \beta + \delta) + (\alpha + \beta)$ for an operative P_i in any possible combination of crashing and voting scenarios. Substituting the delay-bound estimates, the (upper) bound for E_i is $2(115.734 + 118.800 + 5.790) + (115.734 + 118.800) = 715.182$ seconds, ie, 11 minutes and 55.182 seconds.

Finally, let us also estimate, for the sake of comparison, the bound for E_i when 2PC is executed without BC (as described in Section 3). If P_i suffers blocking due to the crash of C, E_i can be arbitrarily long as P_i cannot decide until C recovers. When C does not crash, it turns out that $E_i \leq \omega + 4\delta$: having received "work" from C at its clock time T_i, P_i can receive the broadcast *cast_vote* at or before $T_i + \omega + \delta$; C broadcasts the *verdict* after a 2δ timeout expires following its broadcasting of *cast_vote*; P_i must decide by $T_i + \omega + \delta + 2\delta + \delta$ if it voted *yes*. Thus, using *BC* to eliminate 2PC blocking results in a performance slowdown when C does not crash, and the slowdown is bounded by $3(\alpha + \beta) - (\omega + 2\delta) \approx 3(\alpha + \beta) = 703.611$ seconds. Such a large slowdown should be expected, given the features of public blockchains as discussed in Section 2.2 and in the work of Weber et al[14] and the need to use safe delay-bound estimates so that both BC and the cluster remain synchronous, ie, synchrony violations do not occur.

6.3.2 | Observed latencies

We carried out 200 2PC executions using our implementation involving the Ethereum blockchain. We disallowed crashes and ensured that the "work" given by C is trivial to execute and all P_i's, $1 \leq i \leq 3$, always vote *yes*, ie, $y = |\Pi|$. Note that each execution must result in all three processes deciding *commit*; otherwise, it would mean that phase-1 or phase-2 timeout became "too small" in the prevailing execution environment and expired prematurely. In all 200 experiments, *commit* was indeed the decision.

In each experiment, P_i recorded the local clock times when it received the work, observed TX_C in BC, and decided as T_i, $T_i + D_i$, and $T_i + E_i$, respectively. D_i and $(E_i - D_i)$ represent the latency for P_i to execute only phase 1 and phase 2, respectively.

Table 3 summarizes the minimum, maximum, and average of the 200 latency values experienced by individual processes. We observe that the largest E_i is experienced by P_2 and stands at 4 minutes and 36.880 seconds. The corresponding upper-bound estimate (when $y = |\Pi|$) is $2(\alpha + \beta + \delta) = 2 \times 240.324 = 480.648$ seconds or 8 minutes and 0.648 seconds, which is about twice the maximum observed. In addition to this large discrepancy between the estimated and observed bounds for E_i (when $y = |\Pi|$), we also observe large differences between the maximum and the average (or minimum) latency in each column. The explanation for this lies in the shape of graphs in Figures 5, 6, and 7: the largest data point ends up deciding the estimate $b \in \{\alpha, \beta, \delta\}$ and is substantially larger than most frequently occurring data points. For example, as noted earlier, the largest awareness delay observed in Figure 5 is 115.734 seconds, which determines α; $0.2\alpha = 23.147$ is still larger than the average awareness delay observed (13.455 seconds) and $0.4\alpha = 46.294 > 30.461$, the median. Similarly, in the experiments for β in Figure 6, the peak value of 118.800 seconds was observed in experiment 28 and was adopted as β. Only in two other experiments the block entry delay came close to β, and in the rest, it was below 50% of β, with the minimum observed delay being 2.355 seconds.

6.4 | Impact of synchrony violations on Commit-Validity

We observed in Section 6.3.1 that E_i is the largest when C does not crash and $y < |\Pi|$: $E_i = \Delta + \alpha + \beta$. This is because all P_i's that vote *yes* are forced to wait until $T_i + \Delta$ before they could launch TX_{V_i}, which then causes BC to compute and display the *verdict*. Any attempt to

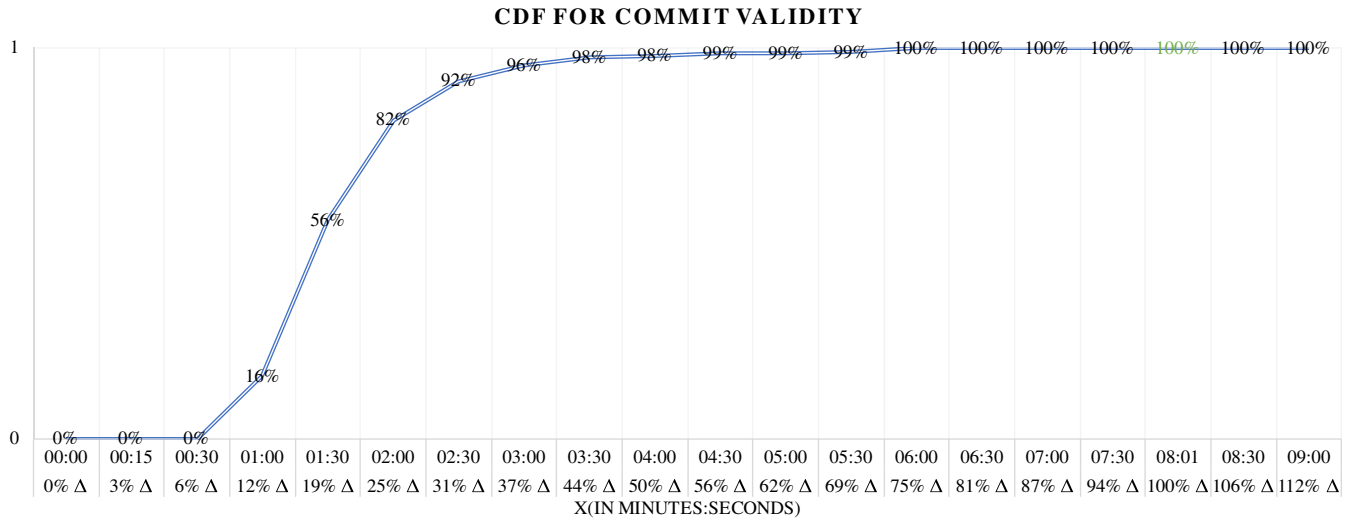

FIGURE 8 Probability for *Commit-Validity*

reduce E_i in this worst case and in other cases and, thus, to speed up 2PC execution in general requires using smaller values for Δ, α, and β; this calls for a less conservative estimation of α, β, and δ as Δ is a function of these delay-bound estimates. Deliberately underestimating delay bounds, however, tends to increase the scope for synchrony violations. We also noted in Section 5.3 that synchrony violations risk only the *Commit-Validity* requirement not being met, leading to unwarranted *aborts*. We will here evaluate the probability of *Commit-Validity* being met as synchrony violations are permitted to occur due to delay bounds being deliberately underestimated.

Recall that when ω is considered small compared to $(\alpha + \beta + \delta)$, the phase-2 timeout $\Delta = \max\{(\alpha + \beta + \delta), \omega\} + (\alpha + \beta + \delta)$ (defined in Section 4.3.4) simply becomes $2(\alpha + \beta + \delta)$, and the phase-1 timeout (see Section 4.3.3), $(\alpha + \beta + \delta)$, becomes $\Delta/2$.

Suppose that each bound estimate $b \in \{\alpha, \beta, \delta\}$ is chosen not as the largest data point observed (as in conservative estimations) but as m times the largest data point, where m is a small positive real number. When $0 < m < 1$, phase-1 and phase-2 timeouts drop to $m\Delta/2$ and $m\Delta$, respectively, and execution latency is reduced; in our experiments, *Commit-Validity* is upheld in an execution only if $D_i < m\Delta/2$ and $E_i < m\Delta$ for all $P_i \in \Pi$. For any given $X = m\Delta$, the probability of *Commit-Validity* being upheld is the fraction of 200 experiments in which $D_i < X/2$ and $E_i < X$ for all $P_i \in \Pi$.

Figure 8 depicts the cumulative distributive function for *Commit-Validity* for $X = m\Delta$, with m ranging from 0.03 to 1.12. (Absolute values of X are in the first row of the x-axis as Minutes:Seconds.) We observe that when X is as small as 0.25Δ, *Commit-Validity* is upheld with a probability as high as 82%. What this means here is that choosing $b \in \{\alpha, \beta, \delta\}$ to be 25% of the largest data point observed leads only to 18% of runs suffering unwarranted *aborts*, whereas it can reduce 2PC execution latency by 75%. Furthermore, the *Commit-Validity* probability rises quickly to 98% for m as small as 0.44, and it becomes 100% for $m \geq 0.75$. The latter indicates that 2PC execution latency can be reduced by 25% without suffering any unwarranted *aborts*. All these observations suggest that (1) small underestimations of delay bounds may not lead to unwarranted *aborts* at all and that (2) there is much room for reducing execution latency considerably at the expense of a modest increase in unwarranted *aborts*.

7 | CONCLUDING REMARKS

Common choices to avoid 2PC blocking are to use a decentralized protocol[12,13] or the (centralized) three-phase commit. These alternatives extract a larger message cost even in the absence of crashes and do not have the structural simplicity of 2PC. We have shown here that the message cost and implementation difficulties of existing 2PC alternatives can be avoided if the 2PC coordinator C simply offloads coordination responsibilities to a blockchain after disseminating database work to servers. Our proposed protocol maintains the low message overhead and the elegant structure of 2PC: those servers that want to commit look up to the crash-free blockchain for progress (instead of crash-prone C) and launch, at most, two blockchain transactions (instead of periodically pinging the crashed C until it recovers). The extra cost arises in two forms, namely, miners' fees and latency sacrifice when a public blockchain is used; the former are very small in fiat currencies, but the latter can be substantial, on the order of hundreds of seconds, as shown by our experiments involving the Ethereum blockchain. We believe that the performance slowdown will not be so serious, if permissioned blockchains had been used, and our future work would focus on such an investigation.

Although the blockchain infrastructure maintains the abstraction of a reliable state machine with an immutable audit trail display, such features are not sufficient to *guarantee* non-blocking atomic commit, unless it meets synchrony requirements. This is another important contribution of this paper that should be borne in mind when building applications similar to atomic commit using blockchain. For example, eVoting, like atomic commit, can be *guaranteed* to be correct only if the blockchain is synchronous; this aspect is not emphasized but is simply assumed in some blockchain-based eVoting systems.[22] Informally, the total number of "*yes*" votes cast is counted in both applications, and the count is displayed

in eVoting, whereas it is used to decide between *commit* and *abort* in atomic commit. Since a dishonest participant can seek to undermine the result of eVoting, it is important for an eVoting system to specify timing requirements to distinguish between a "timely" vote that gets counted and one that arrives "too late" and gets ignored. This naturally leads to synchrony requirements for correctness.

We have applied the traditional "best effort, worst case" method to reliably estimate delay bounds. We then emulated synchrony violations by deliberately choosing to use smaller values as bound estimates and, thereby, examined the extent of *Commit-Validity* violations resulting in unwarranted aborts. We observe that the number of unwarranted aborts occurred to be small even when bound underestimations are considerable. For example, a uniform reduction of 81% across all bound estimates still upholds *Commit-Validity* (ie, zero aborts) in more than 50% of runs ($X = 0.19\Delta$ in Figure 8). This is because the peak delays observed during bound estimation are much larger than the average or median delays. Hence, the "worst case" bound estimation offers built-in tolerance for synchrony violations. Its downside, however, is that the protocol takes much longer to terminate. Thus, there is a trade-off between reducing protocol latency and using smaller than "worst case" bound estimates, which risks violating *Commit-Validity*.

ORCID

Paul Ezhilchelvan https://orcid.org/0000-0002-6190-5685

REFERENCES

1. Satoshi N. Bitcoin: a peer-to-peer electronic cash system. 2008.
2. Tapscott A, Tapscott D. How blockchain is changing finance. Harvard business review. Boston, MA: Harvard Business Publishing; 2017.
3. Aitzhan NZ, Svetinovic D. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Trans Dependable Secure Comput*. 2016. https://doi.org/10.1109/TDSC.2016.2616861
4. Gray J. Notes on data base operating systems. In: Bayer R, Graham RM, Seegmüller G, eds. *Operating Systems: An Advanced Course*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1978:393-481.
5. Lampson LW. Atomic transactions. In: Davies DW, Holler E, Jensen ED, et al, eds. *Distributed Systems - Architecture and Implementation: An Advanced Course*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1981:246-265.
6. Skeen D. Nonblocking commit protocols. In: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD); 1981; Ann Arbor, MI.
7. Hadzilacos V. On the relationship between the atomic commitment and consensus problems. In: Simons B, Spector A, eds. *Fault-Tolerant Distributed Computing*. New York, NY: Springer-Verlag Berlin Heidelberg; 1990:201-08. *Lecture Notes in Computer Science*; vol 448.
8. Arjomandi E, Fischer MJ, Lynch NA. Efficiency of synchronous versus asynchronous distributed systems. *J ACM*. 1983;30(3):449-456.
9. Ezhilchelvan P, Aldweesh A, van Moorsel A. Non-blocking two phase commit using blockchain. In: Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems (CryBlock); 2018; Munich, Germany. http://doi.acm.org/10.1145/3211933.3211940
10. Guerraoui R. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*. 2002;15:17-25.
11. Raynal M, Singhal M. Mastering agreement problems in distributed systems. *IEEE Software*. 2001;18(4):40-47.
12. Raynal M. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In: Proceedings of the 2nd High-Assurance Systems Engineering Workshop (HASE); 1997; Washington, DC.
13. Dutta P, Guerraoui R, Pochon B. Fast non-blocking atomic commit: an inherent trade-off. *Inf Process Lett*. 2004;91(4):195-200.
14. Weber I, Gramoli V, Ponomarev A, et al. On availability for blockchain-based systems. Paper presented at: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS); 2017; Hong Kong, China.
15. Androulaki E, Barger A, Bortnikov V, et al. HyperLedger fabric: a distributed operating system for permissioned blockchains. Ithaca, NY: Cornell University Archive; 2018. http://arxiv.org/pdf/1801.10228v1.pdf
16. Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans Comput Syst*. 2002;20(4):398-461.
17. Solidity. Solidity Documentation. 2017. https://solidity.readthedocs.io/en/v0.5.6/
18. Wood G. Ethereum: a secure decentralised generalised transaction ledger. EIP-150 revision. 2016. http://gavwood.com/Paper.pdf
19. Xu X, Pautasso C, Zhu L, et al. The blockchain as a software connector. Paper presented at: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA); 2016; Venice, Italy. http://doi.org/10.1109/WICSA.2016.21
20. Ethereum. Ropsten testnet explorer. 2017. https://ropsten.etherscan.io/
21. Mist. Ethereum Mist. 2018. https://github.com/ethereum/mist/releases
22. McCorry P, Shahandashti SF, Hao F. A smart contract for boardroom voting with maximum voter privacy. Paper presented at: International Conference on Financial Cryptography and Data Security; 2017; Sliema, Malta.

How to cite this article: Ezhilchelvan P, Aldweesh A, van Moorsel A. Non-blocking two-phase commit using blockchain. *Concurrency Computat Pract Exper*. 2020;32:e5276. https://doi.org/10.1002/cpe.5276