ABSTRACT

Though the 2 Phase Commit protocol (2PC) remains central to distributed database management, it has a provably-inevitable vulnerability to blocking even when a distributed system guarantees the most demanding synchrony or timing-related requirements. This paper investigates eliminating that vulnerability by coordinating 2PC using a blockchain that supports execution of user-defined smart contracts. It demonstrates that the 2PC blocking can be eliminated at a moderate financial cost, if the blockchain also meets the synchrony requirements. Otherwise, despite the blockchain being a reliable state-machine, eliminating 2PC blocking may well be impossible, depending on whether the cluster hosting the database is synchronous or not. Where not possible, the practical consequences turn out to be not so serious: unnecessary aborts occurring with a small probability.

Keywords

1. INTRODUCTION

Since the advent of Bitcoin in 2009 [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 of cryptocurrency domain, such as Finance [13], Banking and Energy Trade [15], have been leveraging blockchain and smart contract technologies to enhance accountability, auditability and trust in their core processes.

This paper investigates using these technologies to enhance the availability of distributed database management systems [6]. Precisely, we revisit a well-known impossibility result [7, 8] related to atomic commit and demonstrate that these new technologies, under certain conditions, can help accomplish what would otherwise be impossible and that the conditions identified are met by the emerging blockchain systems. If a given blockchain system cannot meet these conditions, we assert that the impossibility cannot be ruled out altogether, even though many desirable features of a blockchain might tempt one to make more optimistic conclusions. Thus, this paper has the dual aim of presenting the possibilities to aspire for and the pitfalls to be aware of.

When a database transaction is executed by multiple processes in a distributed system, a commit protocol ensures the essential requirement of that transaction execution being either committed or aborted, at all processes. The 2 phase commit protocol (2PC, for short) is a widely-used one due to its conceptual simplicity and ease of implementation. It is however vulnerable to periods of non-progress or blocking. This vulnerability is proven [7] to be inevitable even in synchronous systems where bounds on delays (e.g., message transfer delays) can be reliably estimated, and the only type of undesirable events that can occur is process crash.

We define what it means for a blockchain system to be synchronous and employ such a blockchain to play specific roles in the execution of 2PC. The resulting protocol, referred to as 2PC with blockchain, is shown to be non-blocking. It also retains the native structure of 2PC which makes the proposed extension easily adoptable in legacy systems. Our contributions also include an Ethereum based implementation to assess the cost of smart contract execution which turns out to be affordably small.

We next observe that some blockchain systems, typically the public ones with miners having the freedom of choice in composing their blocks, may not qualify to be synchronous. When the blockchain used is asynchronous, we prove that eliminating 2PC blocking is not possible, if the distributed system hosting database processes is also asynchronous; it remains an open problem, if the latter system is synchronous.

The structure and the contributions of the paper are as follows. Next section defines the atomic commit problem that 2PC solves, blocking and synchronous versus asynchronous systems. Assuming a synchronous system, Section 3 describes in details the traditional version of 2PC protocol and then explains the causes of 2PC blocking. It thus provides the essential background for Section 4 which contains two of our three contributions: (i) detailed presentation of a non-blocking 2PC with a synchronous blockchain, and (ii) both cost and correctness analyses. Final contribution is in Section 5 which proves the impossibility of non-blocking 2PC when both the blockchain and the distributed system are asynchronous and then discusses this result from a practical perspective. Finally, Section 6 concludes the paper.
2. THE ATOMIC COMMIT PROBLEM

The problem is specified in the context of a set of distributed processes $\Pi = \{P_1, P_2, \ldots, P_n\}$, $n > 1$. 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 $\Pi$, the crashed and the operative. For discussions, we would assume that the former is small and a strict subset of $\Pi$.

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 [9]:

- **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; and,
- **Commit-Validity**: Commit is the only possible decision if every process is operative and votes yes.

**Agreement** requires any two decided processes, be it crashed or operative, to have decided identically. Say, $P_k$ 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. By **Termination**, any solution is guaranteed to be meaningful in practice.

**Abort-Validity** permits a process with no vote, not to exercise its vote. **Commit-validity** rules out trivial solutions such as all processes perforce decide on abort irrespective of their votes. These two requirements, as we shall see in § 5, together make it impossible to guarantee atomic commit even in blockchain based solutions when the worst-case delay estimates being used are not guaranteed to hold.

Observe that any non-trivial solution to atomic commit requires operative processes of $\Pi$ to interact amongst themselves - leading to decentralised protocols, or to a protocol coordinator $C$ - leading to centralised versions. The widely-used 2-Phase Commit (2PC) protocol is a centralised one and would be our focus here. (In practice, the role of $C$ is typically played by a designated process in $\Pi$.)

**Definition.** An atomic commit protocol is said to be blocking, if there can exist executions in which processes cannot decide until a subset of crashed processes ought to recover. 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 it is guaranteed never to block. Whether one can have a non-blocking atomic commit protocol depends on if the distributed system is synchronous or asynchronous [8, 9].

2.2 Synchronous vs Asynchronous Blockchains

We observe that this synchronous vs asynchronous classification holds for blockchain based systems as much as for traditional distributed systems. (Definitions are in § 4.2.) In public ledger 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 [12] 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.

As per [12], Ethereum blockchain confirmation time for a transaction can be unbounded with a significant probability, suggesting large variances in end-to-end processing delays within the blockchain infrastructure. On the other hand, permissioned ledger systems (e.g., HyperLedger [14]), with their hardened modular implementation of consensus protocols (e.g., [10]) over dedicated machines, appear to promise that the delays for transaction confirmation have small mean (in the order of milliseconds) and also small variance and can, therefore, be reliably bounded, thus making such systems candidates for synchronous ledgers.

A significant contribution of this paper is to show that 2PC can be made non-blocking, if the ledger system being used and the cluster hosting processes of $\Pi$ are synchronous.

3. 2PC IN SYNCHRONOUS SYSTEMS

The 2-Phase Commit protocol, 2PC for short, is explained below in the context of database transactions [6]. Shards of a database are distributed over processes in $\Pi$. We assume that a crash-prone process, called the coordinator and denoted as $C$, launches a multi-shard transaction that requires every process in $\Pi$ to execute a set of serialisable operations on their respective shards. We refer to this launching by $C$ as each process in $\Pi$ getting work from $C$.

Let $\omega$ and $\delta$ 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, $\omega$ and $\delta$ 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, phase 1 of Figure 1. This is then followed by setting a timer for $\Delta = 2\delta$ and proceeding to phase 2.

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
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 \(\Delta\):**
  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**

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 serialisable 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 i.e., the transaction would be aborted systemwide; so, it quits executing 2PC as shown in line 1 of Phase 1 for \(P_i\) in Figure 1.

If \(P_i\) has set \(V_i = 1\), it waits to receive cast\_vote. If 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 in 2PC by logging its vote \(V_i = 1\), sending \(V_i\) to \(C\) and proceeding to Phase 2.

Note that while a given \(P_i\) may or may not enter phase 2, \(C\) always does. When its \(\Delta\)-timeout expires, \(C\) counts an absent vote from any \(P_i\) 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 the 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 a terminal state; a state transition is indicated by an unidirectional arrow with a label \(\delta\) where \(I\) indicates the input received by \(P_i\) which causes the transition and \(O\) any output produced by \(P_i\) after the transition. (\(\cdot\) indicates null output.) **WG**, **W1** and **W2** represent states where \(P_i\) is doing the work given, waiting for cast\_vote (see line 1, phase 1 in Fig 1) and for **verdict** (line 1, phase 2 in Fig 1, respectively; \(a\) and \(c\) denote states where \(P_i\) aborts and commits, respectively.

3.1 Inevitability of Blocking in 2PC

While Skeen [7] formally proves this inevitability, we offer an intuitive understanding of the reasons for it by presenting three distinct execution **scenarios** of 2PC.

**Scenario 1:** In this execution of 2PC, \(C\) crashes immediately after broadcasting cast\_vote and all \(P_i\) vote \(V = 1\).

Each \(P_i\) is blocked until \(C\) recovers. Suppose that blocking is avoided by using a recovery protocol that enables operative processes to interact amongst themselves and decide on a **verdict** without waiting for the crashed to recover. Next two scenarios prove that a correct recovery cannot exist.

**Scenario 2:** As in scenario 1 except that one \(P_k\) could not complete its work, decides on abort and then crashes.

If a correct recovery exists, all operative \(P_i, i \neq k\), must decide on abort without waiting for \(P_k\) and \(C\) to recover.

**Scenario 3:** In this execution, \(C\) broadcasts cast\_vote, all \(P_i\) vote \(V = 1\) and \(C\) crashes sending **verdict** = commit only to \(P_k\) which crashes soon after logging the received **verdict**.

If a correct recovery exists, all operative \(P_i, i \neq k\), must decide on commit without waiting for \(P_k\) and \(C\) to recover. Note that execution environment is identical for all operative \(P_i, i \neq k\) in both **scenarios**, but the **verdict** reached ought to be different; so, a correct recovery cannot be designed.

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, would orchestrate the 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, say, \(A\) when multiple sources, e.g., processes in \(\Pi\), can request \(A\)’s execution: BC can be programmed (see smart contract below) to accept only the first request in the total order and ignore the duplicates.

**Wall Clock.** Ordered transactions are first arranged in blocks of fixed size which are then arranged in BC in the increasing order of block timestamps. Assuming that transac-
tions are being continually submitted to BC, the increasing timestamps of the blocks being added constitute a publicly-visible, real-time wall-clock (possibly with irregular ticks); processes of P can use it as a common time-service.

Smart Contract. It 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 is structured as a collection of deterministic functions. Contract code is written in languages like Serpent, LLL or Solidity (our choice in §4.4). The code is compiled into byte-code that will be interpreted by a BC component, such as, Ethereum Virtual Machine.

Ethereum [2]. It is the most popular platform to support smart contract technology and is used in our implementation (§4.5). A user process, such as C, can deploy a smart contract in BC by launching a transaction whose data field contains the Byte-code of the smart contract with parameters appropriately initialised. Once this transaction is accepted in BC, any named process, such as Pi, can invoke a contract function by submitting a transaction. The invoking transaction is constructed with (i) the receiver address pointing to the contract address and (ii) the parameter values for the function call. In addition, in Ethereum, a transaction includes two more fields; GAS and GAS PRICE [2]. The miner who adds the 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 instance when a user process U launches a valid (blockchain) transaction TXi and the instance when a block containing TXi is (irreversibly) added in BC; let α be the awareness bound on the delay that can elapse between the instance when TXi enters BC irreversibly and the instance when any interested party gets aware of TXi in BC. We call the BC infrastructure (together with miner/consensus nodes) synchronous if it supports reliable estimation of finite 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 TXi is periodically scanning BC, etc. (Just like the validity of δ bound requires no message be lost but every message be queued, transmitted, received and delivered - all in a timely manner.) Note that a synchronous BC requires the underlying distributed system to be synchronous.

4.3 2PC with Synchronous Blockchain

We explain here (i) how C hands over the coordination responsibilities for 2PC execution to the BC infrastructure and, (ii) how P, interacts with BC to execute 2PC, i.e., to register its vote and then to receive the verdict.

As in traditional 2PC, C disseminates the work to each P, ∈ Π; it then hands over the responsibilities to the BC infrastructure by launching a (BC) transaction TXC that sets up the 2PC coordination smart contract in BC with initial state = VOTING. (Smart contract code is explained in §4.4.) The role of C ends with launching TXC. Note that C may crash after work dissemination and before launching TXC; in this case, all operative P, must detect this and end up deciding abort as in traditional 2PC execution.

When P, receives work from C, it computes Ti as the local time when a duration that is maximum of {ωi, δ + β + α}, would elapse after the receipt of the work. Ti is the earliest local time when P, can complete its work and become aware of TXC being added to BC, if C had launched TXC.

Thus, if TXC does not appear in BC until a block with timestamp > Ti is added, i.e., until BC wall-clock exceeds Ti, then, by synchrony assumptions, P, can deduce that C crashed without launching TXC: it can subsequently abort as shown by the state transition from W1 to a in Fig 3, where WC denotes the BC wall-clock. The transitions from state WG in Fig 3 are identical to those shown in Fig 2. They have here become off-chain activities [11].

If a P, that completes its work (WG → W1 in Fig 3), gets aware of TXC by local time Ti, it logs locally V1 = 1 (as in Phase 1 of Fig 1) registers its vote by launching TXi to BC. When TXi is accepted in BC, it invokes VOTER function of the smart contract with V1 = 1 as input. (State of P, now transits from W1 to W2 in Fig 3.)

Let TXC.BlkTime be the timestamp of the block containing TXC. Any operative P, gets aware of TXC no later than WC = TXC.BlkTime + α and its TXi, launched in response, would be added to BC by WC ≤ TXC.BlkTime + α + β. (Note: α and β are upper bounds and actual delays can be smaller than them.)

If all Pi vote Vi = 1, then the smart contract would compute verdict = commit and display state = COMMIT in BC. (Details in §4.5.) All Pi observe this state by WC ≤ TXC.BlkTime + 2α + β.

Let Δ = 2α + β. When WC exceeds TXC.BlkTime + Δ, if an operative P, that sent TXi cannot see state = COMMIT in BC, then some Pk did not launch TXi. In that case, P, can safely decide verdict = abort. However, our description here assumes that P, decides verdict = commit or abort only in response to what is being indicated in BC, to be consistent with the traditional 2PC description.

When WC > TXC.BlkTime + Δ and state ≠ COMMIT, P, launches TXV to invoke VERDICT function of the smart contract so that verdict is computed and displayed in BC. In Fig 3, P, does W2 → W3 after launching TXV and then to W4 → a when BC indicates state = ABORT. If several TXV were launched, only one will be effective in executing VERDICT (like A in §4.1).

![Figure 3: State Diagram for 2PC with Blockchain](image-url)
4.4 Smart Contract Pseudo Code

Table 1 presents the pseudo-code of the 2PC coordination smart contract together with its three functions which are called (i) \texttt{REQUEST()} called by \texttt{TXC} to initialise the contract, (ii) \texttt{VOTER()} by \texttt{TXC} to enter the vote of \texttt{Pi} in BC and, (iii) \texttt{VERDICT()} by \texttt{TXC} to request the \texttt{verdict} to be computed.

The description here assumes the following. The contract is already deployed on the blockchain with a unique address. It has an initial state \texttt{INIT}, with three parameters \texttt{Timeout} (initialised to zero), an initially empty set \texttt{A} of named participants and an empty set \texttt{V} of voted participants; its functions have the following interfaces: \texttt{REQUEST(A, Timeout), VOTER(boolean)} and \texttt{VERDICT()}.

\texttt{TXC} submitted by \texttt{C} invokes \texttt{REQUEST} function with \((A = \Pi, Timeout = \Delta)\), where \(\Delta = 2\alpha + \beta\). This initialisation succeeds if \texttt{C} is \texttt{asserted} to have ownership rights to invoke this function and the code is in the initial state \texttt{INIT} - as indicated in the \texttt{Assert} statement. If this assertion succeeds, \texttt{TXC} is accepted and the \texttt{state} of the contract is changed to \texttt{VOTING} which is publicly visible in BC; otherwise, \texttt{TXC} is ignored. (This is always the case: a \texttt{TX} is rejected if the pre-invocation assertion fails; throughout this description, assertions are assumed to succeed, except for duplicate calls on \texttt{VERDICT} function.)

Each \texttt{Pi} in \texttt{W1} checks BC for \texttt{TXC}; when \texttt{state = VOTING}, \texttt{Vi = YES} is sent by submitting \texttt{TXC} that invokes \texttt{VOTER} function. Upon receiving \texttt{TXC}, the contract asserts if \texttt{Pi} is legitimate to vote or not. When \texttt{Pi} is legitimate, \texttt{Pi} is recorded to have voted in the set \texttt{V}. If \texttt{V = \Pi}, then the contract \texttt{state} is changed to \texttt{COMMIT}.

After \(WC = TXC.BlkTime + \Delta\), any \texttt{Pi} in \texttt{W2} that still finds the \texttt{state = VOTING}, invokes \texttt{VERDICT} function by submitting \texttt{TXC}. The invocation succeeds only if (i) \texttt{Pi} \(\in\) \texttt{\Pi}, (ii) sufficient time of \(\Delta = 2\alpha + \beta\) had elapsed since \texttt{TXC} was added into BC and (iii) \texttt{state = VOTING}. If it succeeds, it sets \texttt{state = ABORT}. An attempt to invoke \texttt{VERDICT} when \texttt{state = COMMIT} or \texttt{state = ABORT}, will not meet (iii) and not succeed.

\begin{verbatim}
INIT: Set state := INIT; A := [0x000...0x000] Timeout := 0; V := [0x000...0x000]
REQUEST: Upon C submitting TXC(\Pi, \Delta): Assert (state = INIT and msg.sender == C) Set A := \Pi; Set Timeout := \Delta; state := VOTING.
VOTER: Upon Pi submitting TXC(Vote): Assert (state \in VOTING and msg.sender == Pi) Assert (Pi \notin \Pi) Assert (Vote = \texttt{YES}) Set V := V \cup \{Pi\} if (V := \Pi) then {state := COMMIT;}
VERDICT: Upon Pi submitting TXC: Assert (state = VOTING and msg.sender == Pi) Set: (block.timestamp > TXC.block.timestamp + \Delta) if (state \in VOTING then state := ABORT.
\end{verbatim}

Table 1: Smart Contract pseudo-code for 2PC coordination

4.5 Implementation

We implemented the 2PC-Blockchain contract in Solidity 0.4.10 [4] and tested them on the Ethereum private network with Geth [3]. The experiment is run on a MacBook Pro with a 2.8 GHz Intel i5 CPU and 8 GB RAM. Two experiments are done with one \texttt{C} and two \texttt{Pi}. In the first, both \texttt{Pi} voted. The outcome was \texttt{state} being set to \texttt{COMMIT}. In the second, only one \texttt{Pi} voted and after \(WC = TXC.BlkTime + \Delta\), the \texttt{Pi} that voted launched a transaction that invoked \texttt{VERDICT} function to get the verdict, which set the \texttt{state = ABORT}.

In Table 2, we present the cost of creating and executing the 2PC-Blockchain contract. The cost is in the amount of GAS consumed by each function in 2PC-Blockchain contract converted in US dollars. We used the cheapest GAS PRICE (i.e. \(2 \times 10^{-9}\) ether) in all transactions with exchange rate 1 ether = \$830.61. As can be seen, the financial cost using smart contract for non-blocking 2PC is affordably low.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Cost in GAS</th>
<th>Cost in USD</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>845,550</td>
<td>1.40361</td>
</tr>
<tr>
<td>REQUEST</td>
<td>190,226</td>
<td>0.31581</td>
</tr>
<tr>
<td>VOTING</td>
<td>75,472</td>
<td>0.12525</td>
</tr>
<tr>
<td>VERDICT</td>
<td>55,102</td>
<td>0.09147</td>
</tr>
</tbody>
</table>

Table 2: Cost of using 2PC-Blockchain contract

4.6 2PC with synchronous BC is non-blocking

This is so, for three reasons: (i) the entire system comprising \texttt{\Pi} and BC are synchronous, i.e., the bounds \(\omega, \delta, \beta\) and \(\alpha\) are never violated (ii) BC is crash-free, and (iii) the BC wall-clock continues to tick as new transactions (including those outside of 2PC execution) are presumed to be continually submitted to BC. Therefore, if \(C\) launches \texttt{TXC}, it would never be the case that some \texttt{Pi} sees \texttt{TXC} in BC before \texttt{Tj} and another \texttt{Pi} sees it after \texttt{Tj}.

From (non-)blocking perspective, only two cases exist: \texttt{C} does or does not launch \texttt{TXC}. Consider the former; all transitions shown in blue in Fig 3 are due to \texttt{Pi} interacting with BC in a timely environment, and must occur in bounded time unless \texttt{Pi} itself crashes. If, on the other hand, \texttt{C} crashes before launching \texttt{TXC}, it is detected by all operative \texttt{Pi} that completed the work, from the absence of \texttt{TXC} in BC even after \(WC > Tj\). Thus, they all unilaterally, and also identically, decide on \texttt{abort}, i.e., all transit from \texttt{W1} to \(a\) in Fig 3, while those that could not complete the work transit from \texttt{WG}\(\rightarrow\) to \(a\). Thus, an operative \texttt{Pi} cannot block.

5. ASYNCHRONOUS BLOCKCHAINS

When bounds \(\alpha\) and \(\beta\) cannot be reliably estimated, BC becomes asynchronous. This can invalidate two features pivotal for the correctness of BC based 2PC: (i) for all operative \texttt{Pi}: \texttt{C} has crashed if BC does not add \texttt{TXC} latest by \(WC = Tj\), and (ii) \texttt{commit-validity}: \texttt{verdict = commit} when all \texttt{Pi} launch \texttt{TXC} with \texttt{Vote = 1}. INVALIDATION of (i) occurs when \texttt{TXC} takes much longer to enter BC than the estimated bound; similarly, (ii) is not met when the entry of \texttt{TXC} from \texttt{Pi} into BC is so delayed that \texttt{state = VOTING} even after \(WC > TXC.BlkTime + \Delta\) and \texttt{TXC} from some \texttt{Pi}, \(i \neq j\), meanwhile makes BC compute \texttt{verdict} as \texttt{abort}.

Note that a public BC can be asynchronous even if the underlying distributed system is synchronous. For example, if miners, at the time of \texttt{TXC} launch, also encounter other transactions that are more financially attractive to work on compared to \texttt{TXC}, then \texttt{TXC} could take longer to enter BC, if at all, than any \(\beta\) estimated in more favourable environments [12]. This raises two interesting questions: can we have a non-blocking 2PC, given that BC being used is asynchronous and the cluster hosting \texttt{\Pi} is (1) synchronous
and (2) asynchronous? The answer to (1) is open, though we believe it can be yes, and that to (2) is a definite no.

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

Proof (by contradiction). Let us first observe that since the cluster is asynchronous, crash of a \( P_k \) cannot be perfectly detected by an operative \( P_i \). To simplify the proof, let us assume that \( C \) never crashes and always offloads 2PC coordinating responsibilities to BC. So, any \( P_i \) will certainly observe \( TXC \) in BC if it waits indefinitely. (Note: if the impossibility is shown to hold with a valid simplification, it must also hold when the latter does not apply.)

Let us hypothesise, to the contrary, that the impossibility claim is incorrect and that there exists a non-blocking 2PC protocol \( \mathcal{P} \). Consider two executions of \( \mathcal{P} \) in which (i) every process, except \( P_k \) in the first execution, is operative and wants to commit, and (ii) if a given \( P_i \) is operative in an execution, then it observes \( TXC \) in BC in a timely manner, i.e., before its local time \( T_i \), and launches \( TX_i \).

Since \( \mathcal{P} \) solves atomic commit, each operative process that wants to commit, decides eventually; say, for some \( D, P_i, i \neq k \), decides before its local time \( T_i + D \) in both executions.

**Execution 1**: \( P_k \) does \( WG \rightarrow a \) and then crashes. Further, \( P_k \) does not recover in this execution until local time of \( P_i \) exceeds \( T_i + D \). By the hypothesis on \( \mathcal{P} \), \( P_i \) must decide here on \( abort \) though \( P_k \) remains crashed until \( T_i + D \).

**Execution 2**: \( P_k \) does not crash and launches its \( TX_k \) at local time \( T_k \). \( TX_k \) does not enter BC until after the clock of \( P_i \) reads \( T_i + D \); this is possible because BC is asynchronous. Moreover, any message that is ever sent by \( P_k \) does not reach its destination until after the clock of \( P_i \) reads \( T_i + D \); this is also possible because the database cluster is also asynchronous.

**Execution 2** is indistinguishable for any \( P_i, i \neq k \) from **Execution 1** until its clock time \( T_i + D \). So, as in **Execution 1**, \( P_i \) must decide on \( abort \) before \( T_i + D \). This violates (ii) above which is also the **commit-validity** requirement. (See Section 2.) Thus, the hypothesis about \( \mathcal{P} \) is contradicted and the impossibility proved.

5.1 Impossibility in Practice

A closer look at the proof suggests that asynchrony prevents only **commit-validity** from being guaranteed; the other three requirements are met when delay bound estimates are violated. In practice, the bounds \( \alpha \) and \( \beta \) can be estimated with a reasonable accuracy that they hold with a high probability, say, \( p \). Thus, with a probability \( (1 - p) \), \( TX_c \) will be so delayed that the **verdict** is computed as **abort** before \( TX_c \) enters BC; the **verdict** ought to have been **commit** if all other \( P_i \) also voted **yes**. Similarly, if \( TX_c \) is unduly delayed, **abort** would be decided where synchrony could have resulted in **commit**. So, **abort** cannot be unnecessarily decided if \( TX_c \) and \( n \ TX_i \) are timely, which occurs with probability \( p^{(n+1)} \); thus, if, say, \( n = 4 \) and \( p = 99\% \), then \( 1 - p^{(n+1)} = 5\% \) is the probability that a transaction is unnecessarily aborted.

6. CONCLUDING REMARKS

A popular choice to avoid 2PC blocking is to use 3 phase commit. We have shown here that the extra phase is not needed and blocking can be eliminated in a synchronous cluster by having a synchronous blockchain to coordinate the 2PC execution. If both blockchain and cluster are asynchronous, the former being crash-free is not sufficient to eliminate blocking. We believe that eVoting (even without privacy concerns) is harder than non-blocking commit and its blockchain implementations (e.g., [16]) must require some synchrony guarantees to be correct.

7. REFERENCES