Increasing the Resilience of Atomic Commit, at No Additional Cost \(^1\)

Idit Keidar and Danny Dolev
Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Technical Report CS94-18

October, 1994

\(^1\) This work supported by the United States - Israel Binational Science Foundation, Grant No. 92-00189
Abstract

This paper presents a new atomic commitment protocol that *always* allows a quorum in the system to make progress. Previously suggested quorum-based protocols (*e.g.* [12]) allow a quorum to make progress in case of one failure. If failures cascade, however, and the quorum in the system is “lost” (*i.e.* at a given time no quorum component exists, *e.g.* because of a total crash), a quorum can later become connected and still remain blocked. The importance of this work is in demonstrating, using a simple algorithm, how protocols that always allow a majority to make progress can be constructed.
1 Introduction

Reliability and availability of loosely coupled distributed database systems is becoming a requirement for many installations, and fault-tolerance is becoming an important aspect of distributed systems design. When processors crash, or when communication failures occur, it is desirable to allow as many processors as possible to make progress.

In this paper we present a novel atomic commitment protocol (ACP) that always allows a majority (or quorum) to make progress. Previously suggested quorum-based protocols (e.g. [12]) allow a quorum to make progress in case of one failure. If failures cascade, however, and the quorum in the system is “lost” (i.e. at a given time no quorum component exists, e.g. because of a total crash), a quorum can later become connected and still remain blocked. To our knowledge, the only previously suggested ACP that always allows a quorum to make progress is the ACP that we construct in [10]. The protocol in [10] is not straightforward; it uses a replication service as a building block, while the protocol presented in this paper is easy to follow and self-contained.

The protocol we present is resilient to site failures and network partitioning. Sites may crash and recover. The network may partition into several components\(^1\), and remerge. Whenever a failure is detected, a special recovery procedure is invoked. The protocol does not require that failures be correctly identified in order to work correctly. Undetected failures and false reports of failures may cause degradation in performance. In the absence of false failure reports, a quorum of connected processors may always reach a decision: At any point in the execution of the protocol, if a group G of processors becomes connected, and this group contains a quorum and no subsequent failures occur for sufficiently long, then all the members of G eventually reach a decision. Furthermore, every processor that can communicate with a processor that already reached a decision, will also, eventually, reach a decision. An operational processor that is not a member of a connected quorum may be blocked, i.e. may have to wait until a failure is repaired in order to resolve the transaction. This is undesirable but cannot be avoided; every protocol that tolerates network partitions is bound to be blocking [13].

The importance of this work is in demonstrating, using a simple algorithm, how protocols that always allow a majority to make progress can be constructed. In this paper we present the Enhanced Three Phase Commit (E3PC) protocol, which is a simple extension of the quorum-based three phase commit (3PC) protocol [12]. E3PC achieves higher availability, just by maintaining two additional counters, and with no additional communication. The principles demonstrated in this paper can be used to increase the resilience of a variety of distributed systems, e.g. replicated database systems.

A common way to increase the availability of data and services is replication. If data is replicated in several sites, it can still be available despite site and communication-link failures. Numerous replication schemes that are based on quorums were suggested [6, 8, 9, 3, 4]. These algorithms use quorum systems to determine when data objects are accessible. In order to guarantee the atomicity of transactions, these algorithms use an ACP, and therefore are bound to block when the ACP they use blocks. Thus, with previously suggested ACPs, these approaches do not always allow a connected majority to update the database. Using our ACP these protocols can be made more resilient.

\(^1\)A component is sometimes called a partition. In our terminology, a partition splits the network into several components.
The rest of this paper is organized as follows: Section 2 presents the computation model. Section 3 provides background on distributed transaction management and atomic commitment. The quorum-based three phase commit (3PC) protocol [12] is described in Section 4, and enhanced three phase commit (E3PC), in Section 5. Section 6 concludes the paper. In Appendix A we formally prove the correctness of E3PC.

2 The Model

We assume that the set of processors running the protocol is fixed, and is known to all the processors. We assume that the processors are connected by an underlying communication network, that provides reliable FIFO communication between any pair of connected processors. We consider the following types of failures: failures may partition the network, and previously disjoint network components may re-merge. Sites may crash and recover; recovered processors come up with their stable storage intact. We assume that failures are detected using a (possibly unreliable) fault detector, e.g. a timeout mechanism. While no failures are reported, messages can be delayed arbitrarily, but are not lost.

2.1 Quorums

We use a quorum system to decide when a group of connected processors may resolve the transaction. To enable maximum flexibility we allow the quorum system to be elected in a variety of ways (e.g., weighted voting). For further flexibility, it is possible to set different quorums for commit and abort. In this case, a Commit Quorum of connected processors is required in order to commit a transaction, and an Abort Quorum – to abort. For example, to increase the probability of commit in the system, one can assign smaller quorums for commit, and larger ones for abort.

We assume two predicates: $Q_C(G)$ is true for a given group of processors $G$ iff $G$ is a Commit Quorum; and $Q_A(G)$ is true iff $G$ is an Abort Quorum. The requirement from these predicates is that for any two groups of processors $G$ and $G'$ such that $G \cap G' = \emptyset$, at most one of $Q_C(G)$ and $Q_A(G')$ holds, i.e. every Commit Quorum intersects every Abort Quorum. Numerous quorum systems that fulfill these criteria were suggested. An analysis of the availability of different quorum systems may be found in [11].

3 Background – Distributed Transaction Management

In distributed databases when a transaction spans several sites the database servers at all sites have to reach a common decision whether the transaction should be committed or not. A mixed decision results in an inconsistent database, a unanimous decision guarantees the atomicity of the transaction (provided that each local server can guarantee local atomicity of transactions).

3.1 Problem Definition

A distributed transaction is composed of several sub-transactions, each running on a different site. The database manager at each site can unilaterally decide to ABORT the local sub-transaction, in

---

2 We assume that the underlying communication discovers the loss and either recovers it, or reports of a failure.
which case the entire transaction must be aborted. If all the participating sites agree to COMMIT their sub-transaction (vote **Yes** on the transaction) and no failures occur, the transaction should be committed. We assume that the local database server at each site can atomically execute the sub-transaction, once it has agreed to COMMIT it.

In order to ensure that all the sub-transactions are consistently committed or aborted, the processors run an **atomic commitment protocol (ACP)** such as **two phase commit (2PC)**. The requirements of atomic commitment (as defined in Chapter 7 of [1]) are:

- All the processors that reach a decision reach the same one.
- A processor cannot reverse its decision after it has reached one.
- The **COMMIT** decision can only be reached if all processors voted **Yes**.
- If there are no failures and all processors voted **Yes**, then the decision will be to **COMMIT**.
- At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all processors will eventually reach a decision.

### 3.2 Two Phase Commit

The simplest and most renowned ACP is **two phase commit (2PC)** [7]. Several variations of 2PC were suggested, the simplest version is centralized – one of the processors is designated as the **coordinator**. The coordinator sends a transaction (or request to prepare to commit) to all the participants. Each processor answers by a **Yes** (“ready to commit”) or by a **No** (“abort”) message. If any processor votes No, all the processors abort. The coordinator collects all the responses and informs all the processors of the decision. In absence of failures, this protocol preserves atomicity. Between the two phases, each processor **blocks, i.e.** keeps the local database locked, waiting for the final word from the coordinator. If a processor fails before its vote reaches the coordinator, it is usually assumed that it had voted No. If the coordinator fails, all the processors remain blocked indefinitely, unable to resolve the last transaction. The centralized version of 2PC is depicted in Figure 1.

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction is received: Send sub-transactions.</td>
<td>Sub-transaction is received: Send reply – <strong>Yes</strong> or <strong>No</strong>.</td>
</tr>
<tr>
<td>If all sites respond <strong>Yes</strong>: Send <strong>COMMIT</strong>. If some site voted <strong>No</strong>: Send <strong>ABORT</strong>.</td>
<td>COMMIT or ABORT is received: Process accordingly.</td>
</tr>
</tbody>
</table>

Figure 1: The Centralized Two Phase Commit Protocol
Commit protocols may also be described using state diagrams [13]. The state diagram for 2PC is shown in Figure 1. In this protocol, each processor (either coordinator or slave) can be in one of four possible states:

\( q \) : INITIAL state – A site is in the initial state until it decides whether to unilaterally abort or to agree to commit the transaction.

\( w \) : WAIT state – In this state the coordinator waits for votes from all of the slaves, and each slave waits for the final word from the coordinator. This is the “uncertainty period” for each processor, when it is unknown whether the transaction will be committed or not.

\( c \) : COMMIT state – Decision to commit was made.

\( a \) : ABORT state – Decision to abort was made.

The states of a commit protocol may be classified along two orthogonal lines. In the first dimension, the states are divided into two disjoint subsets: The committable states and the non-committable states. A processor is in a committable state only if it knows that all the sites have agreed to proceed with the transaction. The rest of the states are non-committable. The only committable state in 2PC is the COMMIT state. The second dimension, distinguishes between final and non-final states. The final states are the ones in which a decision has been made, and no more state transitions are possible. The final states in 2PC are COMMIT and ABORT.

3.3 The Extent of Blocking in Commit Protocols

The 2PC protocol is an example of a blocking protocol: operational sites sometimes wait on the recovery of failed sites. Locks must be held in the database while the transaction is blocked. Even though blocking preserves consistency, it is highly undesirable because the locks acquired by the blocked transaction cannot be relinquished, rendering the data inaccessible by other requests. Consequently, the availability of data stored in reliable sites can be limited by the availability of the weakest component in the distributed system.

Skeen et al. proved [13] that there exists no non-blocking protocol resilient to network partitioning. When a partition occurs, the best protocols allow no more than one group of sites to continue while the remaining groups block. Skeen suggested the quorum-based three phase commit protocol, that maintains consistency in spite of network partitions [12]. This protocol is blocking; it is possible for an operational site to be blocked until a failure is mended. In case of failures, the algorithm uses a quorum (or majority) based recovery procedure, that allows a quorum to resolve the transaction. If failures cascade, however, a quorum of processors can become connected and still remain blocked. Skeen’s quorum-based commit protocol is described in Section 4.

As it was proved that completely non-blocking recovery is impossible to achieve, further research in this area concentrated on minimizing the number of blocked sites when partitions occur. Chin et al. [2] define optimal termination protocols (recovery procedures) in terms of the average number of sites that are blocked when a partition occurs. The average is over all the possible partitions, and all the possible states in the protocol in which the partitions occurs. The analysis deals only with states in the basic commit protocol, and ignores the possibility for cascading failures (failures that occur during the recovery procedure). It is proved that any ACP with optimal recovery procedures takes at least three phases, and that the quorum-based recovery procedures are optimal.
In [10] we construct an ACP that always allows a connected majority to proceed, regardless of past failures. To our knowledge, no other ACP with this feature was suggested. The ACP suggested in [10] uses a reliable replication service as a building block, and is mainly suitable for replicated database systems. In this paper, we present a novel commitment protocol, *Enhanced Three Phase Commit (E3PC)*, that always allows a connected majority to resolve the transaction (if it remains connected for sufficiently long). E3PC does not require complex building blocks such as the one in [10], and is more adequate for partially replicated, or non-replicated distributed database systems; it is similar to the quorum-based three phase commit.

4 Quorum-Based Three Phase Commit

In this section we describe Sken’s quorum-based commit protocol [12]. The basic *three phase commit (3PC)* is described in Section 4.1, and the recovery procedure is described in Section 4.2. In Section 4.3 we show that with 3PC a connected majority of the processors can be blocked.

4.1 Basic Three Phase Commit

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction is received: Send sub-transactions to slaves.</td>
<td>Sub-transaction is received: Reply – Yes or No.</td>
</tr>
<tr>
<td>If all sites respond <strong>Yes</strong>: Send PRE-COMMIT.</td>
<td></td>
</tr>
<tr>
<td>If some site voted <strong>No</strong>: Send ABORT.</td>
<td>PRE-COMMIT received: Send ACK.</td>
</tr>
<tr>
<td>Upon receiving a Commit Quorum of ACKs: Send COMMIT.</td>
<td>COMMIT or ABORT is received and processed.</td>
</tr>
<tr>
<td>Otherwise: Block (wait for more votes or until recovery)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2: The Quorum-Based Three Phase Commit Protocol**

The 3PC protocol is similar to two phase commit, but in order to achieve resilience, another non-final “buffer state” is added in 3PC, between the **WAIT** and the **COMMIT** states:

**pc**: PRE-COMMIT state – this is an intermediate state before the commit state, and is needed to allow for recovery. In this state the processor is still in its “uncertainty period”.

The COMMIT and PRE-COMMIT states are *committable* states; a processor may be in one of these states only if it knows that all the site have agreed to proceed with the transaction. The rest of the states are *non-committable*. In each step of the protocol, when the processors change their state, they must write the new state to stable storage, before replying to the message that caused the state change. The quorum-based 3PC is described in Figure 2, and the corresponding state diagram is depicted in Figure 3 (a).

4.2 Recovery Procedure for Three Phase Commit

When a group of processors detect a failure (a processor crash or a network partition), or a failure repair (processor recovery or merge of previously disconnected network components), they run the
recovery procedure in order to try to resolve the transaction (i.e. commit or abort it). The recovery procedure consists of two phases: first elect a new coordinator, and next attempt to form a quorum that can resolve the transaction.

A new coordinator may be elected in different ways [5]. In the course of the election, the coordinator hears from all the other participating processors. If there are failures (or recoveries) in the course of the election, the election can be restarted. Election is a weaker problem than atomic commitment, only the coordinator needs to know that it was elected, while the other processors may crash or detach without ever finding out which processor was elected. If the elected coordinator can no longer communicate with the processors that elected it, a new election can be restarted.

Once a coordinator is elected, it executes a protocol similar to the commit protocol described in the previous section. The new coordinator tries to reach a decision whether the transaction is committable or not, and then tries to form a quorum for its decision (a Commit Quorum for COMMIT and an Abort Quorum for ABORT). If the new coordinator does not know if the transaction is committable, it will try to abort the transaction.

As before, the protocol must take the possibility of failures and failure repairs into account, and furthermore, must take into account the possibility of two (or more) different coordinators existing concurrently in disjoint network components. In order to ensure that the decision will be consistent, a coordinator must explicitly establish a Commit Quorum for COMMIT, or an Abort Quorum for ABORT. To this end, in the recovery procedure, another state is added:

\textbf{pa} : PRE-ABORT state. Dual state to PRE-COMMIT.

The recovery procedure is described in Figure 4. The state diagram for the recovery procedure is shown in Figure 3 (b). The dashed lines represent transitions in which this processor’s state was
not used in the decision made by the coordinator. For example, a processor can be in the \texttt{PRE-ABORT} state when its coordinator decides to \texttt{COMMIT}, because some other participating processor is already \texttt{COMMITTED}.

1. Elect a new coordinator, $r$.
2. The coordinator, $r$, collects the states from all the connected processors.
3. The coordinator tries to reach a decision, as described in Figure 5. The decision is computed using the states collected so far. The coordinator multicasts a message reflecting the decision.
4. Upon receiving a \texttt{PRE-COMMIT} or \texttt{PRE-ABORT} each slave sends an ACK to $r$.
5. Upon receiving ACKs for \texttt{PRE-COMMIT} from a \textit{Commit Quorum} or ACKs for \texttt{PRE-ABORT} from an \textit{Abort Quorum}, $r$ multicasts the corresponding decision: \texttt{COMMIT} or \texttt{ABORT}.
6. Upon receiving a \texttt{COMMIT} (\texttt{ABORT}) message: Process the transaction accordingly.

\begin{figure}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Collected States & Decision \\
\hline
$\exists$ \texttt{ABORTED} & \texttt{ABORT} \\
$\exists$ \texttt{COMMITTED} & \texttt{COMMIT} \\
$\exists$ \texttt{PRE-COMMITTED} $\land Q_C$ (processors in \textit{WAIT} and \texttt{PRE-COMMIT} states) & \texttt{PRE-COMMIT} \\
$Q_A$ (processors in \textit{WAIT} and \texttt{PRE-ABORT} states) & \texttt{PRE-ABORT} \\
Otherwise & \texttt{BLOCK} \\
\hline
\end{tabular}
\caption{The Decision Rule for The Quorum-Based Recovery Procedure}
\end{figure}

\textbf{4.3 Three Phase Commit Blocks a Quorum}

In this section we show that in the algorithm described above, it is possible for a quorum to become connected and still remain blocked. In our example, there are three processors executing the transaction – $p_1$, $p_2$ and $p_3$. The quorum system we use is a simple majority: every two processors form a quorum, and the same quorums are designated for both commit and abort. Consider the following scenario:

$p_1$ is the coordinator. All the processors vote \texttt{Yes} on the transaction. $p_1$ receives and processes the votes, but $p_2$ and $p_3$ detach from $p_1$ before hearing the vote outcome (\textit{i.e.} before receiving the \texttt{PRE-COMMIT} message sent by $p_1$).
p_2 is elected as the new coordinator. It sees that both p_2 and p_3 are in the wait state, and therefore sends a PRE-ABORT message, according to the decision rule. p_3 receives the PRE-ABORT message, acknowledges it, and then detaches from p_2.

Now, p_3 is in the PRE-ABORT state, while p_1 is in the PRE-COMMIT state. If now, p_1 and p_3 become connected, then according to the decision rule, they remain BLOCKED, even though they form a quorum.

**Analysis**

In this example, it is actually safe for p_1 and p_3 to decide PRE-ABORT, because none of the processors could have committed, but it is not safe for them to decide PRE-COMMIT, because p_3 cannot know if p_2 has aborted or not.

We observe that p_3 decided PRE-ABORT “after” p_1 decided PRE-COMMIT, and therefore we can conclude that the PRE-COMMIT decision made by p_1 is “stale”, and no processor has actually reached a COMMIT decision following it. Because otherwise, it would have been impossible for p_2 to reach a PRE-ABORT decision.

The 3PC protocol does not allow a decision in this case, because the processors have no way of knowing which decision was made “later”. Had the processors known that the a PRE-ABORT decision was made “later”, they could have decided PRE-ABORT again, and would have eventually ABORTED the transaction. In E3PC, we provide the mechanism for doing exactly that.

## 5 The E3PC Protocol

We suggest a three phase atomic commitment protocol, *Enhanced Three Phase Commit (E3PC)*, with a novel quorum-based recovery procedure that always allows a quorum of processors to resolve the transaction, even in the face of cascading failures. The protocol is based on the quorum-based three phase commit protocol [12]. E3PC does not require more communication or time than 3PC; the improved resilience is achieved simply by maintaining two additional counters.

Initially, the basic enhanced three phase commit is invoked. If failures occur, the processors invoke the recovery procedure and elect a new coordinator. The new coordinator carries on the protocol to reach a decision. If failures cascade, the recovery procedure may be re-invoked an arbitrary number of times. Thus, one execution of the protocol, (for one transaction), consists of one invocation of the basic E3PC, and of zero or more invocations of the recovery procedure.

In Section 4 we described the quorum-based centralized three phase commit (3PC) protocol, and its recovery procedure. In Section 5.1 we describe how E3PC enhances 3PC. The recovery procedure for E3PC is described in Section 5.2. In Section 5.3 we show that E3PC does not block in the example of Section 4.3. In Section 5.4 we outline the correctness proof for E3PC.

### 5.1 E3PC: Enhancing Three Phase Commit

The basic E3PC is similar to the basic 3PC, the only difference is that E3PC maintains two additional counters. We now describe these counters. In each invocation of the recovery procedure, the processors try to elect a new coordinator. The coordinators elected in the course of an execution of the protocol are sequentially numbered: A new “election number” is assigned in each invocation of the recovery procedure. Note: there is no need to elect a new coordinator in each invocation of
the basic 3PC, or E3PC, the re-election is only needed in case failures occur. The coordinator of the basic E3PC is assigned the “election number” one, even though no elections actually take place. The following two counters are maintained by the basic E3PC, and by the recovery procedure:

**Last_Elected** The number of the last election that this processor took part in. This variable is updated when a new coordinator is elected. This value is initialized to one when the basic E3PC is invoked.

**Last_Attempt** The election number in the last attempt this processor made to commit or abort. The coordinator changes this variable’s value to the value of Last_Elected whenever it makes a decision. Every other participant sets its Last_Attempt to Last_Elected when it moves to the PRE-COMMIT or to the PRE-ABORT state, following a PRE-COMMIT or a PRE-ABORT message from the coordinator. This value is initialized to zero when the basic E3PC is invoked.

These variables are logged on stable storage. When a processor changes its state, it first writes to stable storage the new state and the new values of these variables (if needed), and only then sends a response to the message that caused the state change. Note: State changes must be written to stable storage in 3PC as well as in E3PC.

The second counter, Last_Attempt, provides a linear order on PRE-COMMIT and PRE-ABORT decisions, e.g. if some processor is in the PRE-COMMIT state with its Last_Attempt= 7, and another processor is in the PRE-ABORT state with its Last_Attempt= 8, then the PRE-COMMIT decision is “earlier” and therefore “stale”, and the PRE-ABORT decision is safe. The first counter, Last_Elected, is needed to guarantee that two contradicting attempts (i.e. PRE-COMMIT and PRE-ABORT), will not be made with the same value of Last_Attempt (cf. Lemma 3).

**Notation**

We use the following notation:

- $\mathcal{P}$ is the group of processors that are live and connected, and take part in the election of the new coordinator.
- $Max_{Elected}$ is $\max_{p \in \mathcal{P}} (Last_{Elected}$ of $p)$.
- $Max_{Attempt}$ is $\max_{p \in \mathcal{P}} (Last_{Attempt}$ of $p)$.
- $Is_{Max_{Attempt}Committable}$ is a predicate that is true iff all the members that are in non-final states and their Last_Attempt is equal to Max_Attempt are in a committable state.

**5.2 Quorum Based Recovery Procedure**

The recovery procedure is invoked when failures are detected and when failures are repaired. A processor that cannot communicate with its coordinator invokes the recovery procedure to elect a surrogate coordinator. A blocked coordinator that notices that communication with previously disconnected processors was restored will re-invoke the recovery procedure. Processors cannot “join” the recovery procedure in the middle, instead, the recovery procedure must be re-invoked to let them take part. A processor that hears from a new coordinator ceases to take part in the previous invocation that it took part in, and no longer responds to its previous coordinator. Thus, a
processor cannot concurrently take part in two invocations of the recovery procedure. Furthermore, if a processor responds to messages from the coordinator in some invocation, it necessarily took part in the election of that coordinator.

The recovery procedure for E3PC is similar to the quorum-based recovery procedure described in Section 4.2. As in 3PC, in each step of the recovery procedure, when the processors change their state, they must write the new state to stable storage, before replying to the message that caused the state change. The recovery procedure is described in Figure 6. The possible state transitions in E3PC and its recovery procedure are the same as those of 3PC, depicted in Figure 3.

In Step 1 of the recovery procedure, the processors elect a coordinator $r$. In the course of the election, $r$ hears from all the other processors their value of Last_Elected and determines Max_Elected. Once it is elected, $r$ notifies the other members of its election and of the value of Max_Elected. Each processor that receives this message responds by setting Last_Elected to Max_Elected+1 and sending its local state and its local value of Last_Attempt to $r$.

In Step 3, $r$ collects the states from the other processors, and tries to reach a decision. It is possible to reach a decision before collecting the states from all the processors in $P$. The processors are blocked until $r$ receives enough states to allow a decision. We denote by $S$ the set of processors from which $r$ received the state so far; $r$ constantly tries to compute the decision using the states in $S$, whenever new states arrive, and until a decision is reached. The decision rule is described below. If the decision is not BLOCK, then in Step 4, $r$ writes to stable storage its new state and the new value of Last_Attempt, and then multicasts the decision to all the processors.

In Step 5, each slave atomically writes to stable storage two values: its new state (PRE-COMMIT or PRE-ABORT) and the new value of Last_Attempt. After these values are safely logged, the processor responds with an acknowledgment to $r$.

**Decision Rule**

The coordinator collects the states and the values of Last_Attempt from the live members, and applies the following decision rule to the set $S$ of processors from which it received the state.

- If there exists a processor (in $S$) that is in the ABORTED state – ABORT.
- If there exists a processor in the COMMITTED state – COMMIT.
- If Is_Max_Attempt_Committable is TRUE, and $S$ is a Commit Quorum – PRE-COMMIT.
- If Is_Max_Attempt_Committable is FALSE and $S$ is an Abort Quorum – PRE-ABORT.
- Otherwise – BLOCK.

The decision rule is summarized in Figure 7. It is easy to see, from the new decision rule, that if a group of processors is both a Commit Quorum and an Abort Quorum, it will never be blocked.

**5.3 E3PC does not Block**

We now show that E3PC does not block with the scenario of Section 4.3. In this example, there are three processors executing the transaction – $p_1$, $p_2$ and $p_3$, and the quorum system is a simple majority: every two processors form a quorum, and the same quorums are designated for both commit and abort. We considered the following scenario:
1. Elect a new coordinator \( r \). The election is non-blocking, it is restarted in case of failure. In the course of the election, \( r \) hears from all the other processors their value of \( \text{Last\_Elected} \) and determines \( \text{Max\_Elected} \). \( r \) notifies the processors in \( \mathcal{P} \) of its election, and of the value of \( \text{Max\_Elected} \).

2. Upon hearing \( \text{Max\_Elected} \) from \( r \), set \( \text{Last\_Elected} \) to \( \text{Max\_Elected} + 1 \) and send local state and \( \text{Last\_Attempt} \) to \( r \).

3. The coordinator, \( r \) collects states from the other processors, and tries to reach a decision as described in Figure 7. The decision is computed using the states collected so far, we denote by \( \mathcal{S} \) the set of processors from which \( r \) received the state so far.

4. Upon reaching a decision other than \text{BLOCK}, \( r \) proceeds as follows:
   - Set \( \text{Last\_Attempt} \) to \( \text{Last\_Elected} \).
   - Change state according to the decision.
   - Multicast the decision to all the processors.

5. Upon receiving a \text{PRE-COMMIT} or \text{PRE-ABORT} each slave responds as follows:
   - Set \( \text{Last\_Attempt} \) to \( \text{Last\_Elected} \).
   - Change state to \text{PRE-COMMIT} or \text{PRE-ABORT}, accordingly.
   - Send an Acknowledgment (\text{ACK}) to \( r \).

6. Upon receiving ACKs for \text{PRE-COMMIT} from a \text{Commit Quorum} or ACKs for \text{PRE-ABORT} from an \text{Abort Quorum}, \( r \) multicasts the corresponding decision: \text{COMMIT} or \text{ABORT}.

7. Upon receiving a \text{COMMIT (ABORT)} message from \( r \): Change state to \text{COMMIT (ABORT)}, and process the transaction accordingly.

Figure 6: The Recovery Procedure for E3PC

- Initially, \( p_1 \) is the coordinator, and \( \text{Last\_Elected}_{p_1} = \text{Last\_Elected}_{p_2} = \text{Last\_Elected}_{p_3} = 1 \). All the processors vote \textbf{Yes} on the transaction. \( p_1 \) receives and processes the votes, but \( p_2 \) and \( p_3 \) detach from \( p_1 \) before hearing the vote outcome (\textit{i.e.} before receiving the \text{PRE-COMMIT} message sent by \( p_1 \)). Now \( \text{Last\_Attempt}_{p_1} = 1 \) while \( \text{Last\_Attempt}_{p_2} = \text{Last\_Attempt}_{p_3} = 0 \), and the value of \( \text{Last\_Elected} \) is still one for all the processors.

- \( p_2 \) is elected as the new coordinator, and the new \( \text{Last\_Elected} \) is two. It sees that both \( p_2 \) and \( p_3 \) are in the \text{WAIT} state, and therefore sends a \text{PRE-ABORT} message, according to the decision rule, and moves to the \text{PRE-ABORT} state while changing its \( \text{Last\_Attempt} \) to two. \( p_2 \) receives the \text{PRE-ABORT} message, sets its \( \text{Last\_Attempt} \) to two, sends an acknowledgment, and detaches from \( p_2 \).

- Now, \( p_3 \) is in the \text{PRE-ABORT} state with its \( \text{Last\_Attempt} = 2 \), while \( p_1 \) is in the \text{PRE-COMMIT} state with its \( \text{Last\_Attempt} = 1 \). If now, \( p_1 \) and \( p_3 \) become connected, then, according to the
<table>
<thead>
<tr>
<th>Collected States</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃ ABORTED</td>
<td>ABORT</td>
</tr>
<tr>
<td>∃ COMMITTED</td>
<td>COMMIT</td>
</tr>
<tr>
<td>Is_MaxAttempt_Committable ∧ QC(S)</td>
<td>PRE-COMMIT</td>
</tr>
<tr>
<td>¬Is_MaxAttempt_Committable ∧ QA(S)</td>
<td>PRE-ABORT</td>
</tr>
<tr>
<td>Otherwise</td>
<td>BLOCK</td>
</tr>
</tbody>
</table>

Figure 7: The Decision Rule for The Recovery Procedure of E3PC

decision rule, they decide to PRE-ABORT the transaction, and they do not remain blocked.

We have shown that E3PC does not block a majority in the scenario in which Sken’s quorum-based 3PC did block. It is easy to see, from the new decision rule, that if a group of processors forms both a Commit Quorum and an Abort Quorum, it will never be blocked.

5.4 Correctness of E3PC

In Appendix A we formally prove that E3PC fulfills the requirements of atomic commitment described in Section 3.1. In this section we describe the proof’s outline.

In Lemma 3 we prove that two contradicting attempts (i.e. PRE-COMMIT and PRE-ABORT), cannot be made with the same value of Last\_Attempt. This is true due to the fact that every Commit Quorum intersects every Abort Quorum, and that a Commit Quorum of processors must increase Last\_Elected before a PRE-COMMIT decision, and an Abort Quorum, before PRE-ABORT. Moreover, Last\_Attempt is set to the value of Last\_Elected, which is higher than the previous value of Last\_Elected of all the participants of the recovery procedure. In Lemma 5 we prove that the value of Last\_Attempt at each processor increases every time the processor changes state from a commitable state to a non-final non-commitable state, and vice versa.

Using the two lemmas above we prove Lemmas 6 and 8: If the coordinator reaches a COMMIT (ABORT) decision upon receiving a Commit Quorum (Abort Quorum) of ACKs for PRE-COMMIT (PRE-ABORT) when setting its Last\_Attempt to i, then for every j ≥ i no coordinator will decide PRE-ABORT (PRE-COMMIT) when setting its Last\_Attempt to j. We prove these lemmas by induction on j ≥ i; we show, by induction on j, that if some coordinator r sets its Last\_Attempt to j in Step 4 of the recovery procedure, then Is\_Max\_Attempt\_Committable is TRUE (FALSE) in this invocation of the recovery procedure, and therefore, the decision is PRE-COMMIT (PRE-ABORT).

We conclude that if some processor running the protocol COMMITS the transaction, then no other processor ABORTS the transaction.

6 Conclusions

In this paper we demonstrated how the three phase commit (3PC) [12] protocol can be made more resilient simply by maintaining two additional counters. The new protocol always allows a quorum of connected processors to resolve a transaction: At any point in the execution of the protocol, if a group G of processors becomes connected, and this group contains a quorum\(^3\) of the processors,

\(^3\)If different quorums are used for commit and abort, then we say the group contains a quorum if it contains both a Commit Quorum and an Abort Quorum.
and no subsequent failures occur for sufficiently long, then all the members of $G$ eventually reach a decision. Furthermore, every processor that can communicate with a processor that already reached a decision, will also, eventually, reach a decision. We have shown that 3PC does not posses this feature: if the quorum in the system is “lost” (i.e. at a given time no quorum component exists), a quorum can later become connected and still remain blocked.

The importance of this paper is in demonstrating, using a very simple protocol, a general technique for making distributed systems more resilient, specifically, always allowing a connected quorum in the system to make progress. The technique uses a linear order imposed on the quorums formed in the system, and guarantees that the decisions made by each quorum will be consistent with the decisions of the previous ones.

E3PC may be used in conjunction with quorum-based replication protocols, such as [6, 8, 9, 3, 4], in order to make the database always available to a quorum. The same Quorum System should be used to determine when the data is accessible to a group of processors as for the atomic commitment protocol. Thus, in order to complete a transaction, a group of processors needs to be a quorum of the total number of sites, and not just of the processors that invoked E3PC for the specific transaction. If the data is partially replicated, then for each item accessed by this transaction, a quorum of the sites it resides on is required.

There is a subtle point to consider with this solution: sites that did not take part in the basic E3PC for this transaction may take part in the recovery procedure. The local databases at such sites are not up-to-date, since they do not necessarily reflect the updates performed by the current transaction. Therefore, these processors need to recover the database state from the other processors during the merge, and before taking part in the recovery procedure. In the Virtual Partitions protocol [3, 4], this is done every time the view changes. In this case, we suggest to use the view change as a “fault detector” for E3PC; thus, the recovery procedure is always invoked following a view change, after all the participating processors have reached an up-to-date state.

The technique demonstrated here may be used to make other algorithms more resilient, e.g. an algorithm for maintaining a primary component in the network, to support processing of sequences of distributed transactions, as well as for replication [10]. The same principle may be combined with a dynamic voting scheme for replication (cf. Section 7 in [10]).

References


A Correctness Proof of E3PC

In this Section we prove the correctness of E3PC; we show that E3PC and its recovery procedure fulfill the requirements of atomic commitment (as defined in Chapter 7 of [1]) described in Section 3.1:

- In Theorem 1 below we will prove that all the processors that reach a decision reach the same one.
- In our protocol, a processor cannot reverse its decision after it has reached one. When a processor in a final state (COMMIT or ABORT) participates in some invocation of the recovery procedure, the decision in this invocation of the recovery procedure will correspond with its state.
- The COMMIT decision can only be reached if all processors voted Yes: In the basic E3PC, a committable decision can be made only if all the processors vote Yes. If the recovery procedure is invoked with no processor in a committable state, then according to the decision rule, a committable decision cannot be reached.
- If there are no failures and all processors voted Yes, then the decision will be to COMMIT.
- At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all processors will eventually reach a decision. Our protocol guarantees a much stronger property:

At any point in the execution of the protocol, if a group G of processors becomes connected, and this group contains a quorum\(^4\) of the processors, and no subsequent failures occur for sufficiently long, then all the members of G eventually reach a decision. Furthermore, every processor that can communicate with a processor that already reached a decision, will also, eventually, reach a decision.

This property is obvious from the decision rule.

We now prove that the decision made is unanimous, i.e. that if one processor decides to COMMIT, then no processor can decide to ABORT, and vice versa.

**Lemma 1** If a coordinator r sets its local value of Last Attempt to i and sends a PRE-COMMIT (PRE-ABORT) message to the slaves in Step 4 of the recovery procedure, then a Commit Quorum (Abort Quorum) of processors have set their value of Last Elected to i during the same invocation of the recovery procedure.

**Proof:** Immediate form the protocol, and from the fact that processors cannot “join” the recovery procedure in the middle, instead, it must be re-invoked to let them take part. □

**Lemma 2** At each processor, the value of Last Elected never decreases.

\(^4\)If different quorums are used for commit and abort, then we say the group contains a quorum if it contains both a Commit Quorum and an Abort Quorum.
Proof: The value of *Last_Elected* is modified only in Step 2 of the recovery procedure, when it is changed to *Max_Elected*+1. A processor may execute Step 2 only if it took part in the election of the coordinator in that invocation of the recovery procedure, and its value of *Last_Elected* was used to compute *Max_Elected*, and therefore *Max_Elected* > *Last_Elected*, and *Last_Elected* increases. □

Lemma 3 If two processors, *p* and *q* both set their *Last_Attempt* to the number *i* without changing to a final state, then either both of them set their *Last_Attempt* to *i* as a response to a PRE-COMMIT decision, or both of them set their *Last_Attempt* to *i* as a response to a PRE-ABORT decision.

Proof: A coordinator changes the value of *Last_Attempt* when it reaches a decision (in Step 4 of the recovery procedure, or in the basic E3PC), and it remains in a non-final state if the decision is PRE-COMMIT or PRE-ABORT. Other processors change the value of *Last_Attempt* only in response to a PRE-COMMIT or PRE-ABORT decision, in Step 5 of the recovery procedure, or in response to PRE-COMMIT in the basic E3PC.

Assume the contrary, then w.l.o.g., *p* set its *Last_Attempt* to *i* in response to a PRE-COMMIT decision in the course of some invocation, *I₀*, of the recovery procedure (or the basic E3PC), and *q*, in response to a PRE-ABORT decision, in an invocation *I₁*. From Lemma 1, a Commit Quorum of processors set their *Last_Elected* to *i* in invocation *I₀* and an Abort Quorum of processors set their *Last_Elected* to *i* in invocation *I₁*. Since the coordinator in invocation *I₀* decided to PRE-COMMIT and the coordinator in *I₁* decided to PRE-ABORT, *I₀* and *I₁* were different invocations of recovery procedure, or the basic E3PC.

Since every Commit Quorum intersects every Abort Quorum, there exists a processor, *s*, that set its *Last_Elected* to *i* in both invocations. W.l.o.g. *s* set its *Last_Elected* to *i* in *I₀* before setting it to *i* in *I₁*. From the protocol, a processor cannot concurrently take part in two invocations of the recovery procedure, furthermore, if a processor responds to messages from the coordinator in some invocation, it necessarily took part in the election of that coordinator. Therefore, *s* took part in the election of the coordinator in *I₁*, after it set its *Last_Elected* to *i*, and from Lemma 2, in the course of the election, the coordinator heard from *s* that its value of *Last_Elected* > *i* determined that *Max_Elected* > *i*. The new value of *Last_Elected* for this invocation was *Max_Elected*+1, which is greater than *i*, which contradicts our assumption. □

Lemma 4 At each processor, at any given time, *Last_Elected* > *Last_Attempt*.

Proof: From Lemma 2, the value of *Last_Elected* never decreases, so it is sufficient to show that *Last_Attempt* is never increased to exceed it. By induction on the steps of the protocol in which *Last_Attempt* changes: base: When E3PC is initiated, *Last_Elected* is set to one, and *Last_Attempt*, to zero. Step: Whenever *Last_Attempt* is changed in the course of the protocol, it takes the value of *Last_Elected*. □

Lemma 5 The value of *Last_Attempt* at each processor increases every time the processor changes state from a commitable state to a non-final non-commitable state, and vice versa. The value of *Last_Attempt* never decreases.

Proof: The only non-final commitable state is PRE-COMMIT, and the only way to switch to a PRE-COMMIT state is in response to a PRE-COMMIT decision, when setting *Last_Attempt* to *Last_Elected*. □
Likewise, the only way to switch from a committable state to a non-final non-committable state, is in Step 4 or in Step 5 of the recovery procedure, in response to a \textsc{Pre-Abort} decision, when setting \texttt{Last Attempt} to \texttt{Last Elected}.

It is sufficient to prove that \texttt{Last Attempt} increases when it is set to \texttt{Last Elected} in Step 4 or 5 of the recovery procedure, i.e. that \texttt{Last Attempt} < \texttt{Last Elected} before Step 4. And indeed, in Step 2, \texttt{Last Elected} is set to \texttt{Max Elected} + 1, which is greater than the value of \texttt{Last Elected} was when the recovery procedure was initialized, and from Lemma 4, \texttt{Last Elected} > \texttt{Last Attempt} at all times, therefore, before Step 4, \texttt{Last Elected} is greater than \texttt{Last Attempt}. □

\textbf{Lemma 6} \textit{If the coordinator reaches a \textsc{Commit} decision upon receiving a Commit Quorum of ACKs for \textsc{Pre-Commit} when setting its \texttt{Last Attempt} to \texttt{i}, then for every \texttt{j} \geq \texttt{i} no coordinator will decide \textsc{Pre-Abort} when setting its \texttt{Last Attempt} to \texttt{j}.}

\textbf{Proof:} By induction on \texttt{j}. \textit{Base (j = i):} Immediate from Lemma 3. \textit{Step:} We now assume that no coordinator decides \textsc{Pre-Abort} with \texttt{Last Attempt} = \texttt{k} for every \texttt{j} > \texttt{k} \geq \texttt{i}, and prove for \texttt{j}. From the assumption, no processor can be in a non-final non-committable state with its \texttt{j} > \texttt{Last Attempt} \geq \texttt{i}. Now, assume some coordinator \texttt{r} sets its \texttt{Last Attempt} to \texttt{j} in Step 4 of the recovery procedure, we have to show that \texttt{r} did not decide \textsc{Pre-Abort} during this invocation of the recovery procedure. Assume the contrary, then \texttt{r} collected states, from an \textsc{Abort Quorum} of processors with \texttt{Last Attempt} < \texttt{j}, and therefore, in this invocation \texttt{Max Attempt} < \texttt{j}. Since every \textsc{Commit Quorum} intersects every \textsc{Abort Quorum}, at least one member of \texttt{G}, \texttt{p}, took part in this invocation of the recovery procedure, and sent its state to \texttt{r}. Since \texttt{j} > \texttt{i}, from Lemma 5, \texttt{p} set its \texttt{Last Attempt} to \texttt{i} (and switched to a committable state) before this invocation. But, no processor can be in a non-final non-committable state with its \texttt{j} > \texttt{Last Attempt} \geq \texttt{i}, and therefore \texttt{Is Max Attempt Committable} is \textsc{True} in this invocation, which contradicts the assumption that \texttt{r} decides \textsc{Pre-Abort}. □

\textbf{Lemma 7} \textit{If the coordinator reaches a \textsc{Commit} decision when setting its \texttt{Last Attempt} to \texttt{i}, then for every \texttt{j} \geq \texttt{i} no coordinator will decide \textsc{Pre-Abort} when setting its \texttt{Last Attempt} to \texttt{j}.}

\textbf{Proof:} There are two cases to consider:

- If the coordinator reaches a \textsc{Commit} decision upon receiving a \textsc{Commit Quorum} of ACKs for \textsc{Pre-Commit} when setting its \texttt{Last Attempt} to \texttt{i}, then, from Lemma 6 for every \texttt{j} \geq \texttt{i} no coordinator will decide \textsc{Pre-Abort} when setting its \texttt{Last Attempt} to \texttt{j}.

- If the coordinator reaches a \textsc{Commit} decision during the recovery procedure upon receiving a \textsc{Commit} state, then some coordinator has reached a \textsc{Commit} decision before, when its \texttt{Last Attempt} was < \texttt{i}. We go back, by induction, to the first coordinator that reached a \textsc{Commit} decision. This coordinator must have reached a commit decision according to the previous case. Thus, we can conclude that for every \texttt{j} \geq \texttt{i} no coordinator will decide \textsc{Pre-Abort} when setting its \texttt{Last Attempt} to \texttt{j}.

□
Lemma 8 If the coordinator reaches an ABORT decision upon receiving an Abort Quorum of ACKs for PRE-ABORT when setting its Last_Attempt to i, then for every \(j \geq i\) no coordinator will decide PRE-COMMIT when setting its Last_Attempt to j.

Proof: This Lemma is dual to Lemma 6, and may be proven the same way. □

Lemma 9 If the coordinator reaches an ABORT decision when setting its Last_Attempt to i, then for every \(j \geq i\) no coordinator will decide PRE-COMMIT when setting its Last_Attempt to j.

Proof: There are three cases to consider:

- If the coordinator reaches an ABORT decision during the basic E3PC, this decision is reached because some processor voted No on the transaction. In this case, the coordinator does not PRE-COMMIT, and no processor reaches a commitable state in the course of the protocol. Note: If the recovery procedure is invoked with no processor in a commitable state, then according to the decision rule, a commitable decision cannot be reached.

- If the coordinator reaches an ABORT decision during the recovery procedure upon receiving an Abort Quorum of ACKs for PRE-ABORT when setting its Last_Attempt to i, then, from Lemma 8 for every \(j \geq i\) no coordinator will decide PRE-COMMIT when setting its Last_Attempt to j.

- If the coordinator reaches an ABORT decision during the recovery procedure upon receiving an ABORT state, then some coordinator has reached an ABORT decision before, when its Last_Attempt was < i. We go back, by induction, to the first coordinator that reached an ABORT decision, according to one of the previous two cases, and conclude that for every \(j \geq i\) no coordinator will decide PRE-COMMIT when setting its Last_Attempt to j.

□

Theorem 1 If some processor running the protocol COMMITs the transaction, then no other processor ABORTs the transaction, and vice versa.

Proof: A processor may COMMIT (ABORT) only upon hearing a COMMIT (ABORT) decision from its coordinator. Assume that a COMMIT or ABORT decision was reached for some transaction \(T\). Note: It is possible for more than one coordinator to reach a decision for the same transaction. Let \(i\) be the lowest value of Last_Attempt that a coordinator had when reaching a COMMIT or ABORT decision. There are two cases to consider:

1. Some coordinator reached an ABORT decision when setting its Last_Attempt to \(i\):
   Assume for the sake of contradiction, that some coordinator also reached a COMMIT decision, and let \(j\) be the lowest value of Last_Attempt of a coordinator reaching a COMMIT decision. From the assumption, \(j \geq i\). Furthermore, since \(j\) is the lowest value of Last_Attempt of a coordinator reaching a COMMIT decision, no processor could have started this invocation of the recovery procedure in the COMMITTED state, and the COMMIT decision must have been preceded by a PRE-COMMIT. But from Lemma 9 no coordinator can decide PRE-COMMIT when setting its Last_Attempt to \(j\), and we reach a contradiction.
2. Some coordinator reached a COMMIT decision when setting its `LastAttempt` to \( i \):

The proof is similar to the proof of Case 1 above, but there is one more case to consider: An \textsc{Abort} decision reached in the course of the basic 3PC (not in the recovery procedure), is \textit{not} preceded by a \textsc{Pre-Abort} decision. In this case, `LastAttempt` is set to 1, and the \textsc{Commit} decision couldn't have been reached with a lower value of `LastAttempt`, therefore \( i = 1 \). This case reduces to Case 1 proved above.

\( \square \)