Model Checking the Two-Phase Commit Protocol for Distributed Database Systems

Yisong Yu[1], Naipeng Dong[2], Jun Pang[3], and Jin Song Dong[1]

[1] School of Computing, National University of Singapore, Singapore
[2] School of EECS, The University of Queensland, Australia
[3] Department of Computer Science, University of Luxembourg, Luxembourg

Abstract. The reliability of distributed database management systems is essential in modern applications with inherently distributed architectures, such as cloud services and e-commerce platforms. The atomicity property of distributed transactions is crucial for data integrity and is typically implemented using the Two-Phase Commit (2PC) protocol. However, the correctness of 2PC in the presence of site failures, timeouts, and recovery scenarios remains complex and challenging to validate through traditional proofs alone. This work presents a formal model and verification of the 2PC protocol, including its recovery and termination subprotocols, using the CSP# language and the PAT model checker. Our approach abstracts certain system behaviors, such as communication reliability and log persistence, to improve verification efficiency while preserving correctness semantics. We explore the impact of bounded site failures and timeout occurrences on the protocol's verifiability, revealing a direct correlation between error frequency and state-space explosion.

Keywords: Model Checking · Distributed Database · 2-Phase Commit.

1 Introduction

The rapid digitalisation of the information age has driven computer applications such as web services and e-commerce platforms to adopt distributed architectures for large-scale data processing. This shift has spurred the development of distributed database management systems, which coordinate logically related databases across multiple sites. Compared to centralised systems, distributed databases offer greater reliability through replication, avoiding single points of failure and maintaining service despite site or link outages. To preserve consistency in such environments, user interactions are executed as distributed transactions—atomic units of work spanning multiple sites—whose correctness depends on ensuring atomicity.

Atomicity is achieved through two key primitives, namely *commit* and *abort*, managed by the recovery manager component (that handles both *transaction recovery* and *crash recovery*) of a distributed database management system. These operations ensure that transactions can either complete the remaining actions of a transaction by installing its updated pages into the stable storage,

or undo the already executed actions of a transaction by restoring the data in its updated pages in the underlying stable storage to their prior values upon recovery from any sort of failure occurred during the execution process of the given transaction. However, the correctness of the actual implementation of these primitives is contingent on distributed reliability protocols that dictate *when* to correctly perform the appropriate commit or abort operation to achieve the overall atomicity, especially in the presence of potential failures.

Our study focuses on the widely adopted atomic commitment protocol (e.g., by Google [4] and Microsoft [2]) - *Two-Phase Commit Protocol* and its companions *Recovery Protocol* and *Termination Protocol*, which collectively coordinate participating sites to reach a consistent decision on whether to *commit* or *abort* a given transaction. 2PC remains widely used in modern distributed systems. Google Spanner [4], a globally distributed SQL database, uses 2PC for transaction coordination. Microsoft uses 2PC in Fast Remote Memory [11], and Azure's Host Integration Server supports 2PC over the internet [2]. Open-source databases such as CockroachDB [3] and FoundationDB [7] also implement variants of 2PC to ensure fault tolerance. Recently, 2PC has been adapted for blockchain e.g., [12] and cross chain applications e.g., [10].

Several efforts have been made to formally verify 2PC. It has been specified using TLA+ to verify safety properties [6], and translated into Lean to verify consistency [5]. However, these models omit abort messages and site failures. The work [9] uses the mCRL2 process algebra to model 2PC, but does not account for timeout and recovery. Other studies such as [13] focus on performance analysis, while [8] used model checking to investigate cheating behaviour of participants rather than site failures. We model both the basic 2PC and its Termination and Recovery protocols using CSP# language supported by the PAT model checker [14], to formally verify their correctness and resilience.

2 Distributed Two-Phase Commit Protocol

We refer to the transaction manager at the transaction originating site as *coordinator*, and the transaction managers at other sites as *participants*.

2.1 Protocol Description

Basic 2PC Protocol The basic 2PC Protocol (Figure 1) has two phases:

Preparation (Voting) Phase. The coordinator initiates the commit process by sending a `Prepare` message to all participants. Participants decide to `Vote-Commit` or `Vote-Abort` based on their local state and respond to the coordinator.

Decision Phase. Based on the collected votes, the coordinator decides: Global commit: if all participants vote to commit; Global abort: if any participant votes to abort. Then, the coordinator communicates the decision to all participants. And participants acknowledge receipt of the decision and update their states.

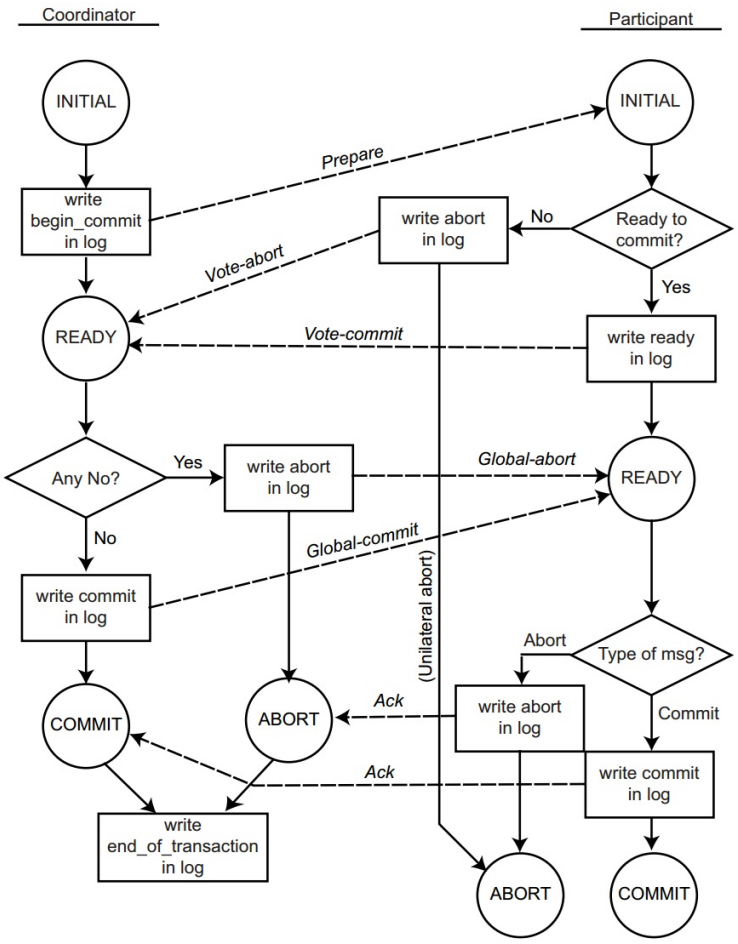

Fig. 1: Basic 2PC [15]. Circles indicate states and dash lines for messages.

Two-Phase Termination Protocol When failures occur, the Termination Protocol handles incomplete transactions to ensure consistency. Failures are detected by the employment of timeout mechanism. The following demonstrates a case-by-case failures occurring at different execution points shown in Figure 2.

Coordinator Timeouts. There are three states where the coordinator can time out, namely WAIT, COMMIT, and ABORT states. In the WAIT state, the coordinator times out while waiting for votes, and then decides to abort globally. In the COMMIT or ABORT state, the coordinator resends the corresponding global decision to unresponsive participants.

Participant Timeouts. There are two states where a participant can time out namely INITIAL and READY states. In the INITIAL state, a participant times out while waiting for a Prepare message, and then unilaterally aborts. In the

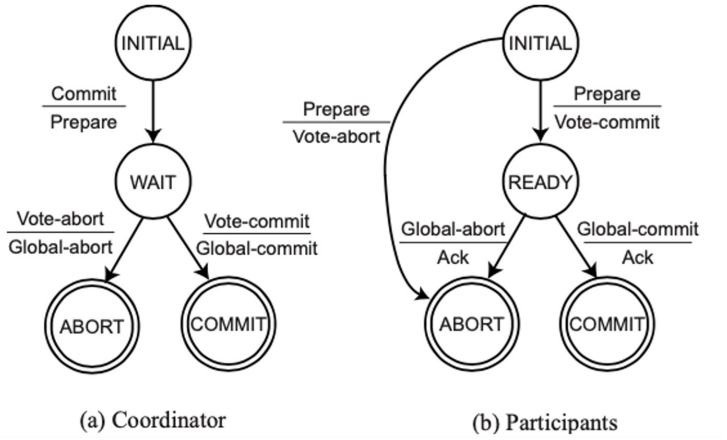

Fig. 2: Termination Based on State Transition [15]. Terminal states are represented by concentric circles. The event that triggers a state transition, by a received message, is at the top; the action taken as a result of a state transition, by a sent message, is at the bottom.

READY state, a participant times out while waiting for the coordinator's global decision, and then remains blocked.

Cooperative Termination Protocol. To reduce blocking, participants in the READY state may communicate directly with others. Participants in the INITIAL state unilaterally abort. Participants in ABORT or COMMIT states share their decision to aid others. Participants in the READY state remain uncertain if no conclusive replies are received. While this protocol mitigates blocking, it does not fully resolve it, especially in cases of coordinator failure.

Two-Phase Recovery Protocol The Recovery Protocol outlines steps for failed sites to rejoin the transaction.

Coordinator Failures: Restart and resend messages depending on last state.

Participant Failures: Resend votes or unilaterally abort based on the logged state. Recovery depends on communication with other sites to synchronize states and ensure consistency.

2.2 Claimed Properties of the Protocol

The 2PC Protocol aims to ensure the following critical properties [15] in the presence of failures, whose formal definitions are presented later in Section 4.
Agreement: All participants reach the same global decision.
Abort Preference: If any participant votes abort, the global decision is abort.
Commit Preference: If no failures occur and all participants vote to commit, the global decision is to commit.
Vote Alignment: A global commit implies all participants voted commit.

3 System Modelling

3.1 PAT Model Checker and CSP#

We use PAT model checker for verifying the properties of its capability in handling concurrent processes which is key to the modelling of distributed systems [14]. PAT accepts models in CSP# language, whose syntax is as follows:

Definition 1 (CSP# Syntax [14]).

$$P,\ Q\ ::= \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P \mid e\{prog\} \rightarrow P \mid c!exp \rightarrow P \mid c?x \rightarrow P \mid P[]Q \mid \mathit{if}(b)\{P\}\ \mathit{else}\ \{Q\} \mid \mathit{ifa}(b)\{P\}\ \mathit{else}\ \{Q\} \mid P;Q \mid P||Q \mid P|||Q \mid P\ \mathit{interrupt}\ Q \mid P\backslash A$$

A system is modelled as a process; and a process or subprocess is denoted as P or Q. *Stop* represents inaction, while *Skip* denotes successful termination. An event $e \rightarrow P$ performs e before continuing as P, and $e\{prog\} \rightarrow P$ additionally executes a program fragment *prog* atomically. Channel operations include sending an expression over channel c ($c!exp \rightarrow P$) and receiving over c and referring it as variable x ($c?x \rightarrow P$), namely sending a value of an expression *exp* to channel c or receiving a value from channel c into the variable x. Choices are expressed with external choice ($P[]Q$), conditional ($\mathit{if}(b)\{P\}\ \mathit{else}\ \{Q\}$), and atomic conditional ($\mathit{ifa}(b)\{P\}\ \mathit{else}\ \{Q\}$). Sequential composition is written as $P;Q$, parallel composition with synchronisation as $P||Q$, and interleaved (non-synchronised) parallelism as $P|||Q$. Process $P\ \mathit{interrupt}\ Q$ behaves as specified by P until the first visible event of Q. $P\backslash A$ turns events in A to invisible ones. More details of the modelling language can be found at the PAT User Manual [1].

3.2 Global Constant / Variable Definitions

A list of global constants, channels and variable definitions is shown in Figure 3, modelling the configurations and messages.

3.3 Coordinator Processes

The `Coordinator` process (Figure 4) models a normal execution of the 2PC Protocol starting from the `Coordinator_Send_Prepare` sub-process that can be interrupted by a random failure event, modelled by the `exception` event. In such cases, the control is handed over to `Coordinator_Error_Handling`, which then passes control to `Coordinator_Recover`. In the worst-case scenario, this recovery process may itself be repeatedly interrupted. To limit the number of allowable site failures, the shared exception event must eventually be disabled, modelled in the parallel `Coordinator_Error` process that is bounded by the `error_count`, where the synchronisation of the two parallel processes is modelled by shared alphabet. We detail each sub-processes in the remaining section.

Preparation. The execution of the coordinator (`Coordinator_Send_Prepare` in Figure 5) starts with a state transition from `C_INITIAL` to `C_WAIT`, followed by

```
#define PARTICIPANT_NUMBER 4;
#define COORDINATOR_BOUNDED_ERROR_COUNT 1; //-1 means unbounded
#define PARTICIPANT_BOUNDED_ERROR_COUNT 1; //-1 means unbounded
#define COORDINATOR_BOUNDED_TIMEOUT_COUNT 2;
enum {C_INITIAL, C_WAIT, C_ABORT, C_COMMIT, P_INITIAL, P_READY, P_ABORT, P_COMMIT,
      C_MSG_PREPARE, C_MSG_GLOBAL_COMMIT, C_MSG_GLOBAL_ABORT, P_MSG_VOTE_COMMIT,
      P_MSG_VOTE_ABORT, P_MSG_ACK, P_MSG_DECISION_REQUEST, P_MSG_DECISION_UNCERTAIN,
      P_MSG_DECISION_COMMIT, P_MSG_DECISION_ABORT, MSG_NULL};
channel network[PARTICIPANT_NUMBER] 5;
channel peer_network[PARTICIPANT_NUMBER * PARTICIPANT_NUMBER] 1;
var coordinator_state = C_INITIAL;
hvar coordinator_sent_messages = [MSG_NULL(PARTICIPANT_NUMBER)];
hvar coordinator_received_messages = [MSG_NULL(PARTICIPANT_NUMBER)];
hvar coordinator_received_messages_are_same = true;
hvar coordinator_timeout = false;
var participants_state = [P_INITIAL(PARTICIPANT_NUMBER)];
hvar participants_received_message = [MSG_NULL(PARTICIPANT_NUMBER)];
hvar participants_sent_message = [MSG_NULL(PARTICIPANT_NUMBER)];
hvar participants_decision_request_responses[PARTICIPANT_NUMBER][PARTICIPANT_NUMBER];
hvar participants_decision_request_result=[P_MSG_DECISION_UNCERTAIN(PARTICIPANT_NUMBER)];
```

Fig. 3: Global Constants and Variables

```
Coordinator() = Coordinator_Error(COORDINATOR_BOUNDED_ERROR_COUNT) ||
  (Coordinator_Send_Prepare(COORDINATOR_BOUNDED_TIMEOUT_COUNT)
   interrupt exception -> Coordinator_Error_Handling(COORDINATOR_BOUNDED_TIMEOUT_COUNT));
Coordinator_Error_Handling(timeout_count) =  Coordinator_Recover(timeout_count) interrupt
    exception -> Coordinator_Error_Handling(timeout_count);
Coordinator_Error(error_count) =
 [error_count != 0](exception->Coordinator_Error(error_count-1)) [] [error_count == 0]Skip;
```

Fig. 4: The Coordinator Process

```
Coordinator_Send_Prepare(timeout_count) = c_prepare{coordinator_state = C_WAIT;} ->
 (||| i: {0..PARTICIPANT_NUMBER-1} @ ((ifa (coordinator_sent_messages[i] == MSG_NULL) {
   atomic{network[i]!C_MSG_PREPARE ->
    c_prepare_sent_to.i{coordinator_sent_messages[i] = C_MSG_PREPARE;} -> Skip}} else {
c_prepare_already_sent_to.i->Skip})\{c_prepare_sent_to.i,c_prepare_already_sent_to.i}));
Coordinator_Collect_Votes(timeout_count);
```

Fig. 5: Coordinator - Preparation

concurrently sending a `C_MSG_PREPARE` message in an interleaved manner to each participant `i` through the corresponding channel `network[i]` if it hasn't already done so and recording this message sent in `coordinator_sent_messages[i]`.

We model the coordinator to record sent messages for handling site failures that may occur mid-process, where the `C_MSG_PREPARE` message only reaches some participants but not all. This will trigger recovery which passes the control flow back to the start of this process, causing the same message to be resent and buffered twice for some participants, complicating participant-side handling.

Vote Collection. Once the coordinator sends `C_MSG_PREPARE` to all participants, `Coordinator_Collect_Vote` process (Figure 6) is invoked to collect the vote from each participant. This process listens on channel `network[i]` to receive `P_MSG_VOTE_COMMIT` or `P_MSG_VOTE_ABORT` if local decision hasn't been received, modelled in `Coordinator_Collect_Votes_Auxiliary`. The received messages are stored in `coordinator_received_message[i]` for decision-making.

The `Coordinator_Collect_Votes` process simulates bounded timeout errors using the `interrupt` construct, when the coordinator is in the `C_WAIT` state. If

```
Coordinator_Collect_Votes_Auxiliary(timeout_count) =
 ||| i: {0..PARTICIPANT_NUMBER-1} @ ((
  if (coordinator_received_messages[i] == MSG_NULL) {
   atomic{network[i]?P_MSG_VOTE_COMMIT ->
    c_vote_commit_collected_from.i{coordinator_received_messages[i] = P_MSG_VOTE_COMMIT;}
    -> Skip []
    network[i]?P_MSG_VOTE_ABORT ->
    c_vote_abort_collected_from.i{coordinator_received_messages[i] = P_MSG_VOTE_ABORT;}
    -> Skip}
  } else {c_vote_already_collected_from.i -> Skip
  }) \ {c_vote_commit_collected_from.i, c_vote_abort_collected_from.i,
        c_vote_already_collected_from.i});
Coordinator_Collect_Votes(timeout_count) =
 ([timeout_count != 0](Coordinator_Collect_Votes_Auxiliary(timeout_count) interrupt
    Coordinator_Terminate(timeout_count - 1)) []
  [timeout_count == 0]Coordinator_Collect_Votes_Auxiliary(timeout_count));
 Coordinator_Make_Global_Decision(timeout_count);
```

Fig. 6: Coordinator - Vote Collection

the initial value passed to the `Coordinator_Collect_Votes` process is greater than 0, it is decremented by 1 each time a timeout occurs. Eventually, it reaches 0, at which point the process can no longer time out and will proceed to decision making. This mechanism effectively bounds the number of allowed timeouts at the coordinator site. The same approach is applied in all subsequent process definitions where coordinator timeouts need to be limited.

Time Modelling. Abstraction is applied on the process level to abstract away the timing details, namely we use the non-timed interrupt construct instead of the timed construct e.g., `interrupt[t]`, as their semantics are considered to be the same in this case. In the matter of result, both of them simulate exactly the same timeout behaviour in this scenario, but the later one using timed `interrupt[t]` would inevitably bloat the state space due to the extra global clock process being concurrently executed with other user-defined processes, resulting in more execution traces being generated with potentially more states that need to be searched for during the verification process. Similarly, the current implementation is considered to be the most efficient with redundant details eliminated as much as possible to simplify the model, and the same idea applies to all subsequent process definitions where timeout errors occur.

Global Decision. As soon as all the local decision messages are received and recorded by the coordinator, the process `Coordinator_Make_Global_Decision` (Figure 7) is invoked to discover whether there's any participant voting to abort by looping through all recorded votes. If at least one negative vote if found, rendering the flag `coordinator_received_messages_are_same` to be false, a global abort signal is sent in the process `Coordinator_Send_Global_Abort`, otherwise `Coordinator_Send_Global_Commit` is invoked to signal a global commit.

The `Coordinator_Send_Global_Commit` process starts with a state transition from `C_WAIT` to `C_COMMIT`, followed by sending a `C_MSG_GLOBAL_COMMIT` message concurrently to each participant `i` through the corresponding channel `network[i]` and recording this message sent in `coordinator_sent_messages[i]`.

The `Coordinator_Send_Global_Abort` process starts with a state transition from `C_WAIT` to `C_ABORT`, followed by concurrently sending a `C_MSG_GLOBAL_ABORT`

```
Coordinator_Make_Global_Decision(timeout_count) =
 ifa (!coordinator_timeout) {tau{coordinator_received_messages_are_same = true;
   var index = 0; while (index < PARTICIPANT_NUMBER) {
    if (coordinator_received_messages[index] == P_MSG_VOTE_ABORT) {
     coordinator_received_messages_are_same = false;} index = index + 1;}} ->
  ifa (coordinator_received_messages_are_same) {
   Coordinator_Send_Global_Commit(timeout_count)
  } else {Coordinator_Send_Global_Abort(timeout_count)}};
#alphabet Coordinator_Send_Global_Commit {
 i: {0..(PARTICIPANT_NUMBER-1)} @ c_global_commit_already_sent_to.i};
Coordinator_Send_Global_Commit(timeout_count) =
 c_global_commit{coordinator_state = C_COMMIT;} -> (||| i: {0..PARTICIPANT_NUMBER-1} @ (
  ifa (coordinator_sent_messages[i] == C_MSG_PREPARE) {
   atomic{network[i]!C_MSG_GLOBAL_COMMIT ->
 c_global_commit_sent_to.i{coordinator_sent_messages[i] = C_MSG_GLOBAL_COMMIT;} -> Skip}
  } else {c_global_commit_already_sent_to.i -> Skip }
 ) \ {c_global_commit_sent_to.i, c_global_commit_already_sent_to.i});
 Coordinator_Receive_ACK(timeout_count);
#alphabet Coordinator_Send_Global_Abort {
 i: {0..(PARTICIPANT_NUMBER-1)} @ c_global_abort_already_sent_to.i,
 i: {0..(PARTICIPANT_NUMBER-1)} @ c_global_abort_no_need_to_send_to.i};
Coordinator_Send_Global_Abort(timeout_count)=c_global_abort{coordinator_state = C_ABORT;}->
 (||| i: {0..PARTICIPANT_NUMBER-1} @ (ifa (coordinator_sent_messages[i] == C_MSG_PREPARE &&
coordinator_received_messages[i]!=P_MSG_VOTE_ABORT){atomic{network[i]!C_MSG_GLOBAL_ABORT ->
    c_global_abort_sent_to.i{coordinator_sent_messages[i] = C_MSG_GLOBAL_ABORT;} -> Skip}
  } else if (coordinator_sent_messages[i] == C_MSG_GLOBAL_ABORT) {
            c_global_abort_already_sent_to.i -> Skip
  } else {c_global_abort_no_need_to_send_to.i -> Skip
  }) \ {c_global_abort_sent_to.i, c_global_abort_already_sent_to.i,
        c_global_abort_no_need_to_send_to.i}); Coordinator_Receive_ACK(timeout_count);
```

Fig. 7: Coordinator - Global Decision

message to each participant `i` voting to commit through the corresponding channel `network[i]` and recording this message in `coordinator_sent_messages[i]`.

Ack Collection. No matter the global decision is to commit or abort, the process moves to the execution of `Coordinator_Receive_ACK` (Figure 8). The process collects an acknowledgement from each participant. Similarly, listening to channel `network[i]` for the `P_MSG_ACK` message is modelled in an auxiliary process `Coordinator_Receive_ACK_Auxiliary`, and the received message is recorded in `coordinator_received_messages[i]` for each participant `i`. The main process `Coordinator_Receive_ACK` simulates the bounded timeout errors of the coordinator in `C_ABORT` or `C_COMMIT` state, in a similar manner as above.

Recovery. The `Coordinator_Recover` process is invoked whenever a site failure occurs to help the coordinator to recover its state. The modelling is straightforward following the protocol specification (Figure 9). Note that since duplicate messages sent to the same participant multiple times will be strictly prohibited by each individual process defined previously, we do not need to guard against these cases again. And considering that the coordinator will always resume in `C_INITIAL` state no matter it fails in `C_INITIAL` or `C_WAIT`, these two scenarios are consolidated into one for simplicity, collectively handled by the `Coordinator_Send_Prepare` process in the else block.

Termination. The `Coordinator_Terminate` process is invoked when a site failure is detected to help the coordinator to gracefully terminate the current active transaction. Similar to how the recovery protocol is implemented, time-

```
Coordinator_Receive_ACK_Auxiliary() =
 ||| i: {0..PARTICIPANT_NUMBER-1} @ ((
  ifa (coordinator_received_messages[i] == P_MSG_VOTE_ABORT ||
       coordinator_received_messages[i] == P_MSG_VOTE_COMMIT) {
   atomic{network[i]?P_MSG_ACK ->
          c_ack_received_from.i{coordinator_received_messages[i] = P_MSG_ACK;} -> Skip}
  } else {c_ack_no_need_to_receive_from.i -> Skip
  } \ {c_ack_received_from.i, c_ack_no_need_to_receive_from.i}) );
Coordinator_Receive_ACK(timeout_count) =
 [timeout_count != 0](Coordinator_Receive_ACK_Auxiliary()
    interrupt Coordinator_Terminate(timeout_count - 1)) []
 [timeout_count == 0]Coordinator_Receive_ACK_Auxiliary();
```

Fig. 8: Coordinator - Ack Collection

```
Coordinator_Recover(timeout_count) =
 ifa (coordinator_state == C_COMMIT) {Coordinator_Send_Global_Commit(timeout_count)
 } else if (coordinator_state == C_ABORT) {Coordinator_Send_Global_Abort(timeout_count)
 } else {Coordinator_Send_Prepare(timeout_count) };
```

Fig. 9: Coordinator - Recovery

```
Coordinator_Terminate(timeout_count) = tau{coordinator_timeout = true;} ->
 ifa (coordinator_state == C_COMMIT) { Coordinator_Send_Global_Commit(timeout_count)
 } else if (coordinator_state == C_ABORT) { Coordinator_Send_Global_Abort(timeout_count)
 } else { c_global_abort{coordinator_state = C_ABORT;} ->
  (||| i: {0..PARTICIPANT_NUMBER-1} @ (atomic {if (!call(cempty, network[i])) {
     network[i]?message -> case {
message == P_MSG_VOTE_ABORT:tau{coordinator_received_messages[i] = P_MSG_VOTE_ABORT} -> Skip
message == P_MSG_VOTE_COMMIT:tau{coordinator_received_messages[i] = P_MSG_VOTE_COMMIT}->Skip
default: Skip //default covers the case where message == C_MSG_PREPARE}};
    network[i]!C_MSG_GLOBAL_ABORT ->
    c_global_abort_sent_to.i{coordinator_sent_messages[i] = C_MSG_GLOBAL_ABORT;} ->
    network[i]?P_MSG_ACK ->
    c_ack_received_from.i{coordinator_received_messages[i] = P_MSG_ACK;} -> Skip } )) };
```

Fig. 10: Coordinator - Termination

out in `C_COMMIT` or `C_ABORT` state is handled in an elegant manner by delegating the control back to `Coordinator_Send_Global_Commit` process and `Coordinator_Send_Global_Abort` process respectively.

However, timeout in the `C_WAIT` state must be handled as a special case instead of simply passing the control back to `Coordinator_Send_Global_Abort`. If there's a message sitting in the buffer of `network[i]` for a participant `i`, message must be one of the following: `P_MSG_VOTE_ABORT` or `P_MSG_VOTE_COMMIT` if that participant `i` has already sent its decision to the coordinator and transitioned from `P_INITIAL` to `P_ABORT` or `P_READY`; `C_MSG_PREPARE` if participant `i` remains in `P_INITIAL` and hasn't processed it yet. In all cases, the coordinator should consume (i.e. by reading) the obsolete message to clear the channel buffer. Recall that `network[i]` supports bidirectional communication, any unprocessed message in the channel can block subsequent communication, stalling the entire system. To prevent this, the coordinator must clear the channel buffer before sending the global abort message (`C_MSG_GLOBAL_ABORT`) to the `network[i]`, and then wait for an acknowledgement(`P_MSG_ACK`). Because each `network[i]` is independent, the above actions are encapsulated in a separate sub-process per participant, interleaved within the `Coordinator_Terminate` process.

```
Participant_Receive_Prepare(i) = ((atomic{network[i]?C_MSG_PREPARE->p_prepare_received_by.i{
      participants_received_message[i] = C_MSG_PREPARE;} -> Skip
 } interrupt Participant_Terminate(i)) \ {p_prepare_received_by.i});
 (Participant_Send_Vote_Commit(i) [] Participant_Send_Vote_Abort(i));

Participant_Send_Vote_Abort(i)=(ifa(participants_received_message[i]==C_MSG_GLOBAL_ABORT &&
    participants_sent_message[i]==P_MSG_ACK){p_global_abort_prematurely_received_by.i->Skip
 } else { atomic{network[i]!P_MSG_VOTE_ABORT -> p_vote_abort_sent_by.i{
    participants_sent_message[i] = P_MSG_VOTE_ABORT; participants_state[i] = P_ABORT;
   } -> Skip } }) \ {p_global_abort_prematurely_received_by.i};

Participant_Send_Vote_Commit(i)=(ifa(participants_received_message[i]==C_MSG_GLOBAL_ABORT &&
    participants_sent_message[i]==P_MSG_ACK){p_global_abort_prematurely_received_by.i->Skip
 } else if (participants_sent_message[i] == MSG_NULL) {
  atomic{ network[i]!P_MSG_VOTE_COMMIT -> p_vote_commit_sent_by.i{
    participants_sent_message[i] = P_MSG_VOTE_COMMIT; participants_state[i] = P_READY;
   } -> Skip }; Participant_Collect_Decision(i)
 } else { p_vote_commit_already_sent_by.i -> Participant_Collect_Decision(i)
 }) \ {p_global_abort_prematurely_received_by.i, p_vote_commit_already_sent_by.i};

Participant_Collect_Decision(i) = ((ifa(participants_received_message[i] == C_MSG_PREPARE){
  atomic{network[i]?C_MSG_GLOBAL_COMMIT -> p_global_commit_received_by.i{
    participants_received_message[i] = C_MSG_GLOBAL_COMMIT; } -> Skip  []
   network[i]?C_MSG_GLOBAL_ABORT -> p_global_abort_received_by.i{
    participants_received_message[i] = C_MSG_GLOBAL_ABORT;}->Skip}; Participant_Send_ACK(i)
 } else { p_global_decision_already_received_by.i -> Participant_Send_ACK(i)
 }) \ {p_global_commit_received_by.i, p_global_abort_received_by.i,
    p_global_decision_already_received_by.i}) interrupt Participant_Terminate(i);

Participant_Send_ACK(i) = (ifa (participants_received_message[i] == C_MSG_GLOBAL_COMMIT &&
    participants_sent_message[i] != P_MSG_ACK) {
  atomic{network[i]!P_MSG_ACK->p_ack_commit_sent_by.i{participants_state[i]=P_COMMIT;}->Skip}
 } else if (participants_received_message[i] == C_MSG_GLOBAL_ABORT &&
    participants_sent_message[i] != P_MSG_ACK) {
  atomic{ network[i]!P_MSG_ACK->p_ack_abort_sent_by.i{participants_state[i]=P_ABORT;}->Skip}
 } else { p_ack_already_sent_by.i -> Skip }) \ {p_ack_already_sent_by.i};
```

Fig. 11: Participant `i` - Basic Execution

3.4 Participant Processes

A participant `i` begins with the `Participant_Receive_Prepare(i)` process (Figure 11), where it listens on `network[i]` for the `C_MSG_PREPARE` message from the coordinator, records it in `participants_received_message[i]`, and decides either to vote commit (invoking `Participant_Send_Vote_Commit(i)`) or abort (invoking `Participant_Send_Vote_Abort(i)`)). The timeout of a participant `i` is simulated using the `interrupt` construct, which triggers a transition to `Participant_Terminate(i)` modelling the Cooperative Termination.

If voting to abort, `Participant_Send_Vote_Abort(i)` sends the message `P_MSG_VOTE_ABORT`, records it in `participants_sent_message[i]`, and transitions from `P_INITIAL` to `P_ABORT`. If a premature `C_MSG_GLOBAL_ABORT` is received and acknowledged due to the coordinator timeout, this will be separately handled by `Participant_Daemon(i)` (Figure 14). Voting to commit is handled in a similar manner. The only difference is that if the commit vote has already been sent before a failure, no further action is required during recovery.

The participant then awaits the global decision in `Participant_Collect_Decision(i)`, listening for `C_MSG_GLOBAL_COMMIT` or `C_MSG_GLOBAL_ABORT`. Upon receiving the message, it records it and, if still in `P_READY`, a timeout may

```
Participant_Recover(i) = ifa (participants_state[i] == P_INITIAL) {
  p_unilaterally_abort_by.i{participants_state[i] = P_ABORT;} -> Skip
 } else if (participants_state[i] == P_READY) {Participant_Send_Vote_Commit(i)} else {Skip};

Participant_Terminate(i) =  ifa (participants_state[i] == P_INITIAL) {
  p_unilaterally_abort_by.i{participants_state[i] = P_ABORT;} -> Skip
 } else { Participant_Cooperative_Terminate(i) };

Participant_Cooperative_Terminate(i) =
 (||| j: {0..PARTICIPANT_NUMBER-1} @ Participant_Cooperative_Terminate_Thread(i, j));
 tau{participants_decision_request_result[i] = P_MSG_DECISION_UNCERTAIN;
  var j = 0; while (j < PARTICIPANT_NUMBER) {
   if (participants_decision_request_responses[i][j] == P_MSG_DECISION_ABORT) {
    participants_decision_request_result[i] = P_MSG_DECISION_ABORT;
   } else if (participants_decision_request_responses[i][j] == P_MSG_DECISION_COMMIT) {
    participants_decision_request_result[i] = P_MSG_DECISION_COMMIT;
   } j = j + 1; } } -> case {
  participants_decision_request_result[i] == P_MSG_DECISION_ABORT:
        p_cooperatively_abort_by.i{participants_state[i] = P_ABORT} -> Skip
  participants_decision_request_result[i] == P_MSG_DECISION_COMMIT:
        p_cooperatively_abort_by.i{participants_state[i] = P_COMMIT} -> Skip
  default: [participants_state[i] == P_COMMIT || participants_state[i] == P_ABORT] Skip };

Participant_Cooperative_Terminate_Thread(i, j) =
 atomic{ peer_network[i * PARTICIPANT_NUMBER + j]!P_MSG_DECISION_REQUEST ->
  ( peer_network[i * PARTICIPANT_NUMBER + j]?P_MSG_DECISION_ABORT ->
   p_decision_abort_received_from_by.j.i{
    participants_decision_request_responses[i][j] = P_MSG_DECISION_ABORT; } -> Skip []
   peer_network[i * PARTICIPANT_NUMBER + j]?P_MSG_DECISION_COMMIT ->
   p_decision_commit_received_from_by.j.i{
    participants_decision_request_responses[i][j] = P_MSG_DECISION_COMMIT; } -> Skip []
   peer_network[i * PARTICIPANT_NUMBER + j]?P_MSG_DECISION_UNCERTAIN ->
   p_decision_uncertain_received_from_by.j.i{
    participants_decision_request_responses[i][j] = P_MSG_DECISION_UNCERTAIN; } -> Skip ) };

#alphabet Participant_Error {exception.i};
Participant_Error(i, error_count) =
[error_count!=0](exception.i->Participant_Error(i, error_count-1)) [] [error_count==0]Skip;

Participant_Error_Handling(i) =
  Participant_Recover(i) interrupt exception.i ->Participant_Error_Handling(i);
```

Fig. 12: Participant `i` - Recovery, Termination and Error Handling

again trigger `Participant_Terminate(i)`. Once the global decision is received, `Participant_Send_ACK(i)` sends a `P_MSG_ACK` and transitions to `P_COMMIT` or `P_ABORT` as appropriate.

Site failures invoke `Participant_Recover(i)` to recover. If failure occurs in `P_INITIAL`, the participant unilaterally aborts. If in `P_READY`, it can resume from `Participant_Send_Vote_Commit(i)`. `Participant_Terminate(i)` models a graceful termination. Timeout in `P_INITIAL` results in an abort, and timeout in `P_READY` invokes `Participant_Cooperative_Terminate(i)` for recovery.

A special feature for participants is that cooperative termination may be invoked by a participant `i` when the coordinator is unavailable, modelled in `Participant_Cooperative_Terminate(i,j)`. It spawns a thread for each participant `j` executed as `Participant_Cooperative_Terminate_Thread(i,j)`. Each thread sends a `P_MSG_DECISION_REQUEST` for participant `j` over the channel `peer_network[i * PARTICIPANT_NUMBER + j]`, waits for a response, and records them in `participants_decision_request_responses[i][j]`. Once all

threads complete, participant i scans the collected responses. If any participant replies with a P_MSG_DECISION_ABORT or P_MSG_DECISION_COMMIT, i transitions from P_READY to P_ABORT or P_COMMIT accordingly. If all responses are P_MSG_DECISION_UNCERTAIN, i must wait for the coordinator's recovery and the eventual global decision broadcast.

At the end of Participant_Cooperative_Terminate(i), a guarded Skip is used to ensure correct semantics, preventing termination unless i reaches a terminal state, namely P_ABORT or P_COMMIT. Note that global decision receipt is handled separately by Participant_Daemon(i).

The Participant_Error(i,error_count) process models bounded site failures using the exception.i event. Each invocation decrements error_count, limiting how many failures can occur. Once error_count reaches 0, no further failures are simulated for participant i. To avoid runtime exception during verification, the process alphabet must be explicitly specified due to its recursive nature with decreasing error_count. Participant_Error_Handling(i) wraps Participant_Recover(i) to allow site failure during recovery. This design enables thorough verification by exposing edge cases.

The Overall Participant Process. With all necessary primitives defined, we can now assemble the complete participant(i) process. This process models two concurrent components: the Participant_Error process that models bounded random site failures, and the normal execution of the 2PC Protocol starting from the Participant_Receive_Prepare(i), including potential interruptions.

```
Participant(i) = Participant_Error(i, PARTICIPANT_BOUNDED_ERROR_COUNT) ||
  Participant_Receive_Prepare(i) interrupt exception.i -> Participant_Error_Handling(i);

Participants() = ||| i: {0..PARTICIPANT_NUMBER-1} @ Participant(i);
```

Fig. 13: Overall Participants Process

A natural question may arise: why are timeout events not similarly bounded for a participant i. The reason lies in the structure of Participant_Terminate(i). Unlike Coordinator_Terminate, which may loop back to earlier stages and thus tolerate multiple timeouts, Participant_Terminate(i) does not return control to Participant_Send_Vote_Abort(i) or Participant_Send_Vote_Commit(i). Therefore, a timeout error can occur at most once, eliminating the need for explicit bounding in the model. The Participants() process models the behaviour of all participants in the system by interleaving the execution of all Participant(i) processes, where i ranges from 0 to PARTICIPANT_NUMBER-1.

3.5 Participants Daemon Processes

As previously mentioned, the Participants_Daemon operates independently in the background, separated from each Participant(i) process. It spawns multiple threads—modelled as Participant_Daemon_Thread(i, j)—equal to the number of participants. Each thread performs two key tasks:

```
Participants_Daemon() = ||| i: {0..PARTICIPANT_NUMBER-1} @ Participant_Daemon(i);

Participant_Daemon(i) =
 (||| j: {0..PARTICIPANT_NUMBER-1} @ Participant_Daemon_Thread(i, j)) |||
 atomic{network[i]?C_MSG_GLOBAL_COMMIT -> p_daemon_global_commit_received_by.i{
   participants_received_message[i] = C_MSG_GLOBAL_COMMIT;} -> network[i]!P_MSG_ACK ->
  p_daemon_ack_commit_sent_by.i{participants_sent_message[i] = P_MSG_ACK;
   participants_state[i] = P_COMMIT;} -> Skip []
  network[i]?C_MSG_GLOBAL_ABORT -> p_daemon_global_abort_received_by.i{
    participants_received_message[i] = C_MSG_GLOBAL_ABORT;} -> network[i]!P_MSG_ACK ->
  p_daemon_ack_abort_sent_by.i{participants_sent_message[i] = P_MSG_ACK;
   participants_state[i] = P_ABORT;} -> Skip };

Participant_Daemon_Thread(i, j) =
 peer_network[j * PARTICIPANT_NUMBER + i]?P_MSG_DECISION_REQUEST -> case {
  participants_state[i] == P_INITIAL:
peer_network[j*PARTICIPANT_NUMBER + i]!P_MSG_DECISION_ABORT->Participant_Daemon_Thread(i,j)
  participants_state[i] == P_READY:
peer_network[j*PARTICIPANT_NUMBER+i]!P_MSG_DECISION_UNCERTAIN->Participant_Daemon_Thread(i,j)
  participants_state[i] == P_COMMIT:
peer_network[j*PARTICIPANT_NUMBER + i]!P_MSG_DECISION_COMMIT->Participant_Daemon_Thread(i,j)
  participants_state[i] == P_ABORT:
peer_network[j* PARTICIPANT_NUMBER + i]!P_MSG_DECISION_ABORT->Participant_Daemon_Thread(i,j)
  default: Participant_Daemon_Thread(i, j)};
```

Fig. 14: Participant Daemon Process

- Handling decision requests: Each thread listens on the channel `peer_network[j * PARTICIPANT_NUMBER + i]` for a `P_MSG_DECISION_REQUEST` message from participant `j`. Upon receipt, it replies with: `P_MSG_DECISION_ABORT` if in `P_INITIAL` or `P_ABORT` state, `P_MSG_DECISION_COMMIT` if in `P_COMMIT` state, or `P_MSG_DECISION_UNCERTAIN` if in `P_READY` state.
- Processing the global decision from the coordinator: Each thread also listens on `network[i]` for a `C_MSG_GLOBAL_COMMIT` message or `C_MSG_GLOBAL_ABORT` message. Upon receipt, it records the message in `participants_received_message[i]`, sends a `P_MSG_ACK` acknowledge the decision, records the ACK in `participants_sent_message[i]`, and transitions to `P_COMMIT` or `P_ABORT` based on the message received.

`Participants_Daemon()` interleaves the execution of `Participant_Daemon(i)` processes (for `i` = 0 to `PARTICIPANT_NUMBER-1`), modelling the collective background behaviour of all participants.

3.6 The Overall System Process

The full system, i,e, the `Distributed_System()` process, is an interleaved execution of `Coordinator()`, `Participants()`, and `Participants_Daemon()`[4].

```
Distributed_System() = Coordinator() ||| Participants() ||| Participants_Daemon();
```

Fig. 15: The Overall System

It abstracts communication link failures, avoiding the complexity of simulating unreliable networks. Log writes are also abstracted as atomic state transitions to reduce interleaving and improve verification efficiency. To keep the state space

[4] Available at `https://github.com/YuuYouYoo/CS5219-CS5232-Project`.

```
//Agreement 1
#assert Distributed_System() |= [] <> (
 four_participant_states_are_commit || four_participant_states_are_abort );
//Agreement 2
#assert Distributed_System() |= [] (
 coordinator_state_is_commit -> <> four_participant_states_are_commit );
//Agreement 3
#assert Distributed_System() |= [] (
 coordinator_state_is_abort -> <> four_participant_states_are_abort );
//Abort Preference
#assert Distributed_System() |= [] ( participant_0_state_is_abort ||
 participant_1_state_is_abort || participant_2_state_is_abort ||
 participant_3_state_is_abort -> <> four_participant_states_are_abort);
//Commit Preference
#assert Distributed_System() |= [] (p_vote_commit_sent_by.0 &&
 p_vote_commit_sent_by.1 && p_vote_commit_sent_by.2 &&
 p_vote_commit_sent_by.3 -> <> coordinator_state_is_commit );
//Vote Alignement
#assert Distributed_System() |= [] (coordinator_state_is_commit
 -> (p_vote_commit_sent_by.0 && p_vote_commit_sent_by.1 &&
  p_vote_commit_sent_by.2 && p_vote_commit_sent_by.3 ));
```

Fig. 16: Formulation of Properties (Snippet of Four Participants Example)

finite and tractable, site failures and timeouts are bounded. Communication link failures may cause delay or message loss. As this model does not implicitly model time for efficiency, latency before timeout is equivalent to message received, and latency after timeout is the same as message loss, as the message being sent will not be persisted but instead dropped. Message loss is captured by no further message sending or receiving after timeout.

4 Investigated Properties and Verification Results

We define a set of assertions to formulate the important properties claimed in the 2PC Protocol, shown in Figure 16 (four participants as an example).

Agreement. This property is formulated by the following three sub-properties

1. *Agreement 1:* All participants will eventually reach the same global decision—either all commit or all abort.
2. *Agreement 2:* If the coordinator reaches the COMMIT state, then all participants must eventually reach COMMIT.
3. *Agreement 3:* If the coordinator reaches the ABORT state, then all participants must eventually reach ABORT.

Abort Preference. If at least one participant reaches the ABORT state, then all participants must eventually reach ABORT.

Commit preference. This property includes a "no failures" condition. However, in our model, sites are assumed to always recover from failures; therefore, this constraint is omitted for clarity. The property is formulated as: If all participants vote to commit a transaction, then the coordinator must eventually decide to globally commit it.

Vote alignment. If the coordinator decides to globally commit the protocol, then it must be the case that all participants have voted to commit the transaction.

Property	#Participant	Result	States	Transitions	Time(s)	Memory(KB)
Agreement 1	2	Valid	145218	409901	3.95	105705
	3	Valid	23629458	98961221	1099	15355173
	4	Time Out	39177688	235443330	-	33458016
Agreement 2	2	Valid	124093	314115	3.11	904484
	3	Valid	18638427	64918577	817	10557383
	4	Time Out	43363694	211240199	-	32599380
Agreement 3	2	Valid	140383	388727	3.67	102038
	3	Valid	23355117	97301121	1066	15022524
	4	Time Out	35760192	213465259	-	30358774
Abort Preference	2	Valid	142495	401165	3.93	105368
	3	Valid	23597859	98722316	1116	15173606
	4	Time Out	38179569	228487994	-	32276466
Commit Preference	2	Valid	124093	314115	3.07	91678
	3	Valid	18638427	64918577	793	10895478
	4	Time Out	43571006	21281683	-	32545493
Vote Alignment	2	Valid	124093	314115	2.68	12330
	3	Valid	18638427	64918577	689	824183
	4	OutOfMemory	-	-	-	-

Table 1: Verification Results

Verification Results. We verified these properties of models with two, three and four participants using PAT 3.5.1 with Intel Core i7 11800H / 32GB RAM. The result is shown in Table 1, where "-" indicates that no value can be provided. When PAT times out, the values in the *States*, *Transitions*, and *Memory* columns reflect the status at the termination of verification. These are indicative only and do not represent complete verification results.

For two and three participants, the protocol satisfies all specified properties, demonstrating correctness and robustness. In contrast, verification for four participants either timed out after two hours or resulted in an Out-of-Memory error, reflecting the exponential growth in state space. The number of states and transitions for *Commit Preference* and *Vote Alignment* are identical for two and three participants, owing to the structural similarity of the corresponding formulas. However, the verification algorithms diverge, as PAT applies formula-specific optimisations, which in turn account for the observed differences in memory consumption. For *Vote Alignment*, memory usage is lower than that of the other properties; however, verification with four participants still exhausted memory due to exponential growth, whereas the other properties terminated by timeout before memory exhaustion.

5 Experiments and Evaluations

One particularly important direction for further investigation is analysing how key parameters—such as the number of participants, the number of allowed site failures (for both the coordinator and participants), and the permitted coordinator timeout—affect state-space size and verification time. This analysis provides deeper insights into the state-space explosion problem.

5.1 Impact of Participant Number on Verification Time

We investigated how increasing the number of participants could affect the verification time, by varying the value of `PARTICIPANT_NUMBER` from 2 to 4.

Configuration	Average Visited States	Average Total Transitions	Average Time Used
0/0/0	1190	2739	0.0341742s
0/0/5	8003	15227	0.2069795s
0/5/0	60033	178209	1.5335324s
5/0/0	10850	24128	0.2838439s
5/5/0	836960	3450350	22.6993082s
0/5/5	666922	1827464	18.6319547s
5/0/5	82698	219370	2.1904446s
5/5/5	6105938	22161937	186.7857362s
-1/-1/-1	N/A (TIMEOUT)	N/A (TIMEOUT)	N/A (TIMEOUT)

Table 2: Experimental Results with Varying Error and Timeout Counts

As shown in Table 1, increasing the participant count from 2 to 3 already leads to a dramatic rise in verification time—from a few seconds to several thousand seconds—while keeping all others unchanged. This trend suggests an exponential growth in complexity, and indeed, with 4 participants, verification becomes infeasible: none of the assertions complete within two hours. The number of visited states approximately doubles, and the total transitions quadruple on average. Notably, verifying the *Vote Alignment* property under the four-participant configuration resulted in an Out-of-Memory Exception, underscoring the severity of state explosion and the limitations of verification scalability.

5.2 Relationship Between Error Counts and Verification Time

To understand the impact of bounded failures and timeouts on verification time, we conducted a series of experiments using various configurations. Each configuration is denoted by a triplet $c/p/t$, where c is the value assigned to `COORDINATOR_BOUNDED_ERROR_COUNT`, p to `PARTICIPANT_BOUNDED_ERROR_COUNT`, and t to `COORDINATOR_BOUNDED_TIMEOUT_COUNT`. For each configuration, we ran multiple experiments across all properties and reported the aggregated (average) metrics for better clarity. Each individual verification result is available online.

From the first four entries in Table 2, we observe that the participant failures (p) have the most significant effect on state-space growth among the three error types. The next three entries explore combinations of two error types while disabling the third (i.e., assigning a value of 0). Among these, the combination of coordinator and participant failures (i.e., $c/p/0$) leads to the largest increase in state-space size and verification time.

The configuration 5/5/5 demonstrates a compounded, exponential effect of combining all three error types. In comparison, the unbounded configuration $-1/-1/-1$, which allows failures and timeouts to occur an unlimited number of times, and causes verification to fail due to time and memory exhaustion. PAT does not complete within the default 120-minute timeout, and increasing the timeout to 24 hours has no practical benefit due to severe system thrashing. These are intuitive: Failures may occur at nearly any execution point, whereas timeouts are restricted to specific waiting states, making site failures more impactful overall. However, the exponential growth in the state space is not immediately obvious from the empirical data alone. We therefore provide a mathematical model to illustrate the underlying combinatorial complexity.

Assume the following parameters:

- X: Number of coordinator execution points that may be interrupted by a site failure;
- Y: Number of execution points per participant that may be interrupted by a site failure;
- Z: Number of coordinator execution points that may be interrupted by a timeout, where $Z << X$;
- n: Maximum number of site failures allowed for the coordinator;
- m: Maximum number of site failures allowed for each participant;
- s: Maximum number of timeouts allowed for the coordinator;
- p: Number of participants involved.

Assuming failures and timeouts occur independently and uniformly across valid execution points, we can estimate the number of execution traces as follows:

- The number of possible execution traces for the coordinator is $X^n * Z^s$.
- The number of possible execution traces for each participant is Y^m.
- Assuming participants operate independently (a simplification, since synchronisation occurs in practice), the total number of possible interleaved execution traces for all participants is $(Y^m)^p$.
- Assuming further that the coordinator operates independently of participants, the total number of possible system execution traces becomes:

$$X^n * Z^s * (Y^m)^p.$$

This estimation assumes exactly n, m and s failures/timeouts. If we relax the constraint to allow *up* to those bounds (i.e., allowing fewer than the maximum), the number of possible traces increases substantially due to the explosion in branching behaviours. Each branch may correspond to a distinct execution path, resulting in exponential growth in explored states with respect to n, m, s, and the number of participants p. Techniques like partial-order reduction may mitigate this to some extent by merging equivalent states, but the fundamental combinatorial explosion remains.

This mathematical model provides strong evidence of the inherent complexity of verifying the 2PC protocol under failure conditions, and underscores the importance of bounding error parameters to achieve practical verification.

6 Conclusion and Future Work

We presented a formal model of the 2PC protocol with bounded failures, timeouts, recovery, and termination, and verified key correctness properties using PAT. Our results confirmed protocol robustness for up to three participants but revealed significant state-space explosion. Through detailed analysis, key contributors were identified: the number of sites involved, and the number of error types. These reflect two inherent properties of distributed systems——scalability and uncertainty——which together introduce intertwined layers of non-determinism.

Addressing this limitation remains an important area for future work, particularly in improving the efficiency of verification engines. One promising direction is the application of nested model checking on system abstractions. If a system exhibits hierarchical structure and can be decomposed into independent sub-problems, these can be verified in parallel with reduced need for synchronisation. This approach offers a potential pathway to mitigate state-space explosion in complex distributed systems.

References

1. Process analysis toolkit (pat) 3.5 user manual print. https://pat.comp.nus.edu.sg/resources/OnlineHelp/htm/
2. Network integration (visited at 13 June 2025), https://learn.microsoft.com/en-us/host-integration-server/what-is-his
3. Parallel commits: An atomic commit protocol for globally distributed transactions (visited at 13 June 2025), https://www.cockroachlabs.com/blog/parallel-commits/
4. Spanner: Truetime and external consitency (visited at 13 June 2025), https://cloud.google.com/spanner/docs/true-time-external-consistency
5. Specifying and simulating two-phase commit in lean4 (visited at 13 June 2025), https://protocols-made-fun.com/lean/2025/04/25/lean-two-phase.html
6. Tlaplus examples module twophase (visited at 13 June 2025), https://github.com/tlaplus/Examples/blob/master/specifications/transaction_\commit/TwoPhase.tla
7. Using fdb clusters as 2pc participants (visited at 13 June 2025), https://forums.foundationdb.org/t/using-fdb-clusters-as-2pc-participants/1060
8. Al-Bataineh, O.I., Reynolds, M.: Epistemic model checking of distributed commit protocols with byzantine faults. In: Proceedings of FormaliSE@ICSE 2019. pp. 51–60. IEEE CS (2019)
9. Atif, M.: Analysis and verification of two-phase commit & three-phase commit protocols. In: Proceedings of International Conference on Emerging Technologies. pp. 326–331 (2009)
10. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E.C., Lin, Q., Ooi, B.C.: Towards scaling blockchain systems via sharding. In: Proceedings of the 2019 International Conference on Management of Data. pp. 123–140. ACM Press (2019)
11. Dragojevic, A., Narayanan, D., Castro, M., Hodson, O.: Farm: Fast remote memory. In: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association (2014), https://www.microsoft.com/en-us/research/publication/farm-fast-remote-memory/
12. Hellings, J., Sadoghi, M.: Byshard: sharding in a byzantine environment. Proc. VLDB Endow. **14**(11), 2230–2243 (2021)
13. Kamil, S.N.S., Thomas, N.: Modelling and analysis of commit protocols with PEPA. In: Proceedings of the 14th European Workshop on Computer Performance Engineering. Lecture Notes in Computer Science, vol. 10497, pp. 266–281. Springer-Verlag (2017)
14. Sun, J., Liu, Y., Dong, J.S., Pang, J.: Pat: Towards flexible verification under fairness. In: Proceedings of the 21st Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 5643, pp. 709–714. Springer-Verlag (2009)
15. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer Publishing Company, 3rd edn. (2011)