EXTENDING THE INTELLIGENT ADAPTIVE PARTICIPANT'S PRESUMPTION PROTOCOL TO THE MULTI-LEVEL DISTRIBUTED TRANSACTION EXECUTION MODEL

Yousef J. Al-Houmaily

Department of Computer and Information Programs, Institute of Public Administration, Riyadh, Saudi Arabia

ABSTRACT

The "intelligent adaptive participant's presumption protocol" (iAP^3) is an integrated atomic commit protocol. It interoperates implicit yes-vote, which is a one-phase commit protocol, besides presumed abort and presumed commit, the most commonly pronounced two-phase commit protocol variants. The aim of this combination is to achieve the performance advantages of one-phase commit protocols, on one hand, and the wide applicability of two-phase commit protocols, on the other. iAP^{3}interoperates the three protocols in a dynamic fashion and on a per participant basis, in spite of the incompatibilities among the three protocols. Besides that, the protocol is backward compatible with the standardized presumed abort protocol. Whereas iAP^3 was initially proposed for the two-level (or flat) transaction execution model, this article extends the protocol to the multi-level distributed transaction execution model, the model adopted by the database standards and widely implemented in commercial database systems. Thus, broadening the applicability scope of the iAP^3.

KEYWORDS

Atomic Commit Protocols, Database Recovery, Database Systems, Distributed Transaction Processing, Two-Phase Commit, Voting Protocols

1. INTRODUCTION

The *two-phase commit* (2PC) protocol [1, 2] is the first known and used *atomic commit protocol* (ACP) [3]. It ensures atomicity of distributed transactions but with a substantial added cost to each transaction execution time. This added cost significantly affect the overall system performance. For this reason, a large number of 2PC variants and optimizations address this important issue (see [4] for a survey of such variants and optimizations).

One-phase commit (1PC) protocols [5, 6, 7] reduce the cost of commit processing by eliminating the *explicit* first phase of 2PC. However, these protocols achieve this at the expense of placing assumptions on either transactions or the database management systems. In 1PC protocols, each participant is required to acknowledge each operation after its execution. This is because, in these protocols, an operation *acknowledgment* (ACK) does not only mean that the transaction preserves the *isolation* and *cascadless* properties at the executing site, but it also means that the transaction is not in violation of any existing consistency constraints at the site. Although this assumption is not too restrictive since commercial systems implement *rigorous* schedulers and database standards specify operation ACK, it clearly restricts the implementation of applications that wish to utilize the option of *deferred consistency constraints validation*. This optionis currently part of the SQL standards [8] and, by using this option, the evaluation of the consistency constraints are

delayed until the end of the execution of transactions [9, 10]. Hence, when a transaction uses this option, there is a need to synchronize the evaluation of deferred constraints across all participating database sites at commit time of the transaction, making 1PC protocols unusable in this case.

The *adaptive participant's presumption protocol* (AP^3) [11] alleviates the above applicability limitation of 1PC protocols by integrating the *implicit yes-vote* (IYV) protocol [6], which is a one-phase commit protocol, with the best known two-phase commit variants, namely, *presumed abort* (PrA) [12] and *presumed commit* (PrC) [12]. Thus, achieving the performance advantages of 1PC protocols whenever possible, on one hand, and the broad applicability of 2PC protocols, on the other. The *Intelligent AP*^{3} (*i*AP^3) extends the (basic) AP^3 [13] by incorporating four advanced features that address and resolve four important issues in the design of atomic commit protocols: two of which enhance *efficiency* while the other two enhance *applicability*.

Whereas both of the (basic) AP^3 and the *i*AP^3 were proposed for the *two-level transaction execution* (TLTE) model, it is imperative to extend these protocols to the more general *multi-level transaction execution* (MLTE) model. This is to provide both of them with a pragmatically wider applicability scope as the MLTE model is the one currently adopted by database standards and implemented in the majority of commercial database management systems. For this reason and because the *i*AP^3 is a superset of the (basic) AP^3, this paper extends the *i*AP^3 to the MLTE model, forming the *multi-level i*AP^3(ML-*i*AP^3).

The structure of the rest of this paper is as follows: Section 2 presents the extension of the protocol to the MLTE model while Section 3 presents the extension of the advanced features of the *i*AP^3 to the MLTE model. Following that, Section 4 discusses the recovery aspects of the protocol in the events of failures. Lastly, Section 5 provides some concluding remarks.

2. THE BASICS OF THE ML-*i*AP^3

The main difference between the ML-*i*AP^3 and the two-level *i*AP^3 is existence of cascaded coordinators (i.e., non-root and non-leaf participants) in the execution trees of transactions. This type of participants, which act as root coordinators with their direct descendants and leaf participants with their direct ancestors, do not exist in the execution tree of a transaction in the two-level *i*AP^3. This is because a participant in two-level *i*AP^3 is either the root participant (i.e., the coordinator) or a leaf participant.

In ML-*i*AP^3, the behavior of cascaded coordinators depend on the selected protocol by each direct descendant and the finally decided protocol by the root coordinator, leading to three possible cases, which are as follows:

1. All participants are 1PC,
2. All participants are 2PC,
3. Participants are mixed 1PC and 2PC.

2.1 THE ML-*i*AP^3 WHEN ALL PARTICIPANTS ARE 1PC

In the ML-*i*AP^3, each operation submitted to a participant (whether the participant is a cascaded coordinator or leaf participant) is augmented with the identity of the root coordinator. Thus, when a participant receives an operation from a direct ancestor for the first time and participates in the execution of a transaction, following IYV protocol, the participant records the identity of the root coordinator in its *recovery-coordinators' list* (RCL) and force writes its RCL onto stable storage. The RCL is to facilitate recovery of the participant in the case it fails. A participant removes the

identity of a root coordinator from its RCL when it commits or aborts the last transaction submitted by the root coordinator.

As in other multi-level commit protocols, when a cascaded coordinator receives an operation from its direct ancestor in the transaction execution tree, it forwards the operation to the appropriate direct descendent(s) for execution. Since we are discussing the case where all participants are 1PC and as IYV is the used 1PC protocol in the ML-*i*AP^3, the behavior of a cascaded coordinator is similar in this case to the behavior of cascaded coordinators in multi-level IYV [6].

In IYV, a participant aborts a transaction if it fails to process one of its operations. Once the transaction is aborted, the participant sends a *negative acknowledgment* (NACK) to its direct ancestor. If the participant itself is a cascaded coordinator, it also sends an abort message to each implicitly prepared direct descendant. Then, the participant forgets the transaction. When the root coordinator or a cascaded coordinator receives NACK from a direct descendant, it aborts the transaction and sends abort messages to all implicitly prepared direct descendants and forgets the transaction. A root coordinator of a transaction also aborts the transaction when it receives an abort primitive from the transaction and sends an abort message to each direct descendant. If the descendent is a cascaded coordinator and receives an abort request from its direct ancestor, it sends an abort message to each direct descendant and forgets the transaction. When a leaf participant receives an abort request, it aborts the transaction without writing a decision log record for the transaction or acknowledging the decision. This is because the ML-*i*AP^3 adopts the presumed abort version of IYV whereby a participant never acknowledges an abort decision [6].

On the other hand, if a cascaded coordinator receives ACKs from all its direct descendants that have participated in the execution of an operation, the cascaded coordinator sends a collective ACK message to its direct ancestor in the transaction execution tree signaling the successful execution of the operation. This message also contains any redo log records generated during the execution of the operation whether at the cascaded coordinator's site or at any of its descendants. Thus, when a transaction finishes its execution, all its redo records are replicated at the root coordinator which is responsible for maintaining the replicated redo log records. The root coordinator also knows all the participants both leaf and cascaded coordinators by the time the transaction finishes its execution phase.

When the root coordinator receives a commit request from a transaction after the successful execution of all its operations, the coordinator commits the transaction. In this case, the coordinator force writes a commit log record. Then, it sends a commit message to each direct descendent. If the direct descendant is a leaf participant, it commits the transaction and writes a non-forced commit log record. The participant acknowledges the commit decision once the commit record is written onto the stable log.

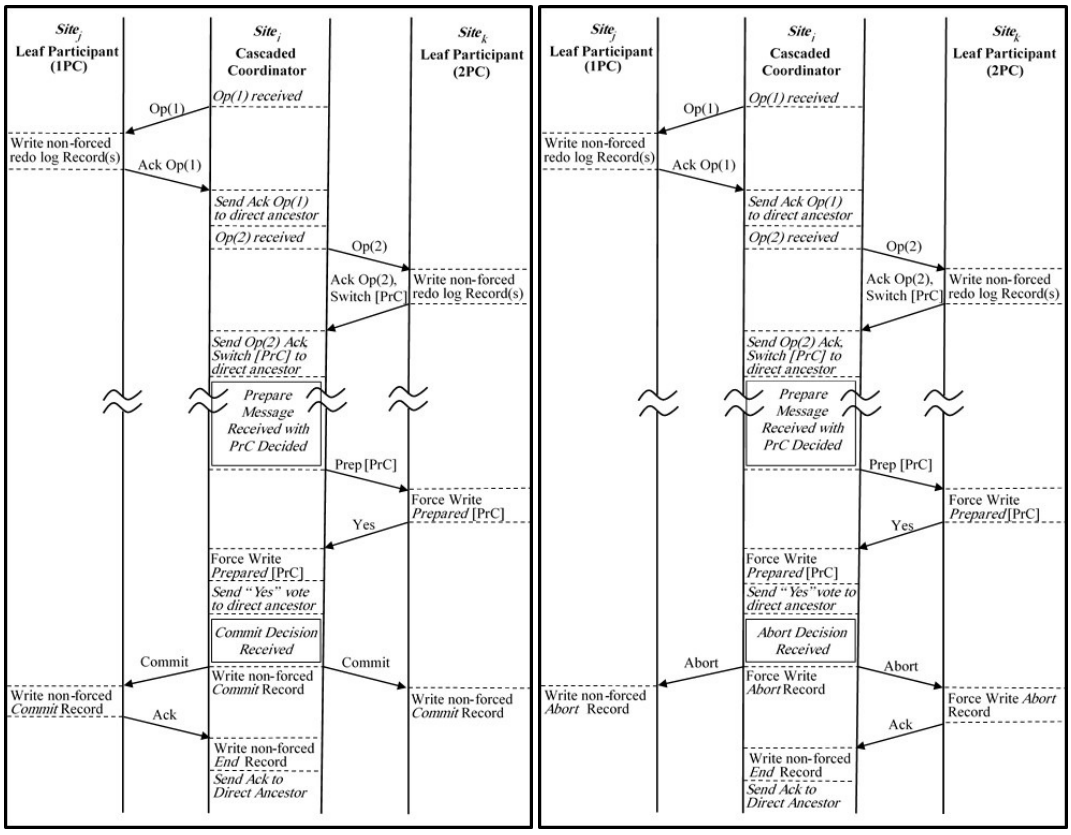
If the descendant is a cascaded coordinator, it commits the transaction, writes a non-forced commit log record, and forwards the commit decision to each of its direct descendants. Then, the cascaded coordinator waits for the commit ACKs. Once the commit ACKs arrive and the commit log record had been flushed onto the stable log, the cascaded coordinator writes a non-forced *end* log record. Then, it acknowledges the commit decision. Thus, the ACK of a cascaded coordinator serves as a collective ACK for the entire cascaded coordinator branch. When the root coordinator receives the commit ACKs from its direct descendants, it writes a non-forced *end* log record. Then, it forgets the transaction.

2.2 THE ML-*i*AP^3 WHEN ALL PARTICIPANTS ARE 2PC

In *i*AP^3, when a participant executes a deferred consistency constraint during the execution of a transaction, it *switches* to either PrA or PrC, depending on the anticipated results of the

consistency constraint. Thus, at the end of the transaction execution phase, the coordinator declares the transaction as 2PC if *all* participants have switched to 2PC. If all participants have switched to PrC, the coordinator selects PrC. Otherwise, the coordinator selects PrA. In either case, the *i*AP^3 can be extended to the MLTE model in a manner similar to the multi-level PrC and multi-level PrA, depending on the selected protocol. The only distinction between the ML-*i*AP^3 and the other two protocols is that the coordinator has to inform the participants about the finally decided protocol during the first phase. In addition, when PrC is used, the ML-*i*AP^3 does not realize the commit presumption of PrC on every two adjacent levels of the transaction execution tree. This is to reduce the costs associated with the *initiation* (or *collecting*) records of PrC. Thus, in this respect, the ML-*i*AP^3 is similar to the *rooted PrC* in which only the root coordinator force writes an initiation log record for the transaction and not cascaded coordinators [14].

Figure 1. Mixed participants in a 2PC cascaded coordinator's branch when PrC is decided.



a. Commit case b. Abort case

2.3 THE ML-*i*AP^3 WHEN PARTICIPANTS ARE MIXED 1PC AND 2PC

Based on the information received from the different participants during the execution of a transaction, at commit time, the coordinator of the transaction knows the protocol of each of the participants. It also knows the execution tree of the transaction. That is, it knows all the ancestors of each participant and whether a participant is a cascaded coordinator or a leaf participant. Based on this knowledge, the coordinator considers a direct descendant to be 1PC if the descendant and *all* the participants in its branch are 1PC. Otherwise, the coordinator considers the direct descendant 2PC. For a 1PC branch, the coordinator uses the 1PC part of ML-*i*AP^3 with the branch, as we discussed in Section 2.1. For a 2PC branch, the coordinator uses the decided 2PC

protocol variant regardless of whether the direct descendant is 1PC or 2PC. That is, the coordinator uses the 2PC part of the ML-*i*AP^3 discussed in Section 2.2. Thus, with the exception in the way a coordinator decide on which protocol to use with each of its direct descendants, the coordinator's protocol proceeds as in two-level *i*AP^3 [13].

In ML-*i*AP^3, each leaf participant behaves in the same way as in two-level *i*AP^3. This is regardless of whether the leaf participant descends from a 1PC or 2PC branch. That is, a participant behaves as 1PC participant if it has not requested to switch protocol or as the decided 2PC protocol variant if has made such a request during the execution of the transaction.

On the other hand, the behaviour of cascaded coordinators is different and depends on the types of its descendant participants in the branch. A cascaded coordinator uses multi-level IYV when all the participants in its branch, including itself, are 1PC. Similarly, a cascaded coordinator uses the multi-level version of the decided 2PC protocol variant when all the participants in its branch, including itself, are 2PC. Thus, in the above two situations, a cascaded coordinator uses ML-*i*AP^3 as discussed in the previous two sections.

When the protocol used by a cascaded coordinator is different from the protocol used by at least one of its descendants (not necessarily a direct descendant), there are two scenarios to consider. The first scenario is when the cascaded coordinator is 2PC while the second scenario is when the cascaded coordinator is 1PC. Since, for each scenario, cascaded coordinators behave the same way at any level of the transaction execution tree, below we discuss the case of the last cascaded coordinator in a branch with mixed 1PC and 2PC protocols.

2.3.1 SCENARIO ONE: A 2PC CASCADED COORDINATOR'S BRANCH WHEN PrC IS DECIDED

When PrC is decided and a cascaded coordinator with mixed participants receives a prepare message from its ancestor after the transaction has finished its execution, the cascaded coordinator forwards the message to each 2PC participant indicating the decided PrC protocol (Figure 1). Then, it waits for the descendants' votes. If any descendant has decided to abort, the cascaded coordinator force writes an abort log record, aborts the transaction, sends a "no" vote to its direct ancestor and an abort message to each prepared to commit direct descendant (including 1PC descendants). Then, it waits for the ACKs of the prepared 2PC direct descendants. Once the cascaded coordinator receives the required ACKs, it writes a non-forced end log record. Then, it forgets the transaction. On the other hand, when the cascaded coordinator and all its 2PC direct descendants votes "yes", the cascaded coordinator force writes a prepared log record. Then, it sends a collective "yes" vote, reflecting the vote of the entire branch, to its direct ancestor and waits for the final decision.

If the final decision is a commit (Figure 1 (a)), the cascaded coordinator forwards the decision to each of its direct descendants, both 1PC and 2PC, and writes a commit log record. The commit log record of the cascaded coordinator is written in a non-forced manner, following PrC protocol. Unlike PrC, however, a cascaded coordinator expects each 1PC participant to acknowledge the commit message but not 2PC participants since they follow PrC. When a cascaded coordinator receives ACKs from 1PC participants, it writes a non-forced end log record. Once the record is written onto the stable log, the cascaded coordinator sends an ACK to its direct ancestor. Then, it forgets the transaction.

On the other hand, if the final decision is an abort (Figure 1 (b)), the cascaded coordinator sends an abort message to each of its descendants and writes a forced abort log record (following PrC protocol). When 2PC participants acknowledge the abort decision, the cascaded coordinator writes a non-forced end log record. Once the end record is written onto stable storage due to a

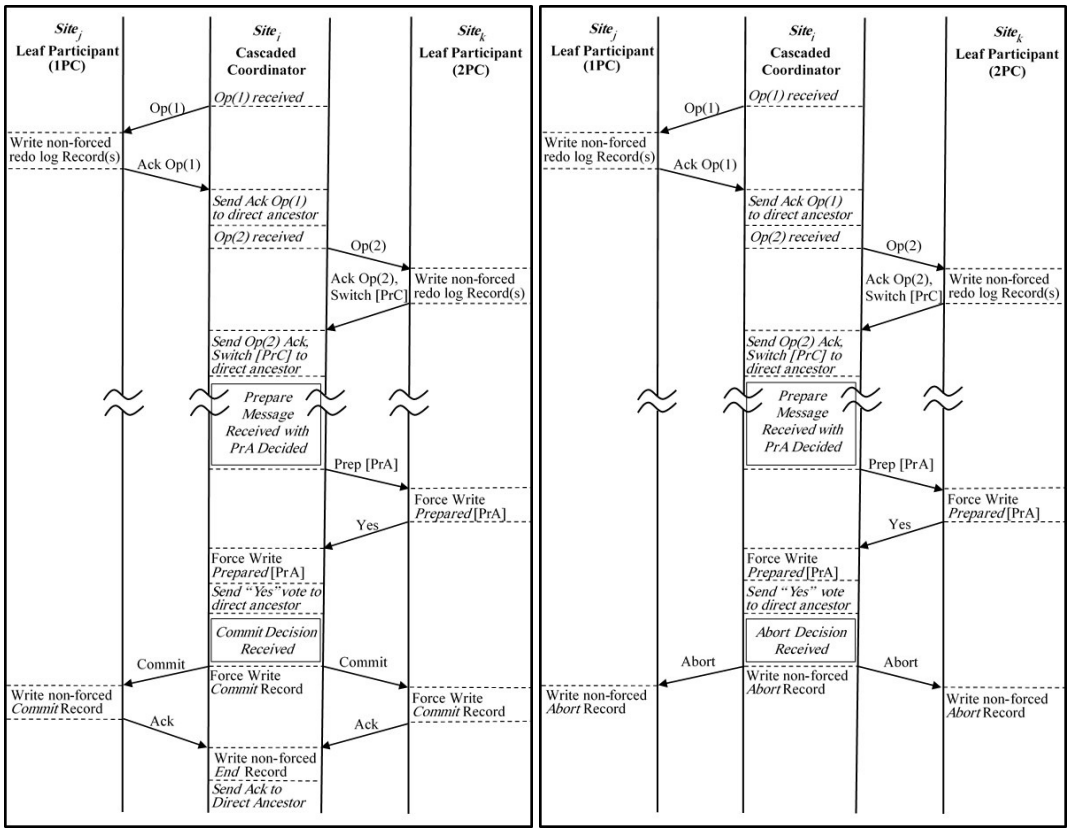
subsequent flush of the log buffer, the cascaded coordinator sends a collective ACK to its direct ancestor and forgets the transaction.

It should be noted that a cascaded coordinator, in this scenario, has to acknowledge both commit and abort decisions. A commit ACK reflects the ACKs of all 1PC participants while an abort ACK reflects the ACKs of all 2PC participants (including the cascaded coordinator's ACK).

2.3.2 SCENARIO ONE: A 2PC CASCADED COORDINATOR'S BRANCH WHEN PrA IS DECIDED

When PrA is decided and a cascaded coordinator with mixed participants receives a prepare message from its ancestor after the transaction has finished its execution, the cascaded coordinator forwards the message to each 2PC participant indicating the decided PrA protocol (Figure 2). Then, it waits for the descendants' votes. If any descendant has decided to abort, the cascaded coordinator writes a non-forced abort log record, aborts the transaction, sends a "no" vote to its direct ancestor and an abort message to each prepared to commit direct descendant (including 1PC descendants). Then, it forgets the transaction. On the other hand, if the cascaded coordinator and all its 2PC direct descendants votes "yes", the cascaded coordinator force writes a prepared log record. Then, it sends a collective "yes" vote, reflecting the vote of the entire branch, to its direct ancestor and waits for the final decision.

Figure 2. Mixed participants in a 2PC cascaded coordinator's branch when PrA is decided.



a. Commit case b. Abort case

If the final decision is a commit (Figure 2 (a)), the cascaded coordinator, following PrA, force writes a commit log record and forwards the decision to each of its direct descendants (both 1PC

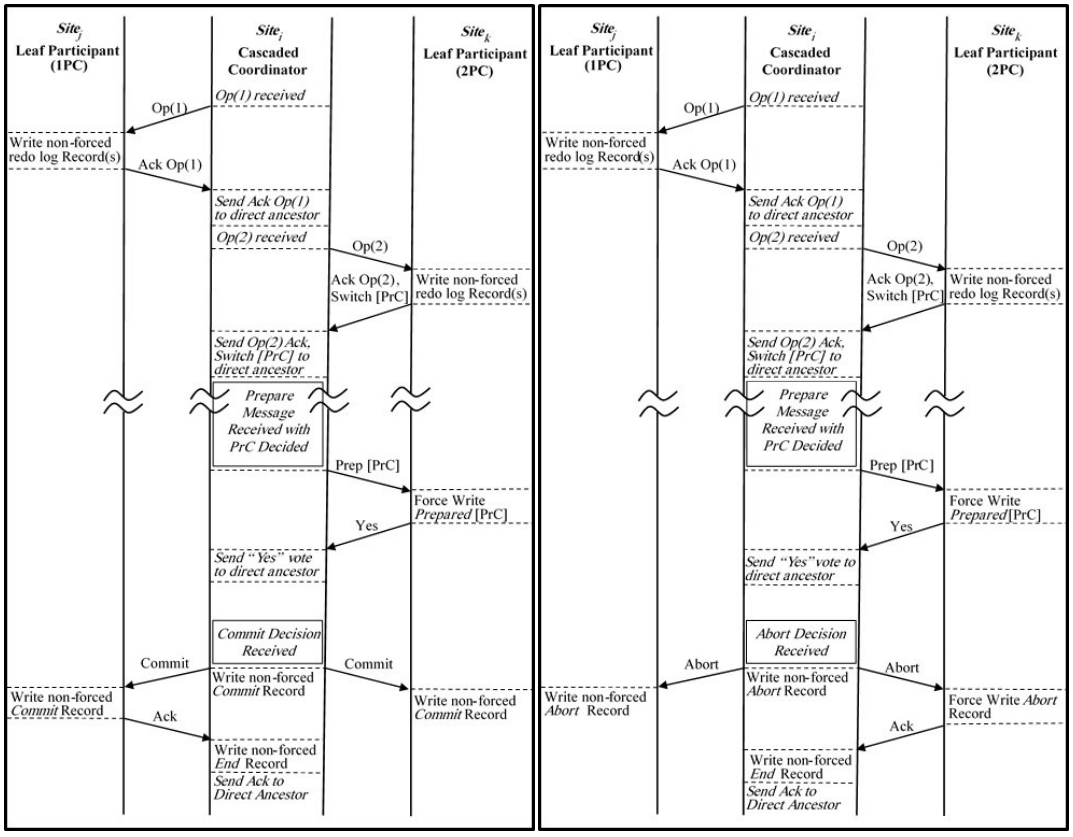
and 2PC). Then, the cascaded coordinator waits for the direct descendants' ACKs. When the cascaded coordinator receives ACKs from both 1PC and 2PC direct descendants, it writes a non-forced end log record. When the end record is written onto the stable log, the cascaded coordinator sends a collective ACK to its direct ancestor and forgets the transaction.

On the other hand, if the final decision is an abort (Figure 2 (b)), the cascaded coordinator sends an abort message to each of its descendants and writes a non-forced abort log record (following PrA protocol). Then, it forgets the transaction.

2.3.3 SCENARIO TWO: A 1PC CASCADED COORDINATOR'S BRANCH WHEN PrC IS DECIDED

In ML-*i*AP^3, a 1PC cascaded coordinator with 2PC participants is dealt with as 2PC with respect to messages. Based on that, when a 1PC cascaded coordinator receives a prepare message from its ancestor, it forwards the message to each 2PC participant and waits for their votes. If any participant has decided to abort, assuming that PrC is decided, the cascaded coordinator aborts the transaction. On an abort, the cascaded coordinator force writes an abort log record, then, sends a "no" vote to its direct ancestor and an abort message to each prepared participant (including 1PC participants). After that, it waits for the abort ACKs from the prepared PrC participants. Once the ACKs arrive, the cascaded coordinator writes a non-forced end log record. Then, it forgets the transaction. If all the PrC participants had voted "yes", the cascaded coordinator sends a "yes" vote. This vote reflects the vote of the entire branch, as shown in Figure 3. Then, the cascaded coordinator waits for the final decision.

Figure 3. Mixed participants in a 1PC cascaded coordinator's branch when PrC is decided.



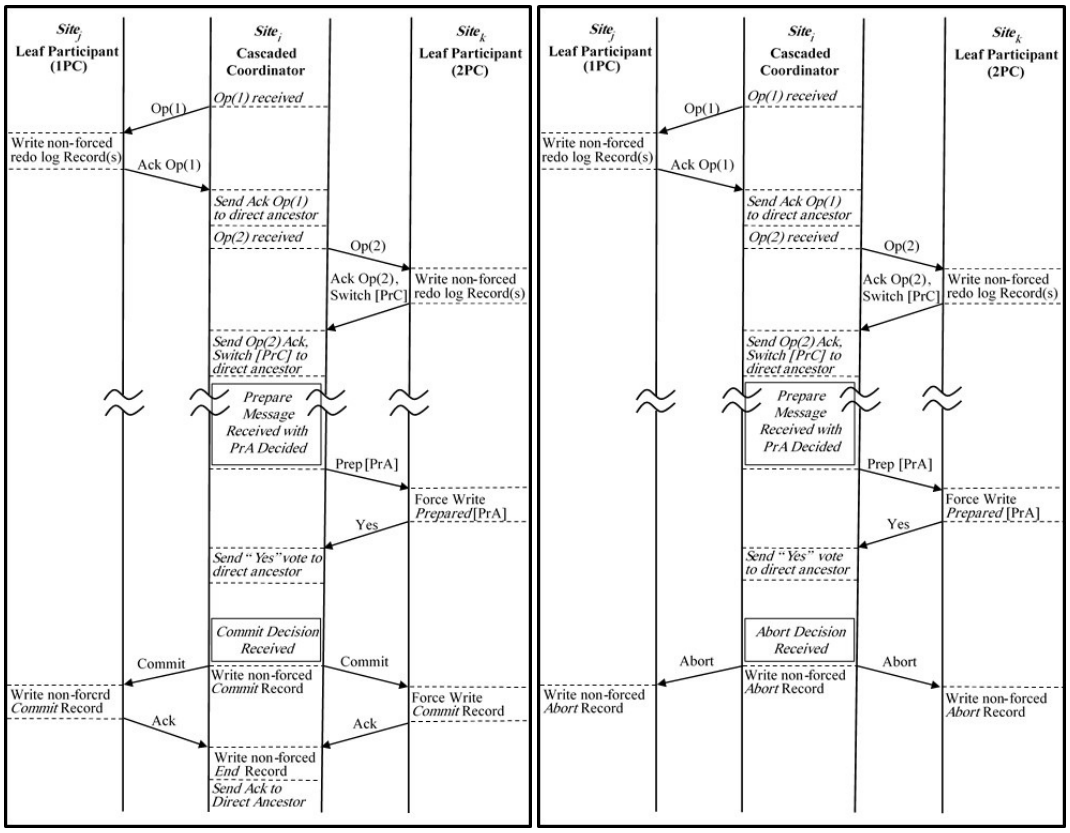
a. Commit case b. Abort case

If the final decision is a commit (Figure 3 (a)), the cascaded coordinator forwards the decision to each of its direct descendants, both 1PC and 2PC, and writes a non-forced commit log record, following IYV protocol. Unlike IYV, however, a cascaded coordinator expects each 1PC participant to acknowledge the commit message but not 2PC participants since they follow PrC. When a cascaded coordinator receives ACKs from 1PC participants, it writes a non-forced end log record. Once the end record is written onto the stable log due to a subsequent flush to the log buffer, the cascaded coordinator sends a collective ACK to its direct ancestor. Then it forgets the transaction.

On the other hand, if the final decision is an abort (Figure 3 (b)), the cascaded coordinator sends an abort message to each of its descendants and writes a non-forced abort log record (following IYV protocol). When 2PC participants acknowledge the abort decision, the cascaded coordinator writes a non-forced end log record. Once the end record is written onto the stable storage, the cascaded coordinator sends an ACK to its direct ancestor. Then, it forgets the transaction.

Notice that a 1PC participant that is cascaded coordinator has to acknowledge both commit as well as abort decisions. This is similar to the case of a 2PC cascaded coordinator with mixed participants and PrC is decided, a commit ACK reflects the ACKs of all 1PC participants (including the cascaded coordinator's ACK) while an abort ACK reflects the ACKs of all 2PC participants.

Figure 4. Mixed participants in a 1PC cascaded coordinator's branch when PrA is decided.



a. Commit case b. Abort case

2.3.4 SCENARIO TWO: A 1PC CASCADED COORDINATOR'S BRANCH WHEN PrA IS DECIDED

When a 1PC cascaded coordinator receives a prepare message from its ancestor, it forwards the message to each 2PC participant and waits for their votes. If any participant has decided to abort, assuming that PrA is decided, the cascaded coordinator aborts the transaction. In this case, the cascaded coordinator writes a non-forced abort log record. Then, it sends a "no" vote to its direct ancestor and an abort message to each of its direct descendants. After that, it forgets the transaction. On the other hand, if the cascaded coordinator and all the PrA participants votes "yes", the cascaded coordinator sends a "yes" vote. This vote reflects the vote of the entire branch, as shown in Figure 4. Then, the cascaded coordinator waits for the final decision.

If the final decision is a commit (Figure 4 (a)), the cascaded coordinator forwards the decision to each of its direct descendants, both 1PC and 2PC, and writes a commit log record. The commit log record of the cascaded coordinator is written in a non-forced manner, following IYV protocol. When the cascaded coordinator receives ACKs from all its direct descendants, it writes a non-forced end log record and sends a collective ACK to its direct ancestor. Then, it forgets the transaction. The ACK message of the cascaded coordinator is sent only after the end record is written onto the stable log due to a subsequent forced write of a log record or log buffer overflow. On the other hand, if the final decision is an abort (Figure 4 (b)), the cascaded coordinator aborts the transaction, sends an abort message to each of its descendants and writes a non-forced abort log record (following IYV protocol). Then, it forgets the transaction.

3. THE ML-*i*AP^3 AND THE ADVANCED FEATURES OF THE *i*AP^3

This section extends the four advanced features of two-level *i*AP^3 to the multi-level distributed transaction execution model. For completeness purposes, a presentation to the details of each feature in the context of two-level *i*AP^3 precedes the extension of the feature to the ML-*i*AP^3.

3.1. THE ML-*i*AP^3 AND READ-ONLY TRANSACTIONS

For read-only transactions, *i*AP^3 uses the principles of the *unsolicited update-vote* (UUV) optimization [14]. More specifically, once a transaction starts executing, it is marked as a read-only transaction. It continues this way so long as its coordinator does not receive any ACK that contains redo log records and any ACK that indicates a protocol switch. This is because only update operations generate redo log records or are associated with consistency constraints. When the coordinator receives an ACK that contains redo log records or an ACK with a switch flag, it means that the transaction has become an update transaction. Accordingly, the coordinator changes the state of the transaction in its protocol table.

At commit time of the transaction, the coordinator refers to its protocol table and identifies read-only participants and update participants. For an update participant, the coordinator also identifies the chosen protocol by the participant. If all participants are read-only, the coordinator sends a *read-only* message to each one of them and forgets the transaction without writing any log records. Otherwise, the coordinator initiates the voting phase with 2PC participants (if any) and, at the same time, sends a read-only message to each read-only participant. Then, the coordinator removes the read-only participants from its protocol. A read-only message received by a participant means that the transaction has finished its execution. Consequently, the participant releases all the resources held by the transaction once it has received a read-only message without writing any log records or acknowledging the message. For update participants, both the coordinator and each of the participants follow *i*AP^3.

When all the participants are read-only, the ML-*i*AP^3 works in a manner similar to the two-level *i*AP^3. That is, the coordinator of a read-only transaction sends a read-only message to each of its

direct descendants and then forgets the transaction. Similarly, when a cascaded coordinator receives a read-only message, it releases all the resources held by the transaction and sends a read-only message to each of its descendants. A leaf participant also releases all the resources held by the transaction and forgets the transaction after receiving a read-only message. Thus, for an exclusively read-only transaction, the coordinator sends one message to each direct descendant without writing any log records. A cascaded coordinator also sends one message to each of its direct descendants without writing any log records. On other hand, a leaf participant does not send any messages or write any log records.

If the transaction is partially read-only, the coordinator sends a read-only message to each read-only direct descendant. For update direct descendants, the coordinator initiates the decided protocol with them. Similarly, each cascaded coordinator sends a read-only message to each read-only direct descendant and follows the decided protocol with the other direct descendants. A cascaded coordinator knows which of its direct descendants is read-only and which is not based on the received ACK messages and the included control information during the execution of the transaction. The behaviour of leaf participants remain the same as in two-level *i*AP^3. Hence, in ML-*i*AP^3, only non-read-only participants (both cascaded coordinators and leaf participants) continue in their involvement in the finally decided commit protocol until its end.

3.2. THE ML-*i*AP^3 AND FORWARD RECOVERY

In *i*AP^3, instead of aborting a partially executed transaction during the recovery procedure after a communication or a site failure, the transaction is allowed to continue its execution after the failure is fixed. This *forward recovery* option is applicable so long as the transaction is *context-free* [15] and it was originally defined and used in IYV [6]. When a transaction chooses to use this option, it indicates its choice to its coordinator at the beginning of its execution. This option allows a transaction to wait on any delays that it may encounter during its execution due to a failure instead of aborting it.

When a transaction chooses the forward recovery option, each 1PC participant replicates both the redo log records and the read locks of the transaction at the coordinator's site. This is accomplished by propagating the generated redo records and read locks in the ACK messages of the operations of the transaction as the transaction executes at the participant. In this way, the coordinator's protocol table contains a partial image of each 1PC participant log and lock table.

After a participant failure, the participant can re-construct the missing parts of its log and lock table with the help of the coordinators. Thus, recovering the states of forward recoverable transactions and allowing them to continue their execution instead of aborting them.

As it is impossible, in general, to determine in advance whether a transaction that has chosen the option of forward recovery that it will create a run-time context at a participant or not, the *i*AP^3 has such transactions at run-time and override their choices. In the *i*AP^3, when the first operation of a 1PC forward recoverable transaction creates a context at a participant, the participant changes the state of the transaction to non-forward-recoverable and informs the transaction coordinator accordingly. This is accomplished by requesting a protocol switch in the ACK of the operation. After that, the participant refrains from sending any read locks for the transaction.

When the coordinator receives an ACK indicating a protocol switch for a 1PC forward-recoverable transaction, the coordinator marks the transaction as non-forward recoverable in its protocol table. After that, the coordinator starts informing the other 1PC participants as it submits new operations to them for execution.

The forward recovery option of two-level *i*AP^3 can be extended to the MLTE model in a straight forward manner. As in two-level *i*AP^3, this option is applicable to transactions that are 1PC across all participants. That is, if the state of a transaction becomes 2PC at a participant, then, the transaction cannot be forward recoverable. Additionally, when a transaction chooses the option of forward recovery, its coordinator indicates that the transaction is forward recoverable in the first operation that it sends to each participant.

As in two-level *i*AP^3, each participant propagates the read locks that the transaction acquires and the redo log records that are generated during the execution of the transaction to its ancestor along with the operations' ACKs. These ACKs and control information are eventually received by the coordinator and stored in a similar way as in two-level *i*AP^3. If a participant decides to change the state of the transaction to become non-forward recoverable, the participant informs its ancestor about this state change in the ACK of the operation that caused the participant to change the state of the transaction. This state change is propagated along with the ACK message of the operation that caused the state change from one ancestor to another until it reaches the coordinator. At that point, the coordinator becomes aware of the change and informs the other participants as it submits new operations to them for execution.

3.3. THE ML-*i*AP^3 AND UPDATING LARGE AMOUNTS OF DATA

In *i*AP^3, when a transaction updates large amounts of data at a participant and the updated data is prohibitively large to be propagated to the coordinator of the transaction, the participant uses a *large amounts of data* (LAD) flag. More specifically, when a 1PC participant updates large amounts of data during the execution of an operation and the updated data is not associated with deferred consistency constraints, it sets this flag in the ACK of the operation and switches to PrC. If the updated data is associated with deferred constraints, the participant chooses the appropriate 2PC variant. The choice of the 2PC variant, in this case, depends on the tendency of the evaluation of these constraints at commit time of the transaction. Once the participant has switched to 2PC, it does not send any more redo log records in the ACKs of update operations. Besides that, the participant changes the state of the transaction to be non-forward recoverable. That is, of course, if the transaction was set as forward recoverable. If this occurs, the participants also stops sending any more read locks for the transaction to the transaction's coordinator. If a participant switches to PrC and later on executed an operation that is associated with deferred consistency constraints that tend to be violated, the participant changes its previously selected protocol to PrA in the ACK of the operation. Thus, PrA is used only when the transaction is associated with deferred constraints that tend to be violated at commit processing time.

When the coordinator receives an ACK with a set LAD flag from a participant, it marks the participant as either PrC or PrA in its protocol table, depending on the chosen 2PC protocol by the participant. The coordinator also changes the state of the transaction to be non-forward recoverable if the transaction was set as forward recoverable and starts informing the other 1PC participants about this new state of the transaction. This is accomplished by indicating the state change in the first operation that the coordinator sends to each of the other 1PC participants. When a participant receives a state change indication in an operation, the participant stops sending any more read locks in the ACK of each operation that it executes on behalf of the transaction.

Extending the above *i*AP^3 feature to the MLTE model is straight forward as any participant can set the LAD flag when necessary. Once the flag is set, the participant becomes a 2PC participant. Then, the behavior of the participant depends on the location of the participant in the transaction execution tree. That is, if the participant is a leaf participant, it follows the 2PC protocol that is finally decided by the coordinator. On the other hand, if the participant is a cascaded coordinator,

it follows one of the two extensions to the basic protocol discussed in Section 2.2 and Section 2.3, depending on the finally chosen protocol by the coordinator.

3.4. THE ML-*i*AP^3 AND BACKWARD COMPATIBILITY

The *i*AP^3 protocol is backward compatible with both PrA coordinators and PrA participants. In the *i*AP^3, an *i*AP^3 participant keeps a list called *presumed-abort coordinators* (PAC) in which it records the identities of all pre-existing coordinators that use PrA. The PAC list is created at system installation time and is continuously updated as new PrA coordinators join or existing ones leave the system. Thus, this list is maintained so long as there are some PrA coordinators exist in the system.

An *i*AP^3 participant refers to its PAC list after the initiation of any new transaction at its site. This is to determine if the coordinator of the transaction is a pre-existing PrA coordinator. If the coordinator is a pre-existing PrA site, the participant deals with it using PrA. That is, the participant marks the transaction as a PrA transaction in its protocol table and does not include any redo log records or read locks in the ACK of any operation that it executes for the transaction. Besides that, the participant deals with the coordinator using PrA at commit processing time of the transaction, including the use of the traditional read-only optimization [12] if this optimization is supported by the coordinator.

In *i*AP^3, an *i*AP^3 coordinator keeps a list called *presumed-abort participants* (PAP) in which it records the identities of all pre-existing PrA participants. Before launching a transaction at a participant, the coordinator refers to its PAP list to determine if the participant is a pre-existing PrA site. If the participant is a pre-existing PrA, converts the transaction to become non-forward recoverable, given that the transaction was set as forward recoverable, before initiating the transaction at the participant. Then, it starts informing the other participants as it sends them new operations for execution.

Using the PAC and PAP lists used in two-level *i*AP^3, the protocol can be easily extended to the MLTE model. In ML-*i*AP^3, when the root coordinator or any participant is a pre-existing PrA site, the whole transaction becomes PrA. When the root coordinator is an *i*AP^3 site, it knows that a transaction has to be PrA once the transaction submits an operation that is to be executed at a pre-existing PrA participant according to the stored PAP list at the coordinator's site. Once the coordinator knows that the transaction has to be PrA, it informs all *i*AP^3 participants as it submits new operations to them for execution or during the commit processing stage (if the operations to be executed by pre-existing PrA participants are the last operations to arrive from the transaction for execution). on the other hand, when the coordinator is a PrA site, an *i*AP^3 participant knows that the transaction has to be PrA once it receives the first operation from the coordinator. This is because the identity of the coordinator is included in the PAC list of the participant. Hence, a transaction that executes at a pre-exiting PrA site becomes PrA across all sites. As such, the root coordinator and all participants follow multi-level PrA. Not only that, but for read-only transactions, all sites follow the traditional read-only optimization if it is supported by the root coordinator.

4.FAILURE RECOVERY IN THE ML-*i*AP^3

The *operational correctness criterion* [16] represents a guiding principal for the correctness of any practical ACP. It specifically states that: 1) all sites participating in a transaction's execution should reach the same outcome for the final state of the transaction, and 2) all participating sites should be able to, eventually, forget the transaction and to garbage collect the transaction's log

records. The operational correctness criterion should hold even in the case of failures and regardless of their number and frequency.

Thus far, we extensively discussed the ML-*i*AP^3 during normal processing. The discussion clearly shows that the protocol strictly observes the operational correctness criterion. This section shows that ML-*i*AP^3 also observes the operational correctness criterion in the case of site and communication failures, which are detected by *timeouts*. The section starts by discussing the recovery aspects of the protocol in the presence of communication failures. Then, it discusses the recovery aspects of the protocol in the presence of site failures.

4.1. COMMUNICATION FAILURES

4.1.1 A ROOT COORDINATOR COMMUNICATION FAILURES

In ML-*i*AP^3, there are three points at which a coordinator may timeout while waiting for a message. In the first point, a coordinator may timeout while waiting for an operation ACK from a participant. When a coordinator times out while waiting for an operation ACK from a participant, it aborts the transaction and sends out abort messages to the rest of the participants.

In the second point, a coordinator may timeout while waiting for a vote from a 2PC participant. When this occurs, the communication failure is dealt with as if it was a "no" vote, leading to an abort decision. As during normal processing, in this case, the coordinator sends out abort messages to all accessible participants and waits for the required ACKs. The anticipated ACKs depend on the finally decided protocol that the coordinator sent in the prepare messages (i.e., the ACKs of PrC participants when PrC is used with *i*AP^3 participants). These ACKs enable the coordinator to write an end log record for the transaction and to forget it. If a participant has already voted "yes" before a communication failure, the participant is left blocked. In this case, it is the responsibility of the participant to inquire about the transaction's status after the failure is fixed. When a participant inquires about a transaction status, it has to include the used protocol with the transaction in the inquiry message. This piece of information guides the ancestors of the participant in their response to the inquiry message if the transaction has already been forgotten. If the transaction was using a presumed-abort based protocol, the direct ancestor of the participant can respond to the inquiry message of the participant with an abort decision without the need to consult with its own direct ancestor. On the other hand, if the used protocol is PrC, the direct ancestor cannot make such a decision alone and has to consult with its own direct ancestor until possibly reaching the root coordinator. This is because only the root coordinator force writes a switch log record in *i*AP^3 and can accurately determine the status of the transaction.

The third point occurs when the coordinator of a transaction times out while waiting for the ACKs of a final decision. As the coordinator needs these ACKs in order to complete the protocol and to forget the transaction, it re-submits the decision to the appropriate participants once communication failures are fixed. In *i*AP^3, a coordinator re-submits a commit decision to each inaccessible 1PC participant, a pre-existing PrA and 2PC *i*AP^3 participant when PrA is used with *i*AP^3 participants. For an abort decision, the coordinator re-submits the decision to each inaccessible *i*AP^3 participant when PrC is used 2PC *i*AP^3 participants. When a participant receives a decision, it complies with the decision, if it has not done so before the failure, and then acknowledges the decision.

4.1.2 A LEAF PARTICIPANT COMMUNICATION FAILURES

Similar to the root coordinator communication failures, there are three points at which a leaf participant may timeout while waiting for a message. In the first point, a participant may detect a

communication failure and it has a pending operation ACK. In this case, the participant aborts the transaction.

The second point is when the participant detects a communication failure and the participant has no pending operation ACK. If this occurs and the transaction is 1PC at the participant, the participant is blocked until the communication failure is fixed. Once the failure is fixed, the participant inquires about the transaction's status. The participant will receive either a final decision or a *still active* message. If the participant receives a decision, it enforces the decision.

The participant also acknowledges the decision if it is a commit decision. If the participant receives a still-active message, it means that the transaction is still executing in the system and no decision has been made yet regarding its final status. Based on that, the participant waits for further instructions. On the other hand, if the communication failure occurs and the participant is 2PC, whether it is an *i*AP^3 or a pre-existing PrA, the participant aborts the transaction.

The third point occurs when a participant is 2PC and is in a prepared to commit state. In this case, if the participant is an *i*AP^3, the participant inquires its direct ancestor about the status of the transaction with a message that indicates the used protocol with the transaction once the communication failure is fixed. Otherwise, being pre-existing PrA, the participant does not indicate its used protocol in the inquiry message. In either of the two cases, the participant will receive the correct final decision from its direct ancestor regardless of whether the transaction is still remembered by its ancestors in the transaction execution tree or not.

4.1.3 A CASCADED COORDINATOR COMMUNICATION FAILURES

In ML-*i*AP^3, there are six points at which a cascaded coordinator may detect a communication failure. Three of these failures may occur with the direct ancestor while the other three may occur with a direct descendant.

4.1.3.1 COMMUNICATION FAILURES WITH THE DIRECT ANCESTOR

In the first point, a cascaded coordinator may detect a communication failure and it has a pending operation ACK (either generated locally or received from one of its direct descendants). In this case, the cascaded coordinator aborts the transaction and sends an abort message to each of its direct descendants.

The second point is when a cascaded coordinator detects a communication failure and it does not have a pending operation ACK. In this case, if the transaction is 1PC at the cascaded coordinator, the cascaded coordinator is blocked until communication is re-established with its direct ancestor. Once the communication failure is fixed, the cascaded coordinator inquires its direct ancestor about the transaction's status. The cascaded coordinator will receive either a final decision or a *still active* message. In the former case, the cascaded coordinator enforces the final decision. Then, if the decision is commit, the cascaded coordinator also acknowledges it. In the latter case, the cascaded coordinator waits for further operations. On the other hand, if the communication failure occurs and the cascaded coordinator or one of its direct descendants is 2PC, the cascaded coordinator aborts the transaction. Once the cascaded coordinator has aborted the transaction, it sends out an abort message to each of its direct descendants.

The third point occurs when a cascaded coordinator is 2PC and is in a prepared-to-commit state. In this case, if the cascaded coordinator is an *i*AP^3, it inquires its direct ancestor about the status of the transaction, indicating the used protocol with the transaction. If the cascaded coordinator is a pre-existing PrA, it also inquires its direct ancestor about the status of the transaction with a message that, of course, does not indicate the used protocol. In either of the two cases, the

cascaded coordinator will receive the correct final decision from its direct ancestor regardless of whether the transaction is still remembered by its ancestors in the transaction execution tree or not.

4.1.3.2 COMMUNICATION FAILURES WITH A DIRECT DESCENDANT

As mentioned above, there are three points at which a cascaded coordinator may timeout while waiting for a message from a direct descendant. In the first point, a cascaded coordinator may timeout while waiting for an operation ACK. In this case, the cascaded coordinator aborts the transaction and sends out an abort message to its direct ancestor and to each of its accessible direct descendants.

In the second point, a cascaded coordinator may timeout while waiting for the votes of 2PC direct descendants. In this case, the cascaded coordinator treats communication failures as "no" votes and aborts the transaction. On an abort, the cascaded coordinator sends out an abort message to its direct ancestor and each accessible direct descendant. Then it waits for the required ACKs. The anticipated ACKs depend on the finally decided protocol that the root coordinator sent in the prepare messages (i.e., the ACKs of PrC participants when PrC is used with *i*AP^3 participants). These ACKs are necessary for the cascaded coordinator. They enable the cascaded coordinator to write an end log record for the transaction and to forget it. If a participant has already voted "yes" before a communication failure, the participant is left blocked. In this case, it is the responsibility of the participant to inquire about the transaction's status after the failure is fixed. When a participant inquires about a transaction status, it has to include the used protocol with the transaction in the inquiry message. This piece of information guides the ancestors of the participant in their response to the inquiry message if the transaction has already been forgotten. If the transaction was using a presumed-abort based protocol, the direct ancestor of the participant can respond to the inquiry message of the participant with an abort decision without the need to consult with its own direct ancestor. On the other hand, if the used protocol is PrC, the direct ancestor cannot make such a decision alone and has to consult with its own direct ancestor until possibly reaching the root coordinator. Again, this is because only the root coordinator force writes a switch log record in *i*AP^3 and can accurately determine the status of the transaction.

The third point occurs when the cascaded coordinator of a transaction times out while waiting for the ACKs of a final decision. As the cascaded coordinator needs these ACKs in order to complete the protocol and to forget the transaction, it re-submits the decision to the appropriate participants once communication failures are fixed. In *i*AP^3, a coordinator re-submits a commit decision to each inaccessible 1PC participant, a pre-existing PrA and 2PC *i*AP^3 participant when PrA is used with *i*AP^3 participants. For an abort decision, the coordinator re-submits the decision to each inaccessible *i*AP^3 participant when PrC is used 2PC *i*AP^3 participants. When a participant receives a decision, it complies with the decision, if it has not done so before the failure, and then acknowledges the decision.

4.2 SITE FAILURES

4.2.1 A ROOT COORDINATOR SITE FAILURES

During the initial scan of the log after a site failure, the coordinator re-builds its protocol table and identifies each incomplete transaction. If the coordinator is a pre-existing PrA coordinator, it will correctly handle *i*AP^3 participants using its own failure recovery mechanisms. This is because each *i*AP^3 participant knows, using its own PAC list, that the recovering coordinator is a pre-existing PrA coordinator. Based on that, each *i*AP^3 participant will deal with the recovering coordinator as a PrA participant. On the other hand, for a recovering *i*AP^3 coordinator, the coordinator needs to consider the following types of transactions during its failure recovery:

- Transactions with only switch records: the coordinator knows that PrC was used with 2PC *i*AP^3 participants as only PrC uses this type of records. The coordinator also knows that the commit processing for each one of these transactions was interrupted before the decision was propagated to the participants. Based on that, the coordinator aborts the transaction and sends an abort message to each 2PC *i*AP^3 participant recorded in the switch record. Then, the coordinator waits for an ACK from each one of them. For the *i*AP^3 participants that did not request a protocol switch during the execution of the transaction, they will inquire about the transaction status after the coordinator has recovered. When the coordinator receives an inquiry message that does not include a flag that determines the used protocol after it has received the required ACK messages and forgotten the transaction, the coordinator will assume that the protocol is a presumed-abort based protocol. Based on that, it will correctly reply with an abort decision that is consistent with the presumption of the protocol used by the participant.

- Transactions with switch records and corresponding commit records but without end records: the coordinator knows that PrC was used with the 2PC *i*AP^3 participants of each one of these transactions. However, the coordinator cannot be sure whether all the participants in the execution of each transaction are 2PC or not. For this reason, the coordinator refers to each transaction's switch record to find out this piece of information. If all participants are 2PC, the transaction is considered completed transaction. Otherwise, the coordinator identifies the set of 1PC participants and sends to them commit messages. Then, it waits for their ACKs.

- Transactions with only commit records: the coordinator knows that the protocol used with each one of these transactions has to be a presumed-abort based protocol. This is because PrC requires a switch log record before the commit decision can be made and written onto the log. Based on that, the coordinator knows that either PrA or IYV was used with the transaction. In either case, the coordinator re-sends its commit decision to all the participants of each transaction and waits for their ACKs.

When a participant receives a decision message from a coordinator after a failure, it means that the coordinator needs an ACK. If the participant had been left blocked awaiting the decision, it enforces the received decision and then acknowledges it. Otherwise, it simply replies with an ACK.

The other types of transactions recorded in the coordinator's log can be safely considered completed transactions and ignored during the recovery procedure of the coordinator. If a participant inquires about a transaction that is not within the coordinator's protocol table after a failure, the coordinator responds with a decision that matches the presumption of the protocol indicated in the inquiry message. If the inquiry message does not include any indication about the used protocol, it has to be from an IYV or a pre-existing PrA participant. In this case, the coordinator responds with an abort message.

4.2.2 A LEAF PARTICIPANT SITE FAILURES

For an *i*AP^3 participant, the participant checks its stably stored RCL upon its recovery from a site failure. If the list is empty, it means that the participant can recover its state using its own log and without communicating with any coordinator in the system. Otherwise, it means that there may be some missing records from the participant's log. According to IYV, these records are replicated at the coordinators' logs. To retrieve these missing records, the participant needs to determine the largest *log sequence number* (LSN). This number is associated with the last record written onto the log and survived the failure. Once the largest LSN is determined, the participant sends a *recovering* message that includes the largest LSN to all *i*AP^3 coordinators recorded in the RCL.

When a coordinator receives a recovering message, it uses the LSN included in the message when identifying the missing redo log records from the participant's log.

While waiting for the reply messages to arrive, the participant initiates the *undo phase* of its recovery procedure and when completed, the *redo phase*. This is accomplished using its own local log. That is, the effects of completed transactions, both committed and aborted, are replayed locally while waiting for the reply messages to arrive. This is because of the use of *write-ahead logging* (WAL).

When an *i*AP^3 coordinator receives a recovering message from a participant, the coordinator checks its protocol table. The coordinator needs to determine each transaction that the failed participant has executed some of its operations and the transaction is either still active in the system or has terminated but did not finish the protocol. The former means that the transaction is still executing at other sites and no decision has been made about its final status, yet; while the latter means that a final decision has been made about the transaction but the participant was not aware of the decision prior to its failure. For each forward recoverable transaction, the coordinator includes the list of the redo log records that are stored in its log and have LSNs greater than the one received in the recovering message in its response. For each forward recoverable transaction, the coordinator also includes all the read-locks received from the participant during the execution of the transaction. On the other hand, for a committed transaction, the coordinator responds with a commit status along with the list of all the transaction's redo records that are stored in its log and have LSNs greater than the one that was included in the recovering message of the participant.

The coordinator sends all the above responses in a single *repair* message to the participant. If a coordinator has no active transactions at the participant's site before the participant's failure, the coordinator responds with an *empty* repair message. This latter reply message indicates that there is no extra information available at the responding coordinator beyond the information that is already available at the participant's site and can be used for the recovery of the participant.

When the participant receives the reply messages, it repairs its log and lock table, and then completes the redo phase. During the recovery procedure of an *i*AP^3 participant, the participant also needs to resolve the states of any prepared-to-commit 2PC transactions that were coordinated by either *i*AP^3 coordinators or pre-existing PrA coordinators. A failed participant accomplishes this by identifying such transactions during the analysis phase of the recovery procedure. For each one of these transactions, the participant inquires its direct ancestor in the transaction tree about the final status of the transaction, indicating the used protocol with the transaction as recorded in the prepared log record. If the coordinator of a transaction is a pre-existing PrA, the participant inquires its direct ancestor without making any indication about the used protocol (following PrA protocol).

When an ancestor receives an inquiry message regarding the status of a transaction, it replies with the decision that it still remembers. If the ancestor does not remember the transaction, it uses the indicated protocol in the inquiry message to guide it in its response. The response of the ancestor is abort if the indicated protocol is PrA. The response is also abort if the message does not indicate the used protocol with the transaction. On the other hand, if the indicated protocol is PrC, the ancestor propagates the inquiry message to its own direct ancestor, and so on. This process continues until one of the ancestors still remembers the transaction and responds with the decision that it still remembers or the message reaches the root coordinator which is the only one that can make a correct presumption about unremembered PrC transactions.

For a PrA leaf participant, the participant follows the recovery procedure of PrA protocol. In this case, the transaction has to be PrA across all participants in the transaction execution tree and the participant will receive the correct decision from its direct ancestor.

4.2.3 A CASCADED COORDINATOR SITE FAILURES

During failure recovery after a site failure, being an intermediate site, a cascaded coordinator has to synchronise its recovery with its direct ancestors, from one side, and its direct descendants, from the other.

As a descendant, a cascaded coordinator checks its stably stored RCL. If the list is empty, it means that there were no *i*AP^3 coordinators with active transactions at the cascaded coordinator's site before the cascaded coordinator's failure. In this case, the cascaded coordinator does not communicate with any *i*AP^3 coordinator for recovery purposes. This is because all the necessary information needed for recovery is available locally in its own log. On the other hand, if the RCL is not empty, it means that there may be some missing records from the cascaded coordinator's log. According to IYV, these records are replicated at the coordinators' logs. To retrieve these missing records, the participant needs to determine the largest LSN. Then, the cascaded coordinator sends a *recovering* message that contains the largest LSN to all *i*AP^3 coordinators recorded in the RCL. This LSN is used by *i*AP^3 coordinators to determine missing redo log records at the cascaded coordinator which are replicated in their logs and are needed by the cascaded coordinator to fully recover.

When an *i*AP^3 coordinator receives a recovering message from a cascaded coordinator, it means that the cascaded coordinator has failed and is recovering from a failure. In this case, the coordinator needs to determine each transaction that the failed cascaded coordinator has executed some of its operations and the transaction is either still active in the system or has terminated but did not finish the protocol. The former means that the transaction is still executing at other sites and no decision has been made about its final status, yet; while the latter means that a final decision has been made about the transaction but the cascaded coordinator was not aware of the decision prior to its failure. For each forward recoverable transaction, the coordinator includes the list of the redo log records that are stored in its log and have LSNs greater than the one received in the recovering message in its response. For each forward recoverable transaction, the coordinator also includes all the read-locks received from the participant during the execution of the transaction. On the other hand, for a committed transaction, the coordinator responds with a commit status along with the list of all the transaction's redo records that are stored in its log and have LSNs greater than the one that was included in the message of the cascaded coordinator.

The coordinator sends all the above responses in a single *repair* message to the cascaded coordinator. If a coordinator has no active transactions at the cascaded coordinator's site before the cascaded coordinator's failure, the coordinator responds with an *empty* repair message. This latter reply message indicates that there is no extra information available at the responding coordinator beyond the information that is already available at the cascaded coordinator's site and can be used for the recovery of the cascaded coordinator.

During the recovery procedure of the cascaded coordinator, the cascaded coordinator also needs to resolve the states of any prepared-to-commit 2PC transactions that were coordinated by either *i*AP^3 coordinators or pre-existing PrA coordinators. A failed cascaded coordinator accomplishes this by identifying such transactions during the analysis phase of the recovery procedure. For each one of these transactions, the cascaded coordinator inquires its direct ancestor in the transaction tree about the final status of the transaction, indicating the used protocol with the transaction as recorded in the prepared log record. If the coordinator of a transaction is a pre-existing PrA, the

cascaded coordinator inquires its direct ancestor without making any indication about the used protocol (following PrA protocol).

When an ancestor receives an inquiry message regarding the status of a transaction, it replies with the decision that it still remembers. If the ancestor does not remember the transaction, it uses the indicated protocol in the inquiry message to guide it in its response. The response of the ancestor is abort if the indicated protocol is PrA. The response is also abort if the message does not indicate the used protocol with the transaction. On the other hand, if the indicated protocol is PrC, the ancestor propagates the inquiry message to its own direct ancestor, and so on. This process continues until one of the ancestors still remembers the transaction and responds with the decision that it still remembers or the message reaches the root coordinator which is the only one that can make a correct presumption about unremembered PrC transactions.

While waiting for the reply messages to arrive, the cascaded coordinator initiates the *undo phase* of its recovery procedure and when completed, the *redo phase*. This is accomplished using its own local log. That is, the effects of completed transactions, both committed and aborted, are replayed locally while waiting for the reply messages to arrive. This is because of the use of *write-ahead logging* (WAL).

When the cascaded coordinator receives the required reply messages from the *i*AP^3 coordinators recorded in its RCL, the cascaded coordinator repairs its log and lock table, and then completes the redo phase. During the recovery procedure of a cascaded coordinator, the cascaded coordinator also needs to resolve the states of any prepared-to-commit 2PC transactions that were coordinated by either *i*AP^3 coordinators or pre-existing PrA coordinators. A failed cascaded coordinator accomplishes this by identifying such transactions during the analysis phase of the recovery procedure. For each one of these transactions, the cascaded coordinator inquires its direct ancestor in the transaction tree about the final status of the transaction, indicating the used protocol with the transaction as recorded in the prepared log record. If the coordinator of a transaction is a pre-existing PrA, the cascaded coordinator inquires its direct ancestor without making any indication about the used protocol (following PrA protocol).

As an ancestor, the cascaded coordinator needs to finish commit processing for each prepared to commit transaction that was interrupted due to the failure without finalizing its commit protocol with the direct descendants. This is accomplished by following the decided protocol recorded in the prepared record of each transaction as during normal processing.

5. CONCLUSIONS

For a practicality reason, any newly proposed ACP has to be extended to the multi-level distributed transaction execution model as it is the one currently adopted by the database standards. Not only that, but it is considered the *de facto* model in the database systems' industry. As the *intelligent adaptive participant's presumption protocol* (*i*AP^3) exhibits a highly appealing efficiency and applicability characteristics, this article concentrated on the details of extending it to the more general multi-level distributed transaction execution model. The extension of the *i*AP^3 includes extending its advanced features and not only the basic ones. We believe that this work should help in the design of any future practical atomic commit protocols.

REFERENCES

[1] Gray, J. "Notes on Database Operating Systems", in Bayer, R., Graham, R. M. & Seegmuller, G. (Eds.): Operating Systems: An Advanced Course, LNCS, Vol. 60, Springer, 1979.

[2] Lampson, B. "Atomic Transactions", in Lampson, B., Paul, M. & Siegert, H.J. (Eds.): Distributed Systems: Architecture and Implementation - An Advanced Course, LNCS, Vol. 105, Springer, 1981.

[3] Al-Houmaily, Y. & Samaras, G. "Two-Phase Commit", in Liu, L. & Tamer Özsu, M. (Eds.): Encyclopedia of Database Systems, Springer, 2009.
[4] Al-Houmaily, Y. "Atomic Commit Protocols, their Integration, and their Optimisations in Distributed Database Systems", Int'l J. of Intelligent Info. and Database Sys., Vol. 4, No. 4, pp. 373-412, 2010.
[5] Stamos, J. & Cristian, F. "Coordinator Log Transaction Execution Protocol", Distributed and Parallel Databases, Vol. 1, No. 4, pp. 383-408, 1993.
[6] Al-Houmaily, Y. & Chrysanthis, P. "An Atomic Commit Protocol for Gigabit-Networked Distributed Database Systems", J. of Systems Architecture, Vol. 46, pp. 809-833, 2000.
[7] Abdallah, M., Guerraoui, R. & Pucheral, P. "Dictatorial Transaction Processing: Atomic Commitment without Veto Right", Distributed and Parallel Databases, Vol. 11, No. 3, pp. 239-268, 2002.
[8] ISO. "Information Technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation)", ISO/IEC 9075-2, 2008.
[9] Al-Houmaily, Y. "On Deferred Constraints in Distributed Database Systems", Int'l Journal of Database Management Systems, Vol. 5, No. 6, December 2013.
[10] Al-Houmaily, Y. "GLAP: A Global Loopback Anomaly Prevention Mechanism for Multi-Level Distributed Transactions", Int'l Journal of Database Management Systems, Vol. 6, No. 3, June 2014.
[11] Al-Houmaily, Y. "On Interoperating Incompatible Atomic Commit Protocols in Distributed Databases", Proc. of the 1st IEEE Int'l Conf. on Computers, Comm., and Signal Processing, 2005.
[12] Mohan, C., Lindsay B. & Obermarck, R. "Transaction Management in the R* Distributed Data Base Management System", ACM TODS, Vol. 11, No. 4, pp. 378-396, 1986.
[13] Al-Houmaily, Y. "An Intelligent Adaptive Participant's Presumption Protocol for Atomic Commitment in Distributed Databases", Int'l J. of Intel. Info. and Database Sys., Vol. 7, No. 3, 2013.
[14] Al-Houmaily, Y., Chrysanthis, P. & Levitan, S. "An Argument in Favor of the Presumed Commit Protocol", Proc. of the 13th ICDE, 1997.
[15] Gray, J. & Reuter, A. "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Inc., USA, 1993.
[16] Al-Houmaily, Y. & Chrysanthis, P. "Atomicity with Incompatible Presumptions", Proc. of the 18th ACM PODS, 1999.

AUTHOR

Yousef J. Al-Houmaily received his BSc in Computer Engineering from King Saud University, Saudi Arabia in 1986, MSc in Computer Science from George Washington University, Washington DC in 1990, and PhD in Computer Engineering from the University of Pittsburgh in 1997. Currently, he is an Associate Professor in the Department of Computer and Information Programs at the Institute of Public Administration, Riyadh, Saudi Arabia. His current research interests are in the areas of database management systems, mobile distributed computing systems and sensor networks.