GLAP: A GLOBAL LOOPBACK ANOMALY PREVENTION MECHANISM FOR MULTI-LEVEL DISTRIBUTED TRANSACTIONS

Yousef J. Al-Houmaily

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

ABSTRACT

The multi-level/hierarchical distributed transaction execution model is currently the model specified in the database standards and practiced in the implementations of commercial database management systems. In this model, a transaction may execute more than one subtransaction with different origins at a participating site, causing the same transaction to appear more than once at the participating site. This system state is well recognized in the literature and is commonly known as a "loopback". When a loopback occurs at a site, certain types of anomalies may arise. The effects of these anomalies on the system behaviour vary depending on the type of the anomaly. In an extreme case, a loopback anomaly may lead to non-serializable executions of transactions, sacrificing the consistency of the entire distributed database system. Thus, it is imperative to characterize the different types of loopback anomalies and to provide a practical and efficient solution to those that are most devastating on the system behaviour. This is the focus of this article.

KEYWORDS

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

1. INTRODUCTION

In the multi-level/hierarchical distributed transaction execution model, each transaction is decomposed, depending on its own data access requirements, into a number of subtransactions executing at different database sites. When a subtransaction executes at a participating site, the site may decompose its assigned subtransaction further into a number of subtrasnactions and launches them to execute at other participating sites. Thus, the execution of each distributed transaction can be represented by a tree hierarchy with a root node, representing the originating site of the transaction, and a number of intermediate and leaf nodes, representing the different participating sites.

When a transaction executes only one subtransaction at each participating site, the resulting execution hierarchy can be represented by a spanning tree. However, in general, the resulting execution tree of a transaction may contain cycles. The presence of a cycle indicates that the transaction is executing more than one subtransaction at a site, each of which with a different originating site. This latter execution behaviour of multi-level/hierarchical transactions is well recognized in the literature and is commonly called a *loopback* [1, 2].

Regardless of whether the execution tree of a transaction is a spanning tree or not, when the transaction finishes its execution across all participating sites, the root node initiates an *atomic commit protocol* (ACP) such as the basic *two-phase commit* (2PC) [3, 4] or one of its variants [5, 6]. This is to ensure a consistent termination state for the transaction (i.e., to either commit or to abort) across all participating sites.

In the absence of loopbacks, transaction management is relatively simpler than in their presence. This is because each system component within a participating site can uniquely identify the transaction without any possible ambiguity. In contrast, this is not generally the case in the presence of loopbacks. Thus, loopbacks may introduce certain types of anomalies that negatively affect the system behaviour such as causing a transaction to deadlock with itself at a participating site or causing an ACP violation that prohibits the transaction from reaching a final termination state.

A Loopback may also introduce an *execution infection*, a devastating system behaviour which may result in a non-serializable execution of transactions. An execution infection may occur whenever a transaction executes operations that are associated with deferred consistency constraints or deferred triggers and, at the same time, the system implements an ACP optimization designed around the early release of read locks [7, 8]. Unlike other types of loopback anomalies, this latter anomaly cannot be captured by each site independently and requires a global, inter-site synchronization mechanism for its prevention.

As loopback anomalies are imminent in the currently adopted transactional execution model, there is an absolute necessity to characterize the different types of loopback anomalies and to discuss their currently available solutions. Besides that, there is a need to provide an effective and efficient solution to those types of anomalies that require global, inter-site coordination among the participating sites for their prevention.

The rest of this paper is structured as follows: Section 2 describes the details of the adopted distributed transaction execution model and some background material. Section 3 discusses the different types of loopback anomalies. Section 4 characterizes the different types of loopback anomalies into *local* loopback anomalies and *global* loopback anomalies. This characterization is based on whether a loopback anomaly requires only *static*, *locally* available transaction management control information for its prevention or also requires *dynamic*, *remotely* available transaction management control information. Section 5 proposes GLAP, a mechanism designed for the prevention of global loopback anomalies. Section 6 discusses two closely related works that were proposed to tackle execution infections while Section 7 summaries the paper and includes some concluding remarks.

2. SYSTEM MODEL AND BACKGROUND

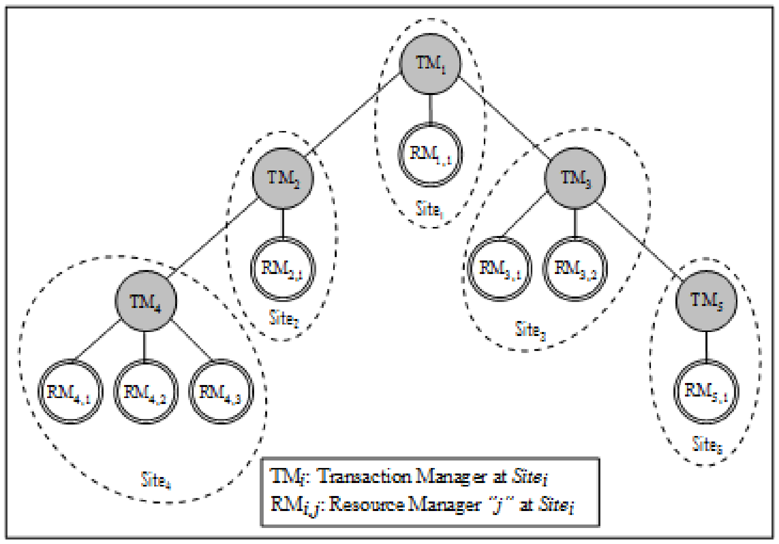
A distributed transaction represents a single logical unit of work that accesses data stored at different database sites interconnected via a communication network. Each database site consists of a *transaction manager* (TM), one or more *resource managers* (RMs), and a *communication manager* (CM).

A TM at a site is responsible for different management and coordination activities. These include assigning identifiers to transactions, monitoring their progress, supervising their completion, and coordinating failure recovery.

RMs are responsible about performing the actual work on data. A RM could be any system that manages shared data and maintains (local) data consistency even in the case of failures. In this article, a RM is assumed to be a relational database management system although it could be any other transactional system such as a transactional queue or a transactional file system.

A CM at a site facilitates inter-site data communication by incorporating a communication protocol and providing a local view of remote sites. It allows locally executing transactions, depending on their own data access requirements, to extend their execution to remote sites. A CM at a site also allows the local TM to perform its transaction management and coordination responsibilities in collaboration with remote TMs.

Figure 1 A typical multi-level/hierarchical transaction execution tree



2.1. Execution of Distributed Transactions

Once a distributed transaction starts executing at a site, it needs to *register* itself with its local TM before it can submit any work to any RM. When the TM (at the *originating* site of a transaction) registers a transaction, it assigns the transaction a system-wide unique identification number (*TxID*). The TM, at the originating site of the transaction, also informs all local RMs about the start of the transaction. In this way, the local TM and all local RMs are aware of the existence of the transaction. Alternatively, a RM could have the ability to *dynamically* register itself, as a participant in the execution of the transaction, with its local TM once it receives some database operation(s) for execution from the transaction [9]. In this latter case, there is no need for the TM to inform such a RM about the start of the transaction once it begins executing at the site.

After registering itself with the local TM, a transaction can execute at any of the local RMs in accordance to the *request/response* processing paradigm. That is, when a transaction submits work for execution to a RM, it waits until the work is executed and acknowledged before it submits any new work to any RM. Depending on the data access requirements of the transaction, it may also initiate work at remote sites. In this latter case, the local TM has to be aware, in advance, of each new work to be initiated at any remote site.

Once the local TM becomes aware of the new remote work to be initiated, it creates a new *execution branch* with a unique *BranchID* for the transaction at the remote site. Then, the new work, using the newly created branch, is submitted through the CM for execution to the remote site. Similarly, the newly initiated work, after registering with the TM at the remote site on behalf of the transaction, may initiate other new works at other remote sites, and so on. Thus, a transaction execution *tree*, similar to the one depicted in Figure 1, is formed for each distributed transaction whereby the TM at each participating site is aware of all locally controlled RMs that participate in the execution of the transaction as well as all neighbouring remote TMs.

After a transaction finishes its execution and indicates its termination point to its local TM, through a *commit* primitive, the TM initiates an ACP such as 2PC or one of its commonly known variants. This is to synchronize the termination of the transaction, in a consistent manner (i.e., to either commit or abort the transaction), across all involved participants.

2.2. The Two-Phase Commit Protocol

In 2PC, each transaction is associated with a designated TM called the *coordinator* which is usually the TM at the site where the transaction is first initiated. Each of the other participants in the execution of the transaction is either an intermediate node representing a remote TM or a leaf participant representing a RM. If the participant is a remote TM, it is called *cascaded coordinator* because it acts as a participant with respect to its direct ancestor, in the transaction execution tree, and a coordinator with respect to its direct descendant(s). For example, in Figure 1, TM_1 is the coordinator of the transaction tree whereas the rest of the TMs are cascaded coordinators. In the figure, TM_2 is an example to a cascaded coordinator which acts as a participant with respect to (its direct ancestor) TM_1, and a coordinator with respect to (its direct descendants) TM_4 and $RM_{2,1}$.

At commit point of a transaction (i.e., when the transaction issues its final commit request), 2PC synchronizes for a consistent final termination state for the transaction across all participants though two consecutive phases. Each of these two phases consists of one round-trip of messages and the first phase is called the *voting* phase; whereas, the second phase is called the *decision* phase. The purpose of the first phase is to gather the votes of all participants, for decision making, at the coordinating TM; whereas, the purpose of the second phase is to propagate the final decision to the participants and to gather their acknowledgments, allowing the coordinating TM to forget fully acknowledged transactions. These two phases, in 2PC, guarantee that (1) a transaction is committed only if all participants agree to commit the transaction, and (2) all participants receive the (same) final decision.

To ensure resilience to failures, each participant is required to write specific log records at certain points during the progress of the protocol. These log records guarantee that, regardless of the frequency and the number of site and communication failures, the protocol will deterministically reach its end. In 2PC, all log records, except for the *end* records that indicate the termination of the protocol at the coordinator and cascaded coordinators, are written in a *forced* manner.

A forced log write of a log record means that the information contained in the log record is essential for the recovery purposes of the protocol after a system failure. Consequently, the log record has to be stored onto a stable storage medium that can sustain system failures before the protocol can proceed in its progress. On the other hand, a non-forced log write of a log record means that the information contained in the log record can be reacquired, when necessary, as part of the recovery procedure of the protocol after a system failure. Hence, it suffices to store the log record in the log buffer (in main memory) without having to propagate it to a stable storage medium. Thus, a forced log write is much more costly, from performance point of view on the protocol, than a non-forced log write. This is because the former type of log writes suspend the protocol for a considerable amount of time until the access to the stable storage medium is completed; whereas, the latter type of log writes does not cause the protocol to be suspended for such a considerable time.

Figure 2 The coordination messages and log writes in the multi-level/hierarchical two-phase commit protocol.

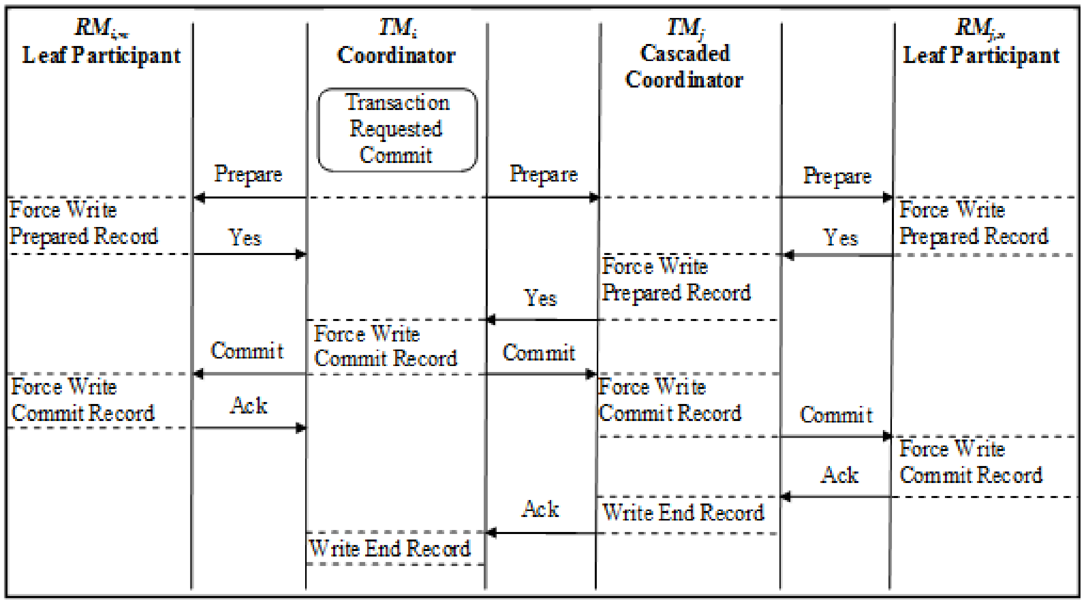

$RM_{i,w}$
Leaf Participant
TM_i
Coordinator
TM_j
Cascaded Coordinator
$RM_{j,x}$
Leaf Participant
Transaction Requested Commit
Prepare
Prepare
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Force Write Commit Record
Commit
Commit
Force Write Commit Record
Ack
Force Write Commit Record
Commit
Force Write Commit Record
Ack
Ack
Write End Record
Write End Record

Figure 2 depicts the sequence of coordination messages and log writes, encountered in 2PC, for a transaction executing at two sites: $Site_i$ and $Site_j$. In the figure, TM_i is the coordinator of the transaction whereas TM_j is a cascaded coordinator. $RM_{i,w}$ and $RM_{j,x}$ are the two resource managers that performed actual work on data for the transaction before it had reached its termination point and the 2PC was initiated.

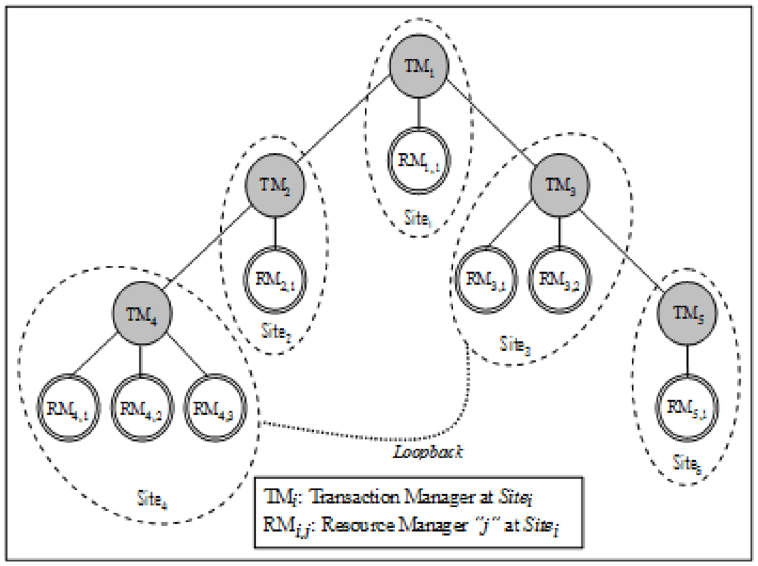
2.2.1. The Read-Only Optimization

As the *number* and the *sequential* nature of messages and forced log writes in any ACP consume a substantial amount of each transaction execution time, there is a continuing research interest in developing ACPs and optimizations around reducing these metrics. Most notable type of optimizations currently available and widely implemented in commercial database management systems is the type in which an optimization is designed around the early release of read locks. In such an optimization, the read locks held by a transaction are released at certain participating sites before the transaction has fully terminated across all participants. One such optimization is the read-only optimization [5]. This optimization is currently part of the database standards [9] and is adopted by the majority of commercial database management systems.

The read-only optimization significantly reduces the cost of 2PC for read-only transactions. This is because any *exclusively* read-only participant, a participant that has not performed any updates to data at its site on behalf of a transaction, is excluded from the decision phase of the transaction. More specifically, when a read-only participant receives a vote-request (i.e., prepare) message, it simply releases all the locks held by the transaction and responds with a "read-only" vote (instead of a "yes" vote which indicates its readiness to commit the transaction). This "read-only" vote means that the transaction has read consistent data and the participant does not need to be involved in the second phase of the protocol because it does not matter whether the transaction is finally committed or aborted. Consequently, this optimization allows each read-only participant to terminate the transaction and to release all the resources held by transaction, including the read-locks, earlier than the update participants and without having to write any log records. This represents the essence of the traditional read-only optimization [5].

Given that the majority of executing transactions in any general database management system are read-only, the significance of the read-only optimization is much more pronounced with multi-level transactions. This is because, in multi-level transactions, entire transaction execution branches could be read-only. Consequently, these read-only branches are excluded from the second phase of 2PC, a significant enhancement to the commit processing cost of transactions.

Figure 3 A transaction execution tree with a loopback



3. LOOPBACKS

In the absence of loopbacks, all work submitted by a transaction to a participating site originates from one ancestor site in the transaction execution tree. This work is executed at the participating site as a single logical unit of work commonly called a *subtransaction*. Hence, in the absence of loopbacks, each distributed transaction executes one subtransaction per each participating site.

A loopback occurs whenever the work submitted by a transaction to a participating site originates from more than one site in the transaction execution tree. This makes the same transaction appears as more than one subtransaction at the executing site. Figure 3 depicts an example to a transaction execution tree with a loopback. In the figure, *$Site_3$* is executing work for the transaction and some of the work is originating from *$Site_1$* while the rest of the work is originating from *$Site_4$*. This makes the transaction appears as two subtransactions at *$Site_3$*: one originating from *$Site_1$* and another originating from *$Site_4$*.

3.1. Loopback Complications

In the absence of loopbacks in a transaction execution tree, all system components at a participating site (i.e., the TM, the CM and each RM) can uniquely identify the transaction and its originating site without any possible ambiguity. Not only that, but when a site sends its vote to its direct ancestor, during commit processing, it means that the site will never receive any further work for the same transaction from any other site. This is not the case in the presence of loopbacks as: (1) a system component at a site may be unable to uniquely identify each of the subtransactions pertaining to the same transaction or its originating site, or (2) a participating site may receive further work for a transaction after the site has already voted. Hence, loopbacks complicate transaction management, requiring special attention. Otherwise, unpredictable system behaviour may arise and, possibly, a destructive one that jeopardizes the consistency of the entire distributed database system. Thus, there is a need to clearly characterize the different types of possible loopback anomalies and to shed light on their currently available solutions. Besides that, there is a need to provide a practical and an efficient solution for those loopback anomalies that are currently left unhandled in the database research literature.

3.2. Types of Possible Loopback Anomalies

Loopback anomalies could be classified along the lines of three types as explained in the next three sections.

3.2.1. Deadlock Anomaly

This type of anomalies occur whenever one RM (at a site) handles the different subtransactions, pertaining to the same transaction, *independently*. In this case, the subtransactions may conflict with each other over their access to data causing the transaction to deadlock with itself at the RM. This occurs when one subtransaction holds data required by another subtransaction and the holding subtransaction cannot release its held data for the other subtransaction until commit time of the (whole) transaction (i.e., when all subtransactions are executed across all participating sites). The following example demonstrates this type of anomalies.

Example 1: In Figure 3, if $RM_{3,1}$ at $Site_3$ executes the work submitted by $Site_1$ and the work submitted by $Site_4$ as two independent subtransactions and the two subtransactions conflict over their access to data, one of them has to wait, indefinitely, for the termination of the other. This is because the subtransaction that holds the data cannot be terminated until the termination of the (whole) transaction (including the waiting subtransaction) and the waiting subtransaction cannot access its required data until the holding subtransaction terminates.

3.2.2. ACP Violation Anomaly

This type of anomalies may arise in two different forms: (1) whenever a cascaded coordinator (at a site) does not recognize all its TM ancestors in the transaction tree during commit processing whereby it interacts with some TM ancestors and does not interact with the other TM ancestors, and (2) whenever a participant (whether it is a cascaded coordinator or a leaf participant) receives new work for a transaction after the transaction has been prepared by the participant.

The following two examples respectively explain these two forms of ACP violation anomalies.

Example 2: As all work executing on behalf of a transaction at a site represents a single logical unit of work regardless of the number of subtransactions performing the work at the site, if a cascaded TM does not recognize all its direct TM ancestors during commit processing, an ACP violation occurs. This is because some of the TM ancestors are left dangling. For example, in Figure 3, if $Site_3$ considers only $Site_1$ during commit processing and ignores $Site_4$, $Site_4$ is left dangling, resulting in an ACP violation.

Example 3: After a transaction has submitted all its work to all required sites and the root coordinator initiated commit processing, a site may prepare the transaction and sends a “yes” vote and later on receives a new work for the same transaction to validate a *deferred consistency constraint* or to execute a *commit–time trigger*. Validating deferred consistency constraints and executing commit-time triggers represent works performed at commit time of a transaction (i.e., when a site receives a prepare to commit message for the transaction during commit processing). Deferred consistency constraints are currently part of the SQL (Structured Query Language) standards [10] but commit-time triggers (i.e., deferred triggers) are not although some implementations support them (e.g., PostgreSQL [11]).

For example, in Figure 3, if *Site$_1$* submits work to *Site$_3$* during the execution phase of the transaction while *Site$_4$* submits work to *Site$_3$* during the commit processing phase of the transaction (i.e., when *Site$_4$* receives the prepare message) to validate a deferred consistency constraint or to execute a commit-time trigger at *Site$_3$*, it is possible for *Site$_3$* to respond with a "yes" vote to the vote-request message of *Site$_1$*, assuming that the work submitted by *Site$_1$* can be committed at *Site$_3$*. After that, when *Site$_3$* receives the work from *Site$_4$*, for the same transaction, to validate the deferred constraint or to execute the commit-time trigger, an ACP violation at *Site$_3$* occurs. This is because *Site$_3$*, according to 2PC and its commonly known variants, should never receive any new work for the transaction after it has already voted.

3.2.3. Execution Infection Anomaly

This type of anomalies may occur in the presence of deferred work that needs to be performed at commit time of a transaction and, at the same time, the transaction is read-only at some participating sites. The following example demonstrates this type of anomalies.

Example 4: Assume that *Site$_1$* has submitted *read-only* work for the transaction T_1, shown in Figure 3, to be executed at *Site$_3$*. Furthermore, assume that *Site$_4$* has executed work for the same transaction and the work requires the validation of a deferred consistency constraint at *Site$_3$*. At commit time of T_1, *Site$_3$* receives a prepare message from *Site$_1$*, but *Site$_4$* does not receive its prepare message until some later time. When *Site$_3$* receives the prepare message from *Site$_1$*, it responds with a "read-only" vote. Then, according to the read-only optimization, *Site$_3$* releases the data held by T_1 and forgets the transaction. Meanwhile, a subtransaction pertaining to another transaction T_2 starts executing at *Site$_3$* and modifies the data released by T_1 as well as the data associated with the consistency constraint that is supposed to be checked, at commit time, by the work of T_1 that executed at *Site$_4$*. Then, T_2 commits at *Site$_3$*. After that, the delayed prepare message of T_1 arrives at *Site$_4$*. At that time, *Site$_4$* launches a new work to check the data associated with the deferred consistency constraint at *Site$_3$*. *Site$_3$* executes the new work and affirmatively acknowledges the completion of the work. Then, *Site$_3$* receives a prepare message from *Site$_4$* and responds with a "yes" vote. Commit processing continues until T_1 is committed across all sites, including *Site$_3$*.

The result of the above scenario represents an example to a non-serializable global execution of transactions with the following cyclic serialization order: ($T_1 \rightarrow T_2 \rightarrow T_1$). This anomaly could occur even if each RM at each site deploys *strict two-phase locking* (S2PL) for concurrency control, the one that guarantees serializability in distributed database systems and the *de facto* concurrency control mechanism in the industry. The reason behind that is due to the read-only optimization that is implemented as part of the ACP.

A similar scenario could also occur even if the deferred consistency constraint to be validated is located within the same site that executed the work associated with the deferred constraint (i.e., *Site$_4$* in our example). Not only that but deferred triggers are also a source to similar scenarios when the adopted ACP implements an optimization designed around the early release of read locks. In fact, any optimization that is designed around the early release of read locks of transactions at certain participants before transactions are fully terminated across all participants is a source of possible transaction execution infections [7, 8].

4. HANDLING LOOPBACK ANOMALIES

A loopback anomaly could be characterized as either a *local* loopback anomaly or a *global* loopback anomaly. This characterization depends on whether the loopback anomaly requires only *static*, *locally* available transaction management control information for its prevention or also requires *dynamic*, *remotely* available transaction management control information.

4.1. Handling Local Loopback Anomalies

For a local loopback anomaly, it is sufficient for each system component, within a site, to uniquely identify each transaction and its executing subtransactions within the site besides the origin of each subtransaction. This can be accomplished by utilizing the mechanisms provided by the database standards through specific system calls.

For example, a RM at a site can identify whether any newly initiated subtransaction is part of an ongoing transaction, a transaction that is already executing other subtransactions at the RM, during the (required) registration process of the newly initiated subtransaction with the local TM. If the TM indicates to the RM that the newly initiated subtransaction is part of a previously initiated work for the same transaction, the RM executes the subtransaction within the *scope* (i.e., context) of the previous work of the transaction. In this way, the RM executes the different subtransactions pertaining to the same transaction, including the newly initiated subtransaction, as a single logical unit of work, avoiding any possible deadlock for the transaction with itself at the RM. In the standards, this mode of execution is called *tightly-coupled* [9].

The standards also allow for the execution of the different subtransactions that pertain to the same transaction to execute in an independent manner as if they belong to different transactions. This mode is defined in the standards to meet the requirements of certain application systems and is called *loosely-coupled* [9]. When a transaction uses this latter mode, there is no guarantee for the transaction to execute in a deadlock-free manner when more than one of its subtransactions execute at a RM. Hence, ensuring deadlock-free executions of subtransactions that pertain to the same transaction at a RM, in this mode, is left as part of the application developer responsibilities.

Another example to local loopback anomalies is the one where a TM responds to one of its ancestor TMs and leaving the others dangling. This can be detected and resolved using the unique identification numbers of transactions and the unique identification numbers of their branches. Using these unique numbers, a TM can unambiguously identify all the subtransactions pertaining to the same transaction and their originating sites. Hence, the TM can deal with each subtransaction, during commit processing, independently. This is even in the presence of more than one subtransaction that pertain to the same transaction executing at its site.

4.2. Handling Global Loopback Anomalies

Unlike local loopback anomalies, global loopback anomalies can neither be detected nor prevented using only static, locally available transaction management control information. This is because the execution behaviour of a transaction at one site may change at any time and affect other sites, causing a global loopback anomaly. That is, the presence of this type of anomalies cannot be predicted so long as a transaction is still executing and their presence may not be confined to a specific site but may span more than one. Thus, there is a need for a practical and efficient mechanism that enables for the prevention of global loopback anomalies. This is the essence of the *global loopback anomaly prevention* (GLAP) mechanism which is introduced in the next section.

5. GLOBAL LOOPBACK ANOMALY PREVENTION

GLAP is an inter-site synchronization mechanism designed to prevent global loopback anomalies. It is based on the *piggybacking* of transaction management control information, in a dynamic manner, among the participating sites during the execution phase of transactions. Specifically, when a site executes work that requires the execution of further work at commit time of a transaction, the site executing the work has to inform the coordinator of the transaction about this transaction execution behaviour. Based on the received information, the coordinator informs all the other participants about this expected transaction execution behaviour. In this way, the other participants can cooperate with the coordinator to prevent any possible global loopback anomaly. This is accomplished by not terminating the transaction at any participating site before the final decision is made by the coordinator and is received by the participants. That is, all resources, including the read-locks, are held at each participant until the transaction is fully terminated across all sites. This includes the deferred work that is expected to cause the global loopback anomaly and regardless of whether the transaction is read-only or not at any given participant.

To inform the coordinator of a transaction about work that requires the initiation of further work at commit time of a transaction, the site executing the work sets a *potential loopback anomaly* (PLA) flag that indicates this expected execution behaviour in the acknowledgment message of the successful execution of the work. When the coordinator receives such an acknowledgment message, the coordinator becomes aware of a possible global loopback anomaly. Based on that, the coordinator informs the other participants about the possibility of the global anomaly. This is accomplished during the first phase of the ACP.

During the voting phase, the coordinator includes a PLA flag in the prepare message that it sends to each of its direct descendents. When a descendent receives the message, it also becomes aware of the possible anomaly and informs its direct descendents (if it has ones) accordingly. In this way, all participants in the transaction execution tree become aware of the possible global anomaly.

When a participant receives a prepare to commit message with a PLA flag, the participant continues to hold all the resources, including the read-locks, on behalf of the transaction until the decision phase. This is even if the participant is an exclusively read-only participant. That is, each participant follows the ACP as if it does not incorporate the read-only optimization within its implemented ACP. Not only that, but each of the other participants should be ready to accept any new work for the same transaction, even after the participant had (affirmatively) voted, and to execute the work within the same run-time scope of the previously executed work for the transaction.

Using GLAP, when the coordinator receives the ("yes") votes of all the participants, the coordinator is guaranteed that all the work pertaining to the transaction has been terminated, across all participants, and no participant has released the resources acquired during the execution of the transaction. Not only that, but the coordinator is also guaranteed that all the work submitted to a participant, regardless of its origin, has been executed within one run-time scope at the participant. Hence, when the coordinator makes the final (commit) decision, it is not possible for the transaction to be involved in any global loopback anomaly.

To demonstrate the effectiveness of the proposed mechanism, reconsider Example 3 (Section 3.2.2) and Example 4 (Section 3.2.3).

In Example 3, when *$Site_4$* executes the work associated with the deferred constraint or the commit-time trigger, using GLAP, it informs the coordinator about the deferred work that it needs to perform at commit time of the transaction. This is accomplished by setting the PLA flag in the acknowledgment message of the successful execution of the work. Based on that, the coordinator becomes aware of a potential global loopback anomaly. Consequently, it informs all the other participants, including *$Site_3$*, during the first phase of commit processing. When *$Site_3$* becomes aware of the potential global loopback anomaly, it will continue to hold all the resources acquired by the transaction even after it has sent a "yes" vote to *$Site_1$*. Not only that, but *$Site_3$* will be ready to accept any new work for the transaction. This is regardless of the origin of the new work that it may receive. When *$Site_3$* receives the commit-time work from *$Site_4$*, it will execute the new work within the same run-time scope of the previously executed work for the transaction. For commit processing, *$Site_3$* will treat the new branch, using the unique *BranchID*, independently from the previous branch of the transaction. Thus, using GLAP, the possible ACP violation, in this example, is eliminated through modifying the ACP such that a participating site does not consider the transaction terminated, in the presence of a possible global loopback anomaly, until it receives the final decision from the coordinator.

In Example 4, when *$Site_4$* executes the work that is associated with the deferred constraint for T_1, using GLAP, *$Site_4$* informs the coordinator about the deferred work that it needs to perform at commit time of the transaction. This is accomplished by setting the PLA flag in the acknowledgment message of the successful execution of the work. Based on that, the coordinator becomes aware of a potential global loopback anomaly. Consequently, it informs all the other participants, including *$Site_3$*, during the first phase of commit processing. When *$Site_3$* becomes aware of the deferred work that is to be executed at *$Site_4$* and its potential global loopback, it will vote "yes" instead of "read-only" and will continue to hold the resources acquired by T_1 including the read locks. Thus, T_2 will be blocked awaiting T_1 to release its held read locks. When *$Site_4$* sends the deferred work to *$Site_3$*, *$Site_3$* will execute the work within the same run-time scope of the previously submitted work by T_1 even though *$Site_3$* has already voted. Thus, T_2 will not be able to interfere with the execution of T_1 causing a non-serializable execution. Hence, using GLAP, the possible execution infection, in this example, is eliminated through modifying the ACP such that a participating site does not use the read-only optimization, in the presence of a possible global loopback anomaly, and executing any new work that it receives within the same run-time scope of the previously submitted work by the same transaction.

6. DISCUSSION

As mentioned in Example 4 (Section 3.2.3), a transaction execution infection could occur even if the deferred consistency constraint to be validated is located within the same site that executed the work associated with the deferred constraint. However, this specific type of transaction execution infections is considered, according to our characterization, as a local loopback anomaly. This is because it could be prevented using only static, locally available transaction management control information. One simple way to prevent this type of execution infection anomalies is to acquire all the locks needed for the validation of a deferred constraint during the execution of the operation that is associated with the consistency constraint and postponing the actual validation of the constraint until commit time of the transaction [8]. In this way, other transactions are prohibited from interfering with the transaction and producing a non-serializable execution. However, this is not the most appropriate mechanism for handling deferred constraints as it forces the system to acquire the locks associated with deferred constraints, unnecessarily, for long periods of time, depriving the system from potential concurrency among executing transactions.

The *unsolicited deferred consistency constraints validation* (UDCCV) [8] and the *timestampped two-phase commit* (T2PC) [7] are two mechanisms that were designed to avoid any release of locks prematurely, prohibiting the occurrence of the above special case of execution infections. Similar to GLAP, UDCCV piggybacks control information in the acknowledgment messages of individual operations to inform the coordinator and, subsequently, the other participants about any potential execution infections. In this way, the coordinator and all the participants are forced to hold all the locks until the commit time of a transaction, prohibiting any possible execution infections. This is accomplished on a per transaction basis and only in the presence of a potential execution infection. However, UDDCV was not designed for the prevention of global loopback anomalies, the ones demonstrated in Example 3 (Section 3.2.2) and Example 4 (Section 3.2.3).

T2PC was also designed for the same purpose as UDCCV but in a rather different way. T2PC is based on *timestamps* as a method for knowing when the participants are terminated from executing a transaction. That is, each participant, in T2PC, sends the *time range* during which a transaction must commit or else aborted along with its vote (and not during the execution phase of the transaction). Thus, the coordinator of a transaction is also responsible about determining the time range that satisfies the requirements of all the participants, if the transaction is to be committed, or else aborting the transaction.

As T2PC incorporates the concept of time, it significantly complicates the simplicity of the 2PC protocol. Not only that, but clock divergence among individual database sites introduces another problem into the protocol. Besides that, and as UDCCV, T2PC was designed in the context of the two-level distributed transaction execution model. Hence, it does not solve the problems of loopback anomalies that exist in the multi-level distributed transaction execution model.

7. SUMMARY AND CONCLUSIONS

When distributed transactions execute according to the multi-level/hierarchical transaction execution model, a transaction may appear at a participating site more than once. This is called a “loopback”. Loopbacks require special database management attention. Otherwise, certain anomalies may occur causing unexpected system behaviour, including non-serializable executions of transactions. This motivated the characterization of loopback anomalies into two types: *local* loopback anomalies and *global* loopback anomalies. The characterization of both types is based on the needed transaction management control information for their prevention: *static* and *local* versus *dynamic* and *remote*. Whereas local loopback anomalies can be prevented using the currently available mechanisms provided by the database standards, global loopback anomalies cannot, requiring new mechanisms for their detection or their prevention. This led to the design of the GLAP (Global Loopback Anomaly Prevention) mechanism.

GLAP is based on the *piggybacking* of transaction management control information among participating sites, in a dynamic manner, as transactions progress in their executions. That is, GLAP exposes the execution behaviour of a transaction at a participating site that may cause a global loopback anomaly to all the other participating sites. Hence, allowing for their cooperation to effectively and efficiently prevent any possible global loopback anomaly

The GLAP mechanism shows that *piggybacking* of transaction management control information can be used not only for the design of highly efficient ACPs, as it is customary the case, but also for solving some subtle transaction management problems that cannot be captured locally at each individual database site.

REFERENCES

[1] C. Mohan, K. Britton, A. Citron, and G. Samaras, “Generalized Presumed Abort: Marrying Presumed Abort and SNA’s LU 6.2 Commit Protocols,” IBM Research Report RJ8684, IBM Almaden Research Center, 1992.

[2] Kshemkalyani, A., G. Samaras and A. Citron, “Context Management and its Applications to Distributed Transactions,” Distributed Systems Engineering, Vol. 5, No.1, pp. 1-11, 1998.

[3] Gray, J. “Notes on Database Operating Systems,” in Bayer, R., R.M. Graham and G. Seegmuller, (Eds.), Operating Systems: An Advanced Course, LNCS, Vol. 60, pp.393–481, Springer-Verlag, 1978.

[4] Lampson, B. “Atomic Transactions,” in Lampson, B., M. Paul and H.J. Siegert, (Eds.): Distributed Systems: Architecture and Implementation – An Advanced Course, Vol. 105, pp.246–265, Springer-Verlag, 1981.

[5] Mohan, C., B. Lindsay and R. Obermarck, "Transaction Management in the R* Distributed Data Base Management System," ACM TODS, Vol. 11, No. 4, pp. 378-396, 1986.
[6] Al-Houmaily, Y., "Atomic Commit Protocols, their Integration, and their Optimisations in Distributed Database Systems," Int'l J. of Intelligent Information and Database Systems, Vol. 4, No. 4, pp. 373-412, 2010.
[7] Lomet, D., "Using Timestamping to Optimize Two Phase Commit," in Proc. of the 2nd Parallel and Distributed Information Systems, 1993.
[8] Al-Houmaily, Y., "On Deferred Constraints in Distributed Database Systems," Int'l Journal of Database Management Systems, Vol. 5, No. 6, December 2013.
[9] X/Open Company Limited, "Distributed Transaction Processing: The XA Specification," (X/Open Doc. No. XO/CAE/91/300), 1991.
[10] ISO, "Information Technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation)," ISO/IEC 9075-2, 2008.
[11] The PostgreSQL Global Development Group, "PostgreSQL 9.2.4 Documentation," 2013.

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.