Adapting Commit Protocols for Large-Scale and Dynamic Distributed Applications

Pawel Jurczyk and Li Xiong

Emory University, Atlanta GA 30322, USA
{pjurczy,lxiong}@emory.edu

Abstract. The continued advances in distributed data intensive applications have formed an increasing demand for distributed commit protocols that are adaptive to large scale and dynamic nature of the applications to guarantee data consistency and integrity. Well known two-phase commit protocol and three-phase commit protocol and their variants do not provide scalability and flexibility desired for a wide range of applications with potentially different requirements on consistency and efficiency. In this paper, we present an adaptation of three-phase commit protocol for dynamic and scalable distributed systems. The protocol is not only resistant to dynamic network and node failures but also provides good scalability. Most importantly, it offers a set of adjustable parameters for a desired level of consistency at lowest cost depending on specific requirements of different applications.

1 Introduction

An emerging class of distributed data intensive applications relies on large scale distributed data sources in a dynamic and wide-area network setting; examples are enterprise end-system management, workflow management, and computer-supported collaborative work. Consider a nation-wide IT network provider that owns hundreds of thousands of network devices across the country and utilizes hundreds of small servers connecting to local devices to store information reported by them. In order to develop applications such as an enterprise-scale device management system or a report generation tool, data from distributed data sources must be operated. In addition to queries, such data operations also include data creations, updates, and deletions.

One of the key desired properties for building such large scale and dynamic distributed data intensive applications is to guarantee data consistency and integrity. The problem of atomic commitment of transactions has been studied in distributed systems and database systems and it requires that a set of nodes have to either commit or abort a transaction, even in the case of network failures or node failures. Some well-known algorithms for atomic commit are *two-phase commit protocol* (2PC) and *three-phase commit protocol* (3PC) and their variants [1,2,3,4,5]. Many of these traditional commit protocols make strong assumptions that are unrealistic for the wide-area large-scale applications. Below we identify a number of research challenges for adapting distributed commit protocols for such applications.

First, the scale of the applications we consider ranges from a handful of nodes to hundreds of nodes and requires good system scalability. Earlier distributed database systems (R* [2]), while providing a data consistency guarantee, use protocols that are blocking and do not scale well. Second, the dynamic nature of the wide-area networks and varying conditions of system nodes and resources pose a stronger requirements on distributed commit protocols. They need to handle a variety of potential failures and situations including network partitioning as well as multiple node failures. Finally, to support a wide range of applications with potentially different requirements for data consistency and efficiency, commit protocols need to be adaptive and potentially offer a trade-off between the two above requirements. In certain cases, the consistency requirement is mandatory (e.g. systems for a bank or financial institution). On the other hand, systems for analysis purposes will not suffer from relaxed consistency and it is more beneficial to offer more efficient operations at the cost of possibly inconsistent states when rare failures are encountered. We strongly believe that the new-generation distributed data systems need to offer this flexibility and leave decisions to users who can choose between full consistency at a higher cost or some risk of inconsistent state in case of failures for a trade-off for efficiency.

Contributions. We present a distributed commit protocol for supporting a wide variety of applications. The protocol has a number of features distinguishing itself from existing solutions. First, the protocol addresses both small scale systems with a handful of nodes and larger systems with hundreds of nodes. Second, it is resilient to network partitioning and multiple node failures. Finally, the protocol provides a flexible solution in the level of consistency through adjustable parameters and offers a trade-off between consistency and efficiency.

2 Three-Phase Commit Protocol

To facilitate the discussion of our proposed protocol, we first briefly describe the three-phase commit protocol and analyze its limitations. Figure 1 presents the state diagrams of the 3PC [1]. The protocol proceeds as follows. First, the coordinator sends commit request to all cohorts. If all cohorts agree to the transaction, the coordinator moves to the next phase. If any cohort does not agree, the transaction is aborted. After all cohorts agreed to the transaction, the coordinator sends prepare to commit message (pre-commit phase) to cohorts. After all cohorts acknowledge the receipt of the prepare message, the coordinator sends commit messages and commits. The basic version of 3PC does not support *network partitioning* or *multiple nodes failure*. If any of these is met, it can end up in inconsistent state. If network partitioning occurs when coordinator is sending pre-commit messages, nodes that received pre-commit would commit while others would abort. It also assumes atomic transitions from one state to another (the message pre-commit can either be sent to all the cohorts or to none of

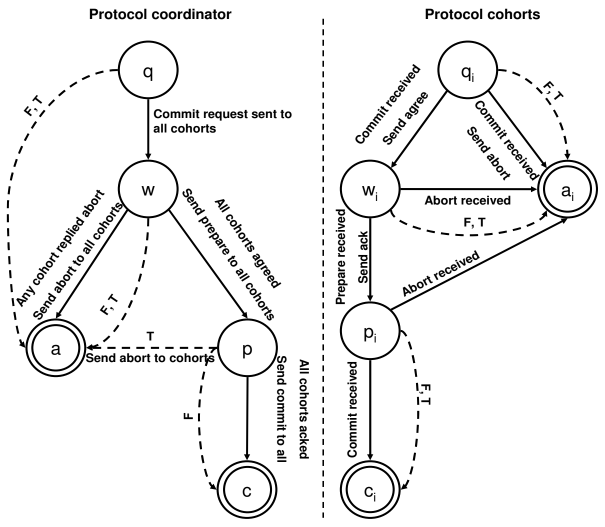

Fig. 1. Basic 3PC protocol

them). Such an assumption is hard to implement in distributed systems without a hardware or operating system support. Extensions of 3PC (Q3PC [3] or E3PC [4] or X3PC), while addressing some of these issues through the idea of quorum, suffer the cost of blocking and also do not offer the desired trade-off between consistency and efficiency.

To achieve our design goal for efficiency and scalability as well as flexibility of consistency level, we also analyzed the 3PC in terms of its efficiency and observed that the efficiency is affected by two factors. First, logging has a large impact. As each log has to be forced to persistent storage before the protocol proceeds, such operations are expected to have impact on the response time of the commit protocol (especially for tightly coupled distributed systems connected by very fast connection backbone). The amount of logged information impacts the response time. Second, the blocking approach in quorum-based 3PC modifications utilizes a timeout (the time after which timeout transition is followed) and blocks transactions that are accesssing data items locked by active transactions. The longer transactions are blocked, the longer many data items can be locked which harms especially wider systems (e.g. systems that offer statistical analysis) where consistency is not of critical importance and can be relaxed.

3 Proposed Protocol

To address the issues with 3PC, we devised a protocol based on our three design goals for large-scale and dynamic distributed applications. In addition to using quorum to decide about unresolved transactions to address network partitioning and multiple node failures, our protocol introduces a set of adjustable parameters so that users can achieve a desired level of consistency and efficiency. In this section, we introduce our assumptions, and present our protocol.

Protocol assumptions. We assume nodes in our commit protocol provide an interface that performs operations in the following way. When a node successfully executes an operation, it guarantees that the operation can be locally committed. Later, the protocol can either commit or roll back the operations. However, when node failure occurs after operations were executed, but before transaction is committed or rolled back, the state of operations is unidentified. We assume that nodes use write ahead log (WAL), i.e. information is logged on persistent storage before any actions on local operations are performed. We also *relaxed* the assumption of *atomic state transition* in 3PC. Our assumption is that, when a state transition is performed, operations associated with this transition are executed in the new state. However, the operations do not have to be executed atomically. For instance, when the transition between from state w to p is performed at the coordinator, the sending of prepare messages starts *after* the transition and the coordinator can fail after sending any number of messagess.

Protocol parameters. The key idea of our protocol is to introduce adjustable parameters to make it configurable so that users can achieve the best consistency and efficiency at the lowest cost in terms of system resources. As we discussed, the amount of log data and the timeout value both have a large impact on the efficiency of 3PC and its variants. This motivates us to design two key parameters for our protocol, namely, *log level* and *transaction timeout*.

Log level. The log level allows the protocol to switch between different levels of logged information: no logging, optimistic logging and full logging. In the case of *no logging*, nodes do not log any information to persistent storage. In the case of *optimistic logging*, each node logs to persistent storage only information about crucial states of the protocol (commit request, prepare or commit). However, no information about operations being performed is logged. In this case recovery options are limited. When a node fails after operations were executed but before they were committed, it can obtain the state of the transaction that it was participating in. However, it does not have information about operations that should be executed to undo/redo the transaction. In the case of *full logging*, each node logs to persistent storage states of protocol and operations for undo/redo recovery. As a consequence, nodes have full information that enables undo or redo of any operations even if it fails before committing or rolling back.

Transaction timeout. This parameter defines the time an unresolved transaction can persist in the system. Note that this is different from the message timeout used in 3PC. When the transaction timeout is reached, decision is made even if some participants of the commit protocol are not available. Such an approach can lead to inconsistent decisions that have to be solved by users (e.g. resubmitting the transaction after the system was repaired). When the timeout is set to infinity, the transaction will not be finished before the agreement is reached.

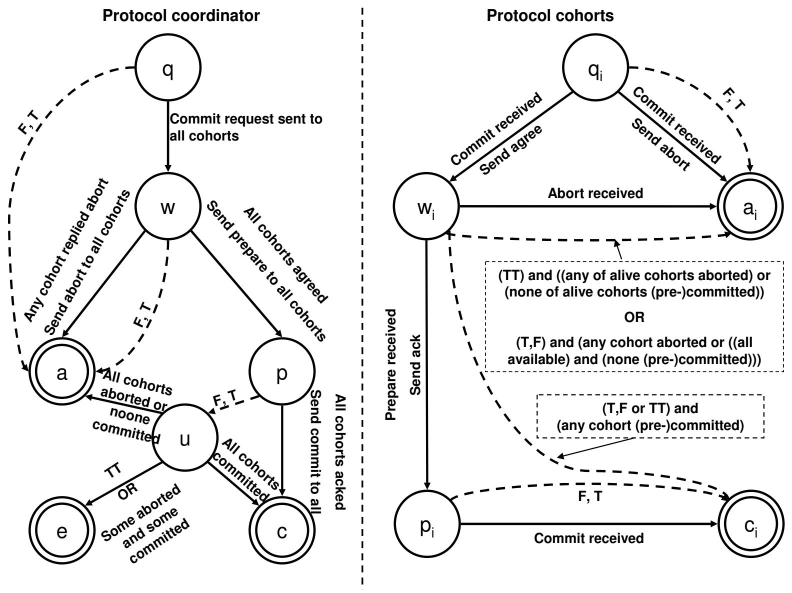
Protocol coordinator
q
Commit request sent to all cohorts
F, T
w
Any cohort replied abort
Send abort to all cohorts
All cohorts agreed
Send prepare to all cohorts
F, T
a
p
All cohorts aborted or noone committed
F, T
u
Send commit to all
All cohorts acked
TT
OR
Some aborted and some committed
All cohorts committed
e
c
Protocol cohorts
q_i
Commit received
Send agree
Commit received
Send abort
F, T
w_i
Abort received
a_i
(TT) and ((any of alive cohorts aborted) or (none of alive cohorts (pre-)committed))
OR
(T,F) and (any cohort aborted or ((all available) and (none (pre-)committed)))
Prepare received
Send ack
(T,F or TT) and (any cohort (pre-)committed)
F, T
p_i
Commit received
c_i

Fig. 2. Commit protocol. T is timeout waiting for next message, TT is *transaction timeout.*

Protocol details. Figure 2 presents the state transition diagram of the proposed protocol. The initial states work similarly to 3PC. We discuss below the later states of the protocol. Readers should be aware that two types of timeouts are used. Timeout T is the timeout when waiting for a message as used in 3PC. *Transaction timeout* TT is the parameter we introduced in our protocol representing the timeout for a transaction to resolve.

At the coordinator site, a new state called u representing *unknown outcome* of the transaction is introduced to solve the issue of potential *non-atomic state transition.* When coordinator is in state p, it can start sending pre-commit to cohorts. When coordinator fails just after sending none or a few messages, but not all of them, the exact outcome of a transaction is unknown (it could not happen in original 3PC because of its atomic state transition assumption). In this case, cohorts can achieve full or partial agreement by communicating with each other. When all cohorts commit, transaction commits on coordinator site and state c is reached. On the other hand, when all cohorts abort, transaction aborts and state a is reached. Finally, when some cohorts commit, some abort or when some cohorts are not available and the transaction timeouts, the error state e is reached. In this case, additional steps have to be performed by user (e.g. the transaction has to be resubmitted). Depending on the *transaction timeout* and *log level*, the protocol may have different outcomes and achieve different consistency and we will analyze it in detail in next section.

At the cohort site, new state transitions are introduced at state w_i to solve the issue of coordinator failure during the state transitions. In the original 3PC, both failure and timeout transitions lead to state a. However, such an approach is not viable anymore given the possible non-atomic transitions in coordinator. In case of coordinator failure during some of the state transitions (e.g. sending pre-commit or abort messages), some cohorts may be notified with the messages and eventually commit or abort the transaction while others are not and have to make a final decision with respect to possible decisions of others. When message timeout T or failure F occurs at state w_i, and all cohorts are present, transaction can be committed or aborted depending on the state of other cohorts. The approach here is similar to the quorum-based protocols but has a key difference with the adjustable *transaction timeout.* If transaction timeout TT occurs, maximum allowable time for transaction expires and decision has to be made even when not all cohorts are available. An explanation is justified for the timeout T as versus the transaction timeout TT at state w_i. Timeout T can be considered as a timer and when it expires, the node checks the transition conditions. If any condition is satisfied, the corresponding transition is followed. If no condition is satisfied, it stays in the same state and resets the timer.

4 Protocol Analysis

The proposed protocol provides either a blocking protocol that is similar to 2PC or quorum-based 3PC and guarantees full consistency, or a non-blocking alternative that is resilient to network partitioning and multiple node failures at the cost of certain inconsistency. We now analyze the protocol in detail.

4.1 Full Consistency Configuration

Claim 1. When *full logging* and *infinite transaction timeout* are used, all cohorts commit or abort transaction.

No failure. In the case of no failure, we will prove *Claim 1* by contradiction assuming the protocol ends with inconsistent decision. The inconsistent decision would require that some nodes obtained commit and some obtained abort messages. However, such a situation is not possible, as all cohorts are notified with either pre-commit and commit or with abort message and once the coordinator makes decision to commit the transaction, this decision is not changed.

Coordinator failure. We consider a few cases of coordinator failure and show the nodes will reach a consistent state.

Case 1: coordinator fails in state p when it starts sending pre-commit messages to cohorts. Recall that we assume when a state transition is performed, operations associated with this transition are executed in the new state. In this case, no cohorts received commit message and some (or all) received pre-commits. Cohorts are either in state w_i (if pre-commit was not received) or p_i (if pre-commit was received). If a cohort is in state p_i, it will commit following a timeout or failure transition. If a cohort is in state w_i, the first condition in transition to *abort* state will never be satisfied as transaction timeout TT is set to infinity.

The second part of the condition can only be satisfied if any cohort received abort (not possible as coordinator failed before sending commit message, but after sending pre-commits) or none of cohorts involved in transaction received pre-commit or commit (again not possible). Clearly, such transition conditions at cohort sites guarantee that if any cohort received commit or pre-commit, others will not abort but will eventually follow failure or timeout transition from state w_i to c_i and *commit* the transaction.

Case 2: coordinator fails in state a when it starts sending abort messages to cohorts. If the abort message is delivered to any cohort, or if none receives pre-commit or commit message, the protocol guarantees that all other cohorts would follow to *abort* state.

Case 3: coordinator fails in state c when it starts sending commit messages to cohorts. If commit messages are being sent, all the cohorts have acknowledged delivery of pre-commit. This means that each cohort has at least reached state p_i. When cohorts in this state do not receive further messages, they follow timeout transition to *commit*, which guarantees consistency.

Cohort failures. We consider a few cases of cohort failure and show the nodes will reach a consistent state.

Case 1: cohort fails in state q_i. In this case, it follows failure transition to *abort* state immediately. When any cohort is in the above state, coordinator is in state w or q and in case of failure or timeout it follows to *abort* state. Therefore, both coordinator and failed cohorts abort transaction. Cohorts that did not fail can be in either state q_i or w_i. As they would not receive further messages from coordinator, they follow timeout transition to *abort* state.

Case 2: cohort fails in state w_i. In this case, coordinator can be either in state w (no pre-commit was sent) or p (some pre-commit messages might have been sent). If coordinator is still in w and fails, it moves to *abort.* Other cohorts will also eventually reach the abort state (including the failed cohort). If coordinator is in state p and it fails, it moves to state u. However, some cohorts might have received pre-commit message and moved to state p_i and eventually *commit* state. The failed cohort in state w_i contacts other cohorts after recovery, which guarantees the consistency. Note that as *full logging* was used, cohort has all the information to perform recovery. Also note that infinite *transaction timeout* guarantees that either the cohort that knows the outcome of transaction has to be contacted or the protocol waits for all cohorts to verify that outcome is unknown and aborts transaction. If pre-commit message was delivered to any cohort, the transaction will be committed. If no one received pre-commit, the transaction will be aborted. Finally, as coordinator being in state u sees that all the cohorts either committed or aborted, it makes the final decision as well.

Case 3: cohort fails in state p_i. In this case, it commits the transaction after recovery. If a cohort is in state p_i, coordinator can be in state p or c and other cohorts can be either in state w_i or q_i. If coordinator is in state p and fails, it fails after sending at least some pre-commit messages as failed cohort received this message. When other cohorts that have not received pre-commit message timeout from state w_i, they will verify the state of other cohorts and eventually *commit* as *transaction timeout* being set to infinity ensures that at least one cohort is aware of the outcome or all cohorts are required for making final decision. If coordinator is in state c and fails, all cohorts must have received the pre-commit message and are in state q_i. Therefore, all the cohorts follow failure or timeout transactions and *commit*.

Network partitioning. Network partitioning disables communication capabilities between sites. If partitioning occurs, messages will not be delivered between sites and in consequence timeout transitions will be followed. As timeout transitions in our protocol are parallel to failure transitions, network partitioning will cause similar scenarios as node failures described above.

4.2 Other Configurations

No logging. When a node fails and recovers, no information about the state of the transaction is available. In fact, the node is not even aware that the transaction existed. There are some scenarios that will still lead to consistent state. For instance, when coordinator fails before sending commit messages but after sending all the pre-commits, all cohorts would timeout from states p_i and the transaction would be committed among all other cohorts. There are many cases, however, that can lead to inconsistency. One example is as follows. Assume coordinator starts sending pre-commit messages and it fails after the first one is sent (when coordinator fails and no logging option is used, it does not follow any failure transitions and simply forgets about the transaction). Assume the cohort that received pre-commit message waits, times out to *commit* state, then fails and forgets about the transaction. Then the rest of the cohorts time out, and start to ask other cohorts about the outcome of the transaction. When failed cohort recovers, however, it does not have any information about the past transaction so it cannot inform other cohorts about the fact it committed the transaction. In consequence, other cohorts abort and the state is inconsistent.

Optimistic logging. Optimistic logging is sufficient to solve the inconsistency scenario above. In this case, each node participating in the transaction logs critical steps in the commit protocol. When the failed cohort recovers, it is aware that the transaction was committed and notifies other cohorts about the status. Therefore, other cohorts pending in state w_i would follow transition to commit state and consistent state is maintained. While solving some of the inconsistency situations, optimistic logging can also lead to inconsistency. Consider the scenario above again and assume another cohort fails in state w_i. Even though the node has the knowledge of the transaction and its state, because of the limited logging information, it may not be able to perform necessary undo/redo recovery depending on the transaction outcome in order to reach a consistent state. Such a problem is solved by *full logging* as discussed in subsection 4.1.

Table 1. Summary of available consistency level depending on protocol parameters

	No log	**Optimistic log**	**Full log**
Finite timeout	Consistency if protocol finishes before timeout (nodes failure cause inconsistency). Possibly no information for users about the outcome of transaction	Consistency if protocol finishes before timeout (nodes failure can cause inconsistency)	Consistency if protocol finishes before timeout
Infinite timeout	Resistant to net. failures, nodes failures lead to inconsistency. Possibly no information for users about the outcome of transaction	Resistant to net. failures, nodes failures can lead to inconsistency	Full consistency, prone to net. and node failures, blocking protocol

Finite transaction timeout. We now take a closer look at what happens when a finite value is specified for transaction timeout. The *transaction timeout* limits amount of time that nodes can wait before being able to contact others in case of network partitioning or node failures. In case of *network partitioning* without node failures, assume some cohorts in state w_i become unreachable from the coordinator, these cohorts will simply wait for messages. When none is received, they face timeout and attempt to contact others to find the outcome of the transaction. However, if network failure is not fixed before *transaction timeout* is reached, decision can be made without respect to some of the cohorts. This can lead to inconsistency. For instance, when coordinator sends pre-commit messages and network partitioning occurs, the group of connected nodes that includes cohorts who received pre-commit would commit while the group of nodes that did not receive any pre-commit would abort.

In case of node failures, the scenarios we discussed for each of the *log levels* remain valid when a finite *transaction timeout* is used. The parameter limits the waiting time for other cohorts to restart. If a failed cohort restarts before *transaction timeout* occurs, the scenario is followed as discussed and consistency is maintained to the extent provided by the given *log level*. If the node does not restart before the *transaction timeout* occurred, some cohorts can make decisions without respect to the state of the transaction on failed cohorts and inconsistent states can be reached. Table 1 contains a summary of available parameters.

5 Conclusion

We have presented a new commit protocol based on 3PC that is scalable and resistant to dynamic network and node failures, and provides a configurable level of consistency depending on specific application and system deployment characteristics. When *full logging* and *infinite transaction timeout* are used, full consistency is guaranteed at the cost of blocking. This is not surprising, as it has been proven that there does not exist a non-blocking commit protocol resilient to multiple nodes failure or network partitioning [1]. In case of other parameter configurations, the protocol is resistant to certain types of failures, but can lead to inconsistency. Our future work aims to provide fault tolerance properties for the protocol. In particular, we are planning to introduce data replications and extend the system with tolerance of Byzantine node failures.

References

1. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. Concurrency control and reliability in distributed systems, 295–317 (1987)
2. Mohan, C., Lindsay, B.G., Obermarck, R.: Transaction management in the r* distributed database management system. ACM Trans. Database Syst. 11(4), 378–396 (1986)
3. Skeen, D.: A quorum-based commit protocol. Technical report, Cornell University, Ithaca, NY, USA (1982)
4. Keidar, I., Dolev, D.: Increasing the resilience of atomic commit, at no additional cost. In: Proc. of the PODS (1995)
5. Kempster, T., Stirling, C., Thanisch, P.: A more committed quorum-based three phase commit protocol. In: Proc. of the DISC (1998)