Module 8: Reliability Protocols

Reliability Protocols ensure both Atomicity and Durability. Transactions consist of multiple operations that need to function as if they were atomic operations. Atomic operations need to work together as a single unit. They change the database from one consistent state to another consistent state. If only some, but not all of the operations were to work, the database would be left in an inconsistent state.

There are many reasons why a database transaction would fail in the middle of operations. Some failures are listed below:

- Transaction Failures
  - Incorrect input data
  - Deadlock
  - Timeouts
- Site Failures
  - Hardware failures
    - Processor
    - Main memory
    - Power supply
  - Software failures
    - OS code
    - DBMS code
- Media Failures
  - Secondary storage
  - All or part of the data on disk is considered destroyed
- Communication Failures (unique to distributed DBMS)
  - Errors in messages
  - Improperly ordered messages
  - Lost messages

Techniques for Maintaining the Integrity of Secondary Storage

Frequent Backups – Periodically, data is backed up onto long-term storage media. When data is lost, the backed up data is used to restore the lost data. If backups are done infrequently, there is a risk that the newest data will not have been backed up and therefore cannot be restored. The risk will exist no matter how frequently the data is backed up, although frequent backups minimize the amount of data that would be lost.

Disk Shadowing – Every data element that is written to disk is simultaneously written to a *shadow* disk. If the primary disk crashes, the secondary disk is used until the primary disk is restored/replaced with the data from the shadow disk (see Figure 1). The benefit is that data is always available even when a disk crashes. The cost is that double the needed storage must be purchased to support a disk shadowing scheme. If there is concern that two disks may simultaneously crash, triple the needed storage must be purchased.

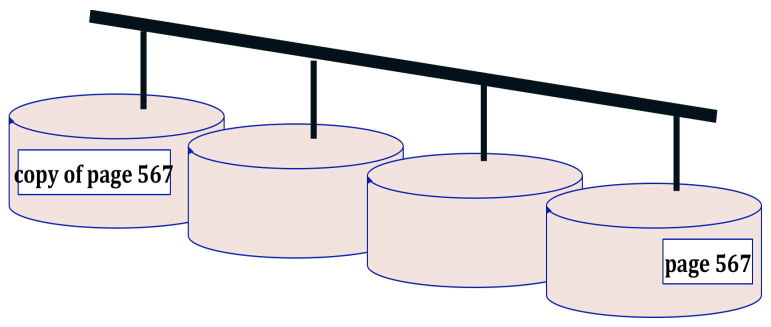


Figure 1. A Disk Shadow Scheme

RAID - Redundant Array of Independent Disk (RAID) is a clever scheme that does not require double the storage to maintain up-to-the-moment backup data. In a RAID scheme, multiple disks are striped with data. For example, Figure 2 depicts four bits of data (0110) written to four disks. Each bit is written to one of the disks. A fifth disk is used to store the exclusive or (XOR) of the bits. An XOR of two bits is equal to 1 if one bit is 0 and the other is 1; otherwise it is equal to 0. In this case, the XOR of 0110 is (((0 xor 1) xor 1) xor 0) => ((1 xor 1) xor 0) => (0 xor 0) => 0. If one of the disks crashes, its value is restored by applying XOR to the remaining bits including from the backup disk. In this case (((0 xor 1) xor 0) xor 0) => ((1 xor 0) xor 0) => (1 xor 0) => 1. Thus, in the event of a disk crash, operations can continue without stopping. When the crashed disk is replaced, its values are restored by performing XOR operations on all the rest of the disks including the backup disk, and writing the values to the replaced disk.

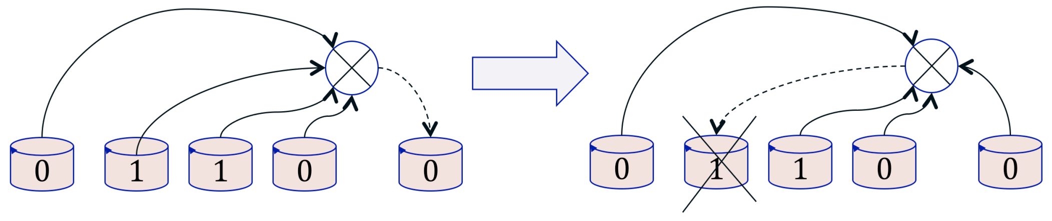


Figure 2. The RAID Approach

Recovery Architecture

Cache storage is used to speed access to data. We make several assumptions about data access. First, secondary storage (e.g., disk) is very stable. While this is not always true, it is generally much more stable than main memory. Second, access to secondary storage is relatively slow. Third, access to cache is relatively fast. Figure 3 depicts a cache architecture.

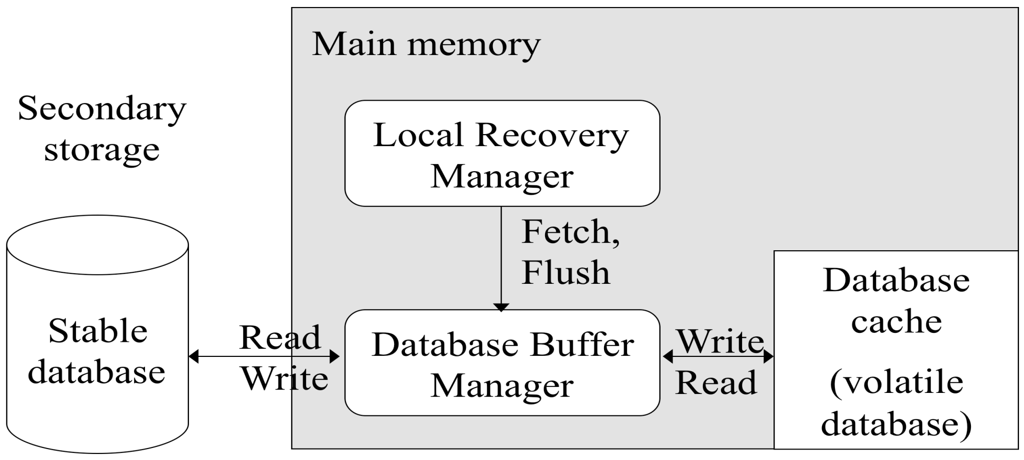


Figure 3. Standard Cache Architecture

The algorithm that is employed is the following:

1. Upon request, search the cache for the requested memory page.
2. If it is available, return the requested page.
3. If it is not available, retrieve the corresponding page from stable memory and place the cache.
4. Many cache algorithms anticipate the next several pages that will be requested and returns them to cache.
5. If the cache is full, write one of the cache pages back to stable memory and replace it with the requested page.

There are a number of algorithms used to determine which page(s) to write back to stable storage. The most common is the Least Recently Used (LRU) algorithm, which keeps access timestamps for each page in cache and returns to stable storage the page with the oldest timestamp.

(Watch Module 8A video – Approaches to Cache Management)

ARIES Recovery Algorithm

The ARIES Recovery algorithm provides a simple approach to recovery management. A log record is written every time a row is written, modified, or deleted in the database. When a crash occurs, the ARIES algorithm executes a three phase recovery algorithm.

1. ANALYSIS Phase

In this phase, the dirty pages in cache with the oldest timestamp are identified. This is where the REDO phase starts. All pages older than this time stamp have been written to stable storage. In addition, the algorithm identifies the active transactions at the time of the crash. Since these transactions did not complete, they are UNDOne.

2. REDO Phase

   This phase starts from earliest dirty page and repeats all actions starting from the appropriate point in the log. The purpose of this phase is to bring the database to its state before crash.

3. UNDO Phase

   This phase undoes the actions of the transactions that did not commit. It only undoes the uncommitted transactions. Any modifications during this phase are themselves logged to prevent re-undoing of changes in the event that the system crashed during this phase.

The Log

The log is a history of all actions executed by the DBMS. It must be written to before an actual change is executed; otherwise the record may be lost during a crash. Log records are given a unique ID called the log sequence number (LSN), which are granted in monotonically increasing order.

The log captures the following information:

1. Page updates – the update record is written to the log file and the LSN must be written on the page as well.
2. Commit transaction
3. Abort transaction
4. End transaction
5. Undoing an update – these are for rollbacks or aborts. A compensating log record is written so that undo records are not undone in case of another crash as described above.

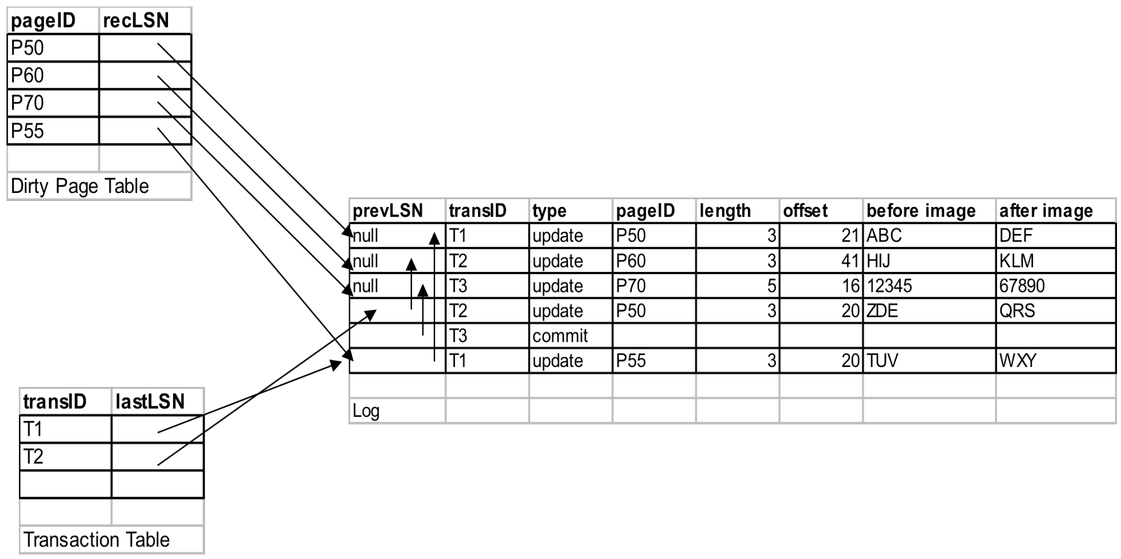
The log record contents must include the previous LSN of the transaction, the transaction ID, and the type of record (update, commit, abort, end, or undo update as listed in the previous paragraph).

In addition, update records must contain the page ID of the modified page, the offset on the page of the changed data, the length of data changed, the before image, and the after image.

Finally, two tables are stored. The first table is the Dirty Page Table, which is used for the REDO phase. There is one entry for each dirty page in the cache that contains the first log record that caused the page to become dirty. The second table is the Transaction Table, which is used for the undo phase. Each row contains the most recent log record for each uncompleted transaction. Entries for complete transactions are deleted from the table.

The following demonstrates the progression of a transaction and how these tables are populated.

(Watch Module 8B video – Example Transaction)



pageID	recLSN
P50	
P60	
P70	
P55	

Dirty Page Table

prevLSN	transID	type	pageID	length	offset	before image	after image
null	T1	update	P50	3	21	ABC	DEF
null	T2	update	P60	3	41	HIJ	KLM
null	T3	update	P70	5	16	12345	67890
	T2	update	P50	3	20	ZDE	QRS
	T3	commit					
	T1	update	P55	3	20	TUV	WXY

Log

transID	lastLSN
T1	
T2	

Transaction Table

Figure 4. The State of the Log, Dirty Page Table, and Transaction Table at the Crash

The following provides an example of how the ANALYSIS and REDO phases work.

(Watch Module 8C video – Analysis and Redo Phases)

The following provides an example of how the UNDO phase works.

(Watch Module 8D video – Undo Phase)

In summary, the ARIES algorithm is a commonly used algorithm that performs recovery management when a single database crashes. The next section discusses the algorithms used to implement distributed database reliability protocols, which incorporate ARIES protocols on individual database systems.

Distributed Reliability Protocols

The goal of distributed reliability protocols is to maintain *atomicity* and *durability* of distributed transactions that execute over multiple databases. Each database individually implements reliability protocols such as the ARIES algorithm to ensure the atomicity and durability of transactions. The rest of this module focuses on implementing reliability protocols when individual transactions span multiple database systems.

The first operation that a recovery protocol must address is *commit and recovery*. This ensures that if one site fails in its part of a transaction, the other sites terminate as well. The second operation that must be addressed is *termination and recovery.* Sites need to deal with failure and they need to recover once the failed sites restart.

Distributed reliability protocols attempt to implement *non-blocking* and *independent recovery* protocols. Non-blocking protocols ensure that transactions that are terminated at one site do need to wait for failed sites to recover. *Independent* recovery protocols let sites recover without having to consult other sites in a transaction. *Independence* implies *non-blocking*, but not vice versa. We discuss these in more detail as we discuss both the two-phase and three-phase commit protocols.

Finally, distributed reliability protocols require systems to participate as *actors* with different responsibilities. Typically, there is a coordinator that is responsible for coordinating the states of the other active members for transactions. In addition, there are participants that carry out one of the components of a distributed transaction on one machine.

Two- Phase Commit Protocol (2PC)

The 2PC protocol ensures atomic commitment of distributed transactions. All sites must agree to commit before any permanent effects take place on any of the sites. There are a number of reasons why a specific site will not terminate a transaction. For example, a site scheduler may not be willing to terminate a transaction based on the state of its concurrency algorithm. Another reason a transaction may not be ready to commit is because deadlocks prevent it from committing.

The steps of the 2PC algorithm are depicted in Figure 5.

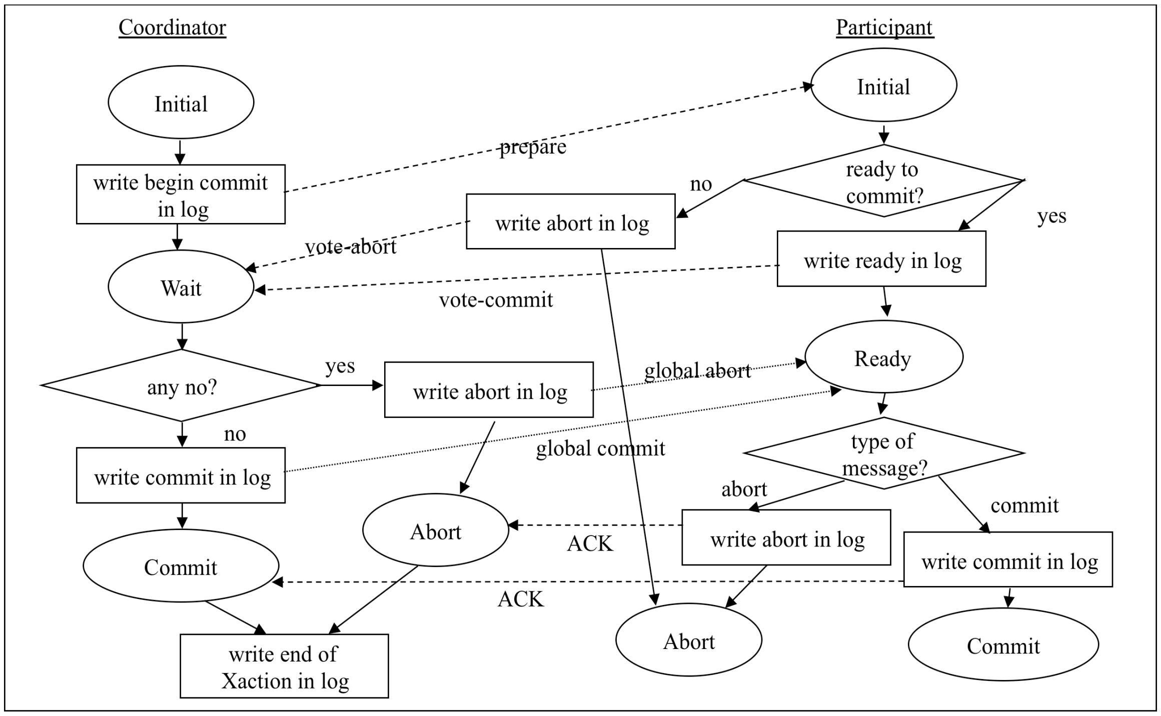


Figure 5. Steps of the 2PC Protocol

The following steps summarize the 2PC protocol:

1. The *Coordinator* starts by asking participants to prepare for distributed transaction, then enters wait state.
2. The *Participants* determine whether or not they can commit or not.
3. The *Participants* vote.
   a. If one votes to commit, it enters ready state and waits for the coordinator's response.
   b. If one votes abort, it forgets the transaction.
4. The *Coordinator* decides whether the transaction should continue.
   a. If all participates vote commit, it sends a global-commit to all participants.
   b. If just one aborts, it sends a global-abort to all participants.
5. The *Participants* act.
   a. If they receive a global-commit, they commit their transactions.
   b. If they receive a global-abort, they abort their transactions.
6. Transactions logs are written at various points to support local recovery algorithms if any of the sites fail.

In the 2PC algorithm, each participant can unilaterally abort. Once a participant aborts, it cannot change its vote. The result is that the transaction will abort. Furthermore, even if a participant is ready to commit, the transaction can still abort or commit depending on the final determination of the coordinator. This is because the coordinator makes the final decision based on an all-or-nothing vote that ensures atomicity of the distributed transaction. It is possible that the coordinator and participants enter states where they wait for each other, but do not hear from the other. In this case, they both set timers and if the time limit is reached, they end the transaction without committing.

Figure 6 illustrates the two phase aspect of the protocol. The first phase prepares the transactions to determine if all participant wish to commit. The second phase carries out the transaction so that everyone commits or aborts.

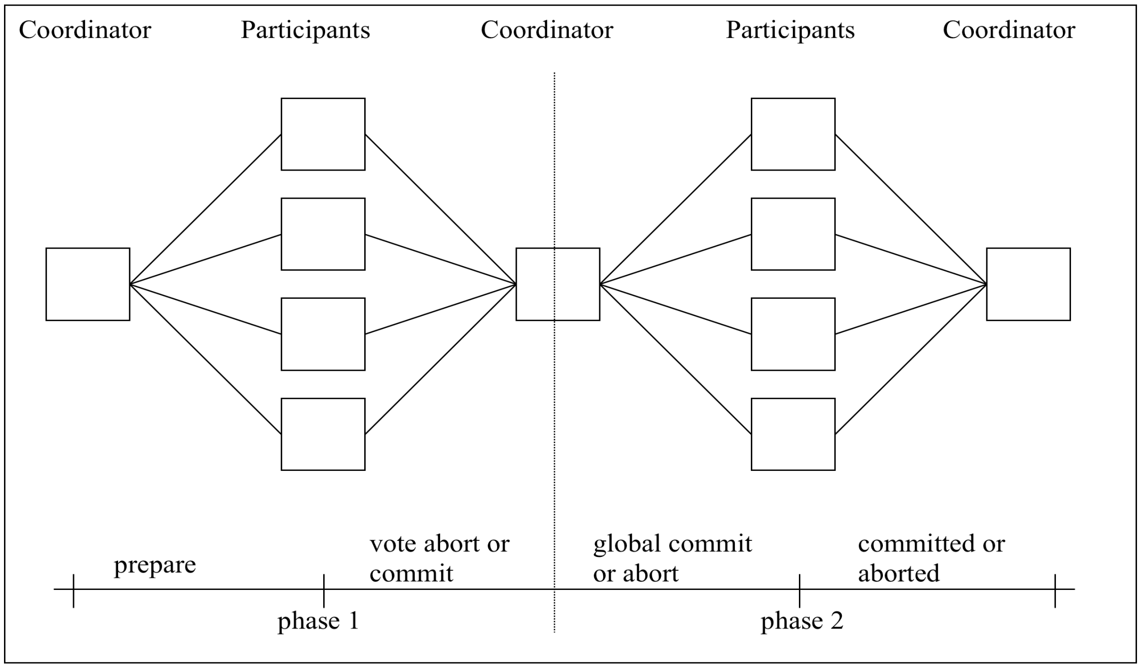


Figure 6. Illustration of the Two Phases of 2PC

The centralized 2PC algorithm has some drawbacks. There are many messages that are sent between the coordinator and participants, which can require much bandwidth. Furthermore, all participants must be synchronized. An alternative algorithm is Linear 2PC, which addresses some of these issues. In Linear 2PC, there is an ordering of sites for communications for those participating in the distributed transaction (1, 2, …, N); the coordinator is always the first site.

Figure 7 depicts the algorithm. The steps are:

1. The *Coordinator* (#1) sends a prepare message to participant. (#2)
2. Participant (#2) decides to abort or commit.
    a. If abort, (#2) sends abort message to (#3)
    b. If commit, (#3) sends commit message to. (#3)
3. All subsequent *Participants* act in the same way.
    a. If they receive an abort, they send abort message to the next participant.
    b. If they receive a commit, they determine if they can commit and send their vote onward.
4. The last *Participant* sends a global-commit or global-abort back from (#N) to (#N-1) to … to (#1).

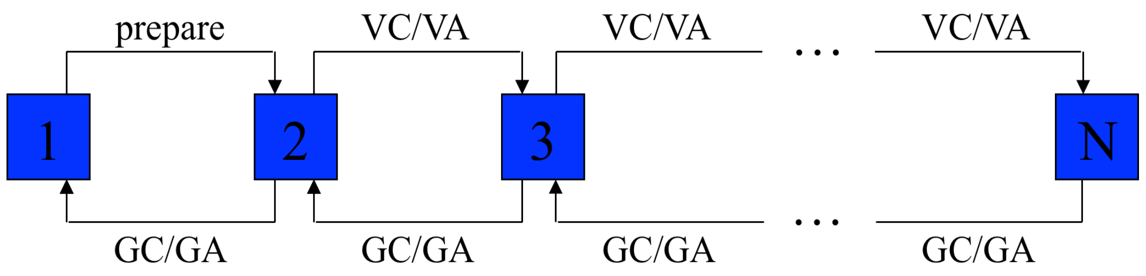


Figure 7. The Linear 2PC Algorithm

The drawback of Linear 2PC is that it is slow. Another approach is called the Distributed 2PC algorithm. All participants communicate with each other and independently arrive at a conclusion. This is depicted in Figure 8. All votes of the participants in the first phase are passed back to the coordinator and all the participants. Each decides how to act based on all the votes.

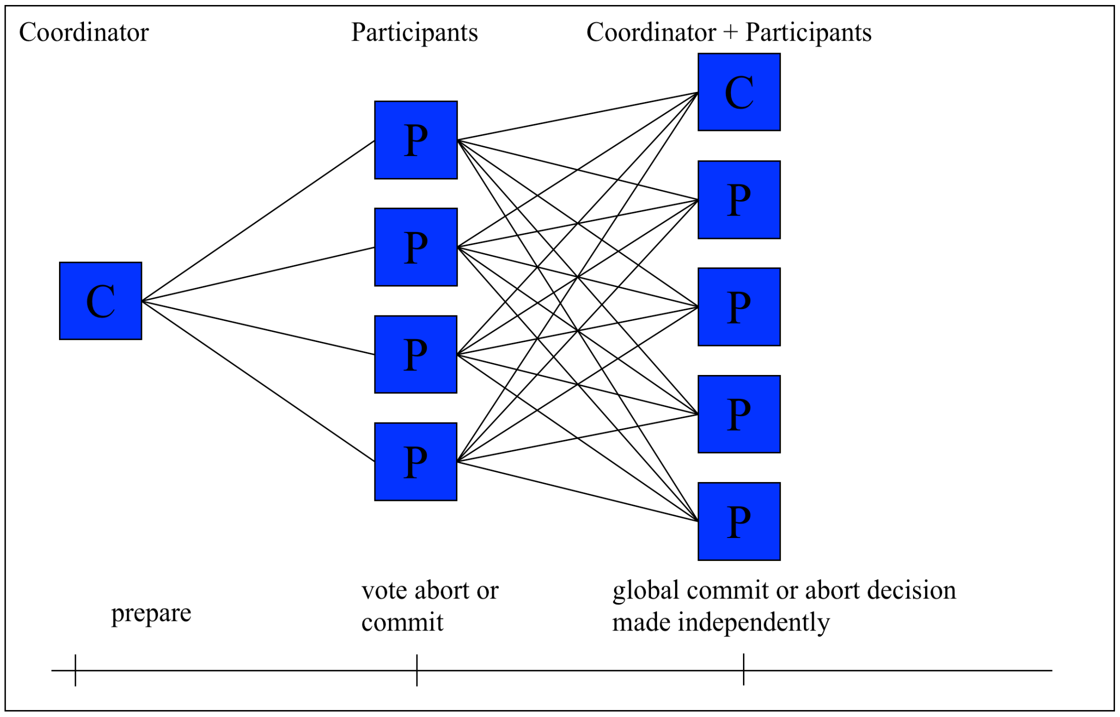

Coordinator
Participants
Coordinator + Participants
C
P
P
P
P
C
P
P
P
P
prepare
vote abort or commit
global commit or abort decision made independently

Figure 8. Illustration of the Distributed 2PC Algorithm

The advantage of Distributed 2PC is that it eliminates the second phase of communication. The disadvantages of Distributed 2PC are that each participant must be aware of every other participant and that $O(N^2)$ messages must be sent if there are N participants.

Site Failures

The distributed reliability protocols need to recover when a site fails. It is desired that the protocols be *independent,* where each site performs its own recovery, and *non-blocking*, where each site can proceed without waiting for other sites' operations. If they are independent, they are non-blocking. However, the reverse is not necessarily true – non-blocking does not imply independence. It is possible to design such protocols for a single site failure, but not for multiple site failures. The next paragraphs and video link below demonstrate the termination and recovery algorithms for 2PC and show how they are inherently blocking. The Three Phase Commit (3PC) will then be described to make the termination and recovery algorithms non-blocking.

(Watch Module 8E video – Termination Protocols)

Three- Phase Commit Protocol (3PC)

The 3PC Protocol addresses the blocking issues of 2PC. The essential problem with 2PC is that if the Coordinator fails when a Participant is in the READY state, there are situations in which the Participant does not know if the Coordinator has decided to COMMIT or ABORT. As a result, the Participant sits in a blocked state because its next two possible states are either COMMIT or ABORT actions. The Participant would not block if there were no state that were adjacent to both COMMIT and ABORT. Furthermore, the Participant would not block if a non-committable state was adjacent to a COMMIT state. 3PC addresses these issues as depicted in Figure 9.

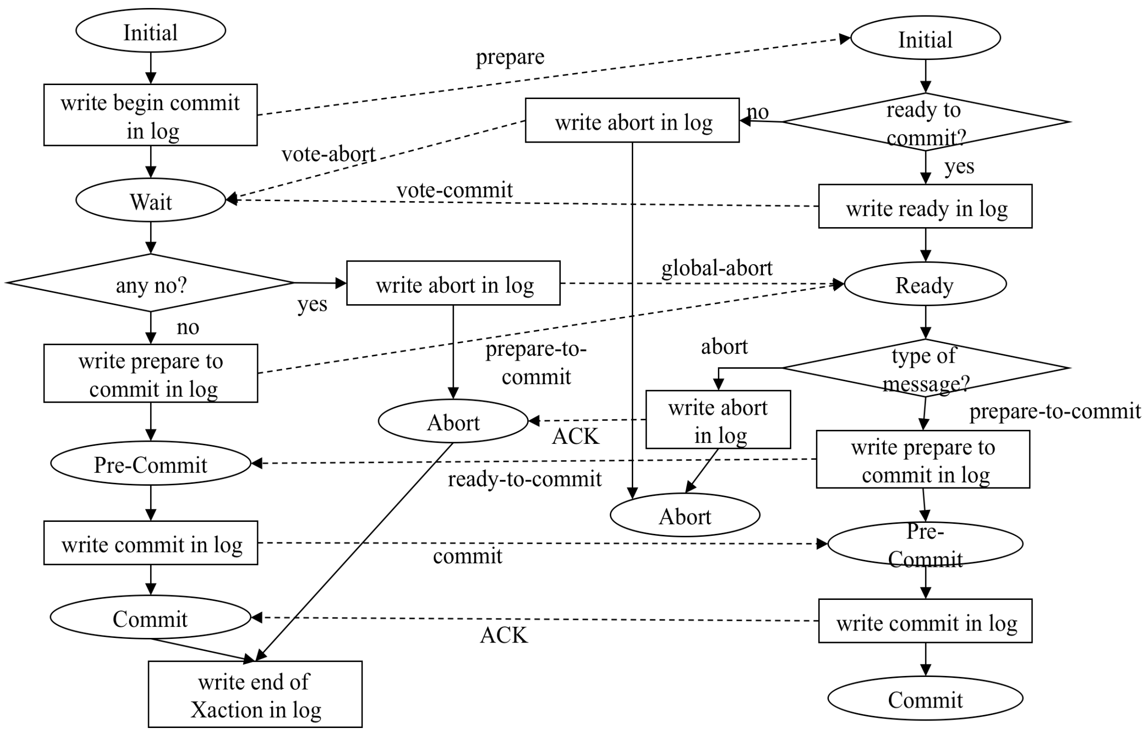


Figure 9. The 3PC Protocol

The following steps summarize the 2PC protocol:

1. The *Coordinator* starts by asking participants to prepare for distributed transaction, then enters wait state.
2. The *Participants* determine whether or not they can commit or not.
3. The *Participants* vote.
    a. If one votes to commit, it enters ready state and waits for the coordinator’s response.
    b. If one votes abort, it forgets the transaction.
4. The *Coordinator* decides whether the transaction should continue.
    a. If all participates vote commit, it sends a **prepare-to-commit** to all participants and moves into the **pre-commit** state.
    b. If just one aborts, it sends a global-abort to all participants and aborts.
5. The *Participants* act.
    a. If they receive a **prepare-to-commit**, they move to a **pre-commit** state and send a **ready-to-commit** to the coordinator.
    b. If they receive a global-abort, they abort their transactions.
6. The *Coordinator* acts.
    a. If it receives a **ready-to-commit** from all the participants, it sends a commit to all the participates.
    b. It moves to the commit state and commits the transaction.
7. The *Participants* act.
    a. If they receive a commit, they move to the commit state and commit the transaction.
8. Transactions logs are written at various points to support local recovery algorithms if any of the sites fail.

The timeout and termination protocols of 3PC are discussed in the following video.

(Watch Module 8F video – 3PC Timeout and Termination Protocols)

Conclusion

The reliability protocols round out the ACID properties of transactional database systems by addressing the Atomicity and Durability properties. The ARIES algorithm ensures reliability for single database systems. The 2PC and 3PC algorithms ensure reliability for distributed database systems. 3PC is an improvement over 2PC in that each system can act independently without blocking.