# [How do two-phase commits prevent last-second failure?](http://stackoverflow.com/questions/171876/how-do-two-phase-commits-prevent-last-second-failure)

I am studying how two-phase commit works across a distributed transaction. It is my understanding that in the last part of the phase the transaction coordinator asks each node whether it is ready to commit. If everyone agreed, then it tells them to go ahead and commit.

What prevents the following failure?

1. All nodes respond that they are ready to commit
2. The transaction coordinator tells them to "go ahead and commit" but one of the nodes crashes before receiving this message
3. All other nodes commit successfully, but now the distributed transaction is corrupt
4. It is my understanding that when the crashed node comes back, its transaction will have been rolled back (since it never got the commit message)

I am assuming each node is running a normal database that doesn't know anything about distributed transactions. What did I miss?

Ans

No, they are not instructed to roll back because in the original poster's scenario, some of the nodes have already committed. What happens is when the crashed node becomes available, the transaction coordinator tells it to commit again.

Because the node responded positively in the "prepare" phase, it is required to be able to "commit", even when it comes back from a crash.

The crashed DB doesn't have to replay the transaction. The way traditional DB transactions work is they are actively making those changes in the transaction log, which only occurs if it is able to obtain the necessary row/table/page locks. Essentially after the first phase, the transaction has already modified the DB, but other queries cannot see those changes until they are marked as committed. All that is left to do is mark them as committed.

"The protocol assumes that there is stable storage at each node with a write-ahead log, that no node crashes forever, that the data in the write-ahead log is never lost or corrupted in a crash, and that any two nodes can communicate with each other."

But the argument is how the ode is supposed to know to replay the transaction (instead of rolling it back) when it comes back from a crash. As far as I understand it, the database isn't aware of the distributed transaction; something on top of it does.

But the database has to know about the distributed transaction to support 2PC correctly. If you build something on top, then you don't have a reliable 2PC implementation

Summarizing everyone's answers:

1. One cannot use normal databases with distributed transactions. The database must explicitly support a transaction coordinator.
2. The nodes are not instructed to roll back because some of the nodes have already committed. What happens is that when the crashed node comes back, the transaction coordinator tells it to commit again.

# [Does Two-phase Commit Protect Against Final Commit Failures?](http://stackoverflow.com/questions/11172588/does-two-phase-commit-protect-against-final-commit-failures)

Suppose all the databases involved in a distributed transaction implemented with two-phase commit signal that they are ready to commit and have the necessary locks. The coordinator signals to commit and all databases execute their portion of the transaction, but one SQL database encounters a divide-by-zero error as a result of a programming oversight that fails to consider that possibility. Since the coordinator already signaled commit to everyone what happens as a result of that divide-by-zero?

Some questions and views:

Do you mean that the definition of the precommit phase is that everyone actually fully executes their portion of the transaction and that commit phase is defined by simply writing "<committed>" to a log but the critical point is that no actualy execution of the transaction occurs during the commit phase? All the articles on two-phase commit I've encountered never exactly clearly state when each database executes their portion of the transaction

Well, what *actually* happens is implementation specific. But yes, that would pretty much be what happens (changes are made and the only thing that the distributed databases are waiting for is the ack from the coordinator in order to "close the deal" by committing)

Answers

The second commit phase normally does not contain user code that can fail. The participating resource managers need to guarantee that no failure can occur. If this guarantee is violated no guarantees can be provided by the protocol.

Two phase commit tries to solve the [Two Generals Problem](http://en.wikipedia.org/wiki/Two_Generals%27_Problem). There is no full solution to this problem. TPC is an approximation.

Another way TPC can fail is in case of a network partition. Some resource managers might perform the final commit but some might not receive that message. Again, this problem is unsolvable. Even retries cannot solve it.

You can even trigger this problem under real world conditions: Run all participating nodes in a stress test and pull the network cable at an arbitrary point. With high probability your distributed databases are now inconsistent because some commit messages got lost an a very inconvenient time.

# [Two phase commit question](http://stackoverflow.com/questions/7389382/two-phase-commit-question)

 Assume I've two DBs, A and B using 2PC in two different locations. Before A and B are ready to commit a transaction, both DBs will report back to the transaction manager and saying they are ready to commit. So, when the transaction manager is acknowledged, it will send a signal back to A and B telling them go ahead. Here is my question.... Let's say A received the signal and commit the transaction. (everything is completed)... B is about to do the same but someone unplug the power cable cause the whole server shutdown. When B is back online, what B will do? and how B do it? Remember, A is committed but B is not, and we are using 2PC. (So, the design of 2PC stops working, isn't it? :-) "

Two phase commit does not guarantee that a distributed transaction can't fail, but it does guarantee that it can't fail silently without the TM being aware of it.

In order for B to report the transaction as being ready to commit, B must have the transaction in persistent storage (i.e. B must be able to guarantee that the transaction can commit in all circumstances). In this situation, B has persisted the transaction but the transaction manager has not yet received a message from B confirming that B has completed the commit.

The transaction manager will poll B again when B comes back online and ask it to commit the transaction. If B has already committed the transaction it will report the transaction as committed. If B has not yet committed the transaction it will then commit as it has already persisted it and is thus still in a position to commit the transaction.

In order for B to fail in this situation, it would have to undergo a catastrophic failure that lost data or log entries. The transaction manager would still be aware that B had not reported a successful commit.1

In practice, if B can no longer commit the transaction, it would imply that the disaster that took B out had caused data loss, and B would report an error when the TM asked it to commit a TxID that it wasn't aware of or didn't think was in a commitable state.

Thus, two phase commit does not prevent a catastrophic failure from occuring, but it does prevent the failure from going unnoticed. In this scenario the transaction manager will report an error back to the application if B cannot commit.

The application still has to be able to recover from the error, but the transaction cannot fail silently without the application being made aware of the inconsistent state.

1. Note that this sort of failure could also lose data from previously committed transactions. Two phase commit does not guarantee that the resource managers can't lose or corrupt data or that DR procedures don't screw up.

# [Two-phase commit: availability, scalability and performance issues](http://stackoverflow.com/questions/22554382/two-phase-commit-availability-scalability-and-performance-issues)

I have read a number of articles and got confused.

Opinion 1: 2PC is very efficient, a minimal number of messages are exchanged and latency is low. Source: <http://highscalability.com/paper-consensus-protocols-two-phase-commit>

Opinion 2: It is very hard to scale distributed transactions to high level, moreover they reduce throughput. As 2PC guarantess ACID It puts a great burden due to its complex coordination algorithm. Source: <http://ivoroshilin.com/2014/03/18/distributed-transactions-and-scalability-issues-in-large-scale-distributed-systems/>

Opinion 3: “some authors have claimed that two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.” Source:<http://courses.cs.washington.edu/courses/csep552/13sp/lectures/6/spanner.pdf>

Opinion 4: The 2PC coordinator also represents a Single Point of Failure, which is unacceptable for critical systems - I believe it is a coordinator. Source:<http://www.addsimplicity.com/adding_simplicity_an_engi/2006/12/2pc_or_not_2pc_.html>

First 3 opinions contradict each other. The 4-th one I think is correct. Please clarify what is wrong and what is correct. It would be great also to give facts why that is.

The 4th statement is correct, but maybe not in the way you are reading it. In 2PC, if the coordinator fails, the system cannot make progress. It therefore often desirable to use a fault-tolerant protocol like Paxos (see [Gray and Lamport](http://dl.acm.org/citation.cfm?id=1132863.1132867) for example), which will allow the system to safely progress when there are failures.

Opinion 3 should be read in context of the rest of the Spanner paper. The authors are saying that they have developed a system which allows efficient transactions in a distributed database, and that they think it's the right default tradeoff for users of the system. The way Spanner does that is well detailed in the paper, and it is worth reading. Take note that Spanner is simply a way (a clever way, granted) of organizing the coordination which is inherently required to implement serializable transactions. See[Gilbert and Lynch](http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf) for one way to look at the limits on coordination).

Opinion 2 is a common belief, and there are indeed tradeoffs between availability and richness of transaction semantics in real-world distributed systems. Current research, however, is making it clear that these tradeoffs are not as dire as they have been portrayed in the past. See [this talk by Peter Bailis](https://www.youtube.com/watch?v=_rAdJkAbGls)for one of the research directions. If you want true serializability or linearizability in the strictest sense, you need to obey certain lower bounds of coordination in order to achieve them.

Opinion 1 is technically true, but not very helpful in the way you quoted it. 2PC is optimal in some sense, but seldom implemented naively because of the availability tradeoffs. Many adhoc attempts to adress these tradeoffs lead to incorrect protocols. Others, like Paxos and Raft, successfully address them at the cost of some complexity.

# [3 phase commit protocol](http://stackoverflow.com/questions/11265565/3-phase-commit-protocol)

Assume there are two participants A and B and a Coordinator C:

1)C sent precommit message to A and before it sends precommit message to B both A and C simulataneously fail. 2)The transaction is now restarted and B ends up aborting it because no reply from A. 3)A commits the transaction because its has already got the precommit message.

Wasn't this also the original problem in 2PC that 3PC was supposed to address? How is 3PC solving the problem? What am I missing. Thanks.

I guess maybe the way it would go is if coordinator and all participants knowing state of a system fail then the transaction aborts once the new coordinator is elected (i guess in line with what you said). And if the coordinator fails after sending the precommit message and at least one of the particpant having a precommit message, the rest in the system can just go ahead and commit since they already know the state on the system. So at no point is the system is in an undefined state

If participants wait for doCommit from the coordinator atomicity could fail in the case where coordinator sends commit to some particpants and fails before it sends commit to the other partcipants. So I think that wouldnt work. It would sort of be the same flaw that 2 phase commit protocol has

# What is Two-Phase Commit ?

Two-phase  commit  is  mechanism  that guarantees a distributed transaction either commits on all involved nodes or rolls back on all involved nodes to maintain  data  consistency  across the global distributed database. It has two phase, a Prepare Phase and a Commit Phase.