**Consensus Protocols: Two-Phase Commit**

For the next few articles here, I’m going to write about one of the most fundamental concepts in distributed computing – of equal importance to the theory and practice communities. The consensus problem is the problem of getting a set of nodes in a distributed system to agree on something – it might be a value, a course of action or a decision. Achieving consensus allows a distributed system to act as a single entity, with every individual node aware of and in agreement with the actions of the whole of the network.

 For example, some possible uses of consensus are:

* deciding whether or not to commit a transaction to a database
* synchronizing clocks by agreeing on the current time
* agreeing to move to the next stage of a distributed algorithm (this is the famous replicated state machine approach)
* electing a leader node to coordinate some higher-level protocol

Such a simple-sounding problem has surprisingly been at the core particularly of theoretical distributed systems research for over twenty years. How come? As I see it, the answers are threefold.

First, it turns out that consensus is a sufficiently difficult problem that it can be used to characterize the differences between different strengths of system model. As [highlighted in my post on the FLP result](http://hnr.dnsalias.net/wordpress/?p=49), consensus is solvable in synchronous systems, but not in asynchronous ones if there is the possibility of any failures. Therefore the problem is at the phase boundary between the two models and turns out to be heavily dependent on timing assumptions.

Second, getting a consensus protocol right is hard. Simple solutions don’t work very well, exhibiting undesirable behaviors and occasionally acting incorrectly. Mike Burrows, inventor of the Chubby service at Google, says that “there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. The Paxos protocol, conceived by Leslie Lamport, is famously subtle and a bit difficult to understand (although Lamport didn’t necessarily help this problem by presenting the protocol initially by an allegory to Greek parliamentary proceedings). Whether or not you believe this point of view, it’s clear that a good consensus protocol is surprisingly hard to find.

Thirdly, consensus is an important problem. Distributed databases rely on it. In fact, most distributed systems are built upon it. Group membership systems, fault-tolerant replicated state machines, data stores – all the typical distributed systems are at some level predicated upon being able to solve consensus. Partly, this is because consensus is isomorphic to another important problem – atomic broadcast, which is the ability to deliver messages in a network reliably and in a total order to all nodes. We will see later in these articles why atomic broadcast and consensus are two instances of the same problem.

## Formal Definition

So consensus is very important. The first thing a theoretician does with apparently important problems is to formalize them, the better for reasoning about them. I’m going to keep the theory light here, but we need to get straight what constitutes a solution to the consensus problem. Thankfully, it involves only three properties.

We say that a node decides once it has reached its decision about the value it thinks everyone else agrees on. More formal definitions say that each node has an output register that, once the protocol has terminated, contains the value to agree upon. Writing to that register is the act of deciding.

Nodes propose values by suggesting them as the value to agree upon. The value that a node proposes is considered pre-determined – the protocol should not mandate the node to propose any particular value (we’ll see exactly why shortly).

With those two definitions in mind, and given a set N of nodes in a distributed system, we’ll consider a consensus protocol correct if and only if:

1. Agreement – all nodes in N decide on the same value
2. Validity – the value that is decided upon must have been proposed by some node in N
3. Termination – all nodes eventually decide

Agreement is an easy one to understand: we can’t really call something consensus if no consensus has been achieved. We might consider weakening this requirement to say that only a majority of nodes have to be in agreement, but that doesn’t actually change the strength of the problem or its required solutions much.

Validity is a bit less intuitive – and seems kind of obvious at the same time. The problem with the agreement property is that it trivially allows protocols that just instruct every node to decide a default value, no matter what the actual state of the network. For example a database commit protocol that always voted “don’t commit”, no matter what the transaction, satisfies the agreement condition. Validity ensures that these protocols don’t pass muster.

Termination is also a reasonable requirement, but it bears thinking about why it’s needed – agreement only constrains how nodes decide, not if they decide. We need termination to ensure that the protocol actually does something useful, otherwise again we’d be open to trivial solutions that just don’t do anything – satisfying agreement and validity.

Note that we’ve not said anything – yet – about tolerance to failures. Designing a protocol that works even when participants may crash and fail will turn out to be crucially difficult, so we’ll be revisiting the effect of failures later.

## Two-Phase Commit

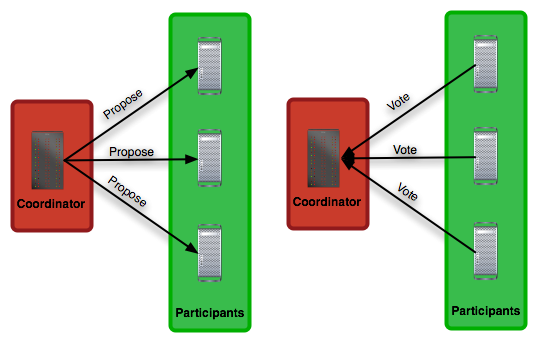
Simple solutions to the consensus problem seem obvious. Think about how you would solve a real world consensus problem – perhaps trying to arrange a game of bridge for four people with three friends. You’d call all your friends in turn and suggest a game and a time. If that time is good for everybody you have to ring round and confirm, but if someone can’t make it you need to call everybody again and tell them that the game is off. You might at the same time suggest a new day to play, which then kicks off another round of consensus.

This is a very natural way – although slightly simplified, omitting several human-specific optimizations – to achieve consensus. We can identify two steps in the process:

1. Contact every participant, suggest a value and gather their responses
2. If everyone agrees, contact every participant again to let them know. Otherwise, contact every participant to abort the consensus.

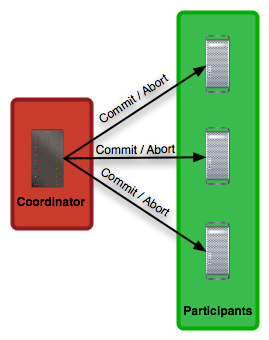
This skeleton describes possibly the simplest, most often used consensus algorithm called two-phase commit, or 2PC. As its name suggests, 2PC operates in two distinct phases. The first proposal phase involves proposing a value to every participant in the system and gathering responses. The second commit-or-abort phase communicates the result of the vote to the participants and tells them either to go ahead and decide or abort the protocol.

The process that proposes values is called the coordinator, and does not have to be specially elected – any node can act as the coordinator if they want to and therefore initiate a round of 2PC.

[](http://the-paper-trail.org/blog/wp-content/uploads/2010/01/tpc-fault-free-phase-1.png)

*Figure 1: Two-phase commit, fault-free execution, phase one.*

Observe that the consensus here is in regard to whether or not to accept the value proposed by the coordinator, not on the value itself. So the nodes are not achieving consensus about what that value should be, they are achieving consensus on whether or not to accept that value. This is a binary variable: yes or no, or commit or abort. Participants have no mechanism by which to say “actually, I’d rather we voted on x than the proposed y” – if they want to do that they have to initiate their own round of 2PC.

[](http://the-paper-trail.org/blog/wp-content/uploads/2010/01/tpc-fault-free-phase-2.png)

*Figure 2: Two-phase commit, fault-free execution, phase two.*

Does 2PC, as described, without worrying about failures, solve the consensus problem? Agreement is easily shown: every node decides the value proposed by the coordinator in phase one if and only if they are told do so by the coordinator in phase two. The coordinator sends the same decision to every node, so if one is told to decide a value they all are.

Validity is also satisfied. 2PC does not trivially commit or abort – it aborts in every case except the one where every node agrees to commit. In all cases, the final value decided upon has been voted on by at least one node.

Finally termination is guaranteed if every node is guaranteed to make progress and eventually return its vote to the coordinator, which then eventually communicates them to every node. Note that even asynchronous models guarantee this property in the absence of failures: eventually every message is processed and all responses are sent. There are no loops in the specification of 2PC, so no way for it to continue executing forever.

Therefore, 2PC does offer a solution to consensus. It has the benefit of being quite efficient – the number of messages exchanged is 3n for n nodes; it’s hard to see how that could be improved upon greatly.

However, as I have strongly hinted earlier, there is a fly in 2PC’s ointment. If nodes are allowed to fail – if even a single node can fail – then things get a good deal more complicated.

## Crashes and Failures

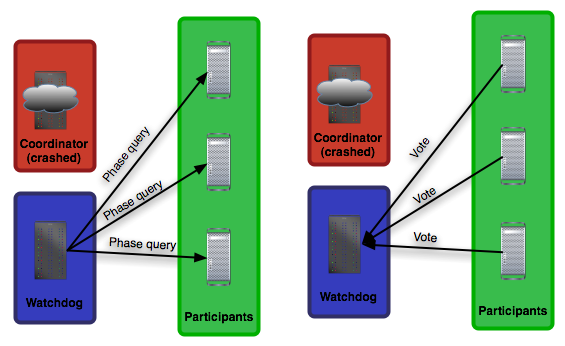
As discussed elsewhere on this blog, nodes can crash in several ways. Most simply, they can crash and never recover. This is the ‘fail-stop’ model of distributed system failure. Instead, nodes could crash and may at some later date recover from the failure and continue executing. This is the ‘fail-recover’ model. Most generally, a node could deviate from the protocol specification in completely arbitrary ways. This is called ‘Byzantine failure’, and designing protocols to cope with it is an active area of research, and rather difficult. We consider the most commonly used model of failure, fail-recover (it is also arguably true that this is the most commonly observed failure mode as well).

How could a node failure throw a spanner in the works for 2PC? To answer this, we have to look at every state the protocol can take, and consider what happens if either the coordinator or one of the participants crashes in that state.

In phase one, before any messages are sent out, the coordinator could crash. This doesn’t cause us too much worry, as it simply means that 2PC never gets started (and therefore, vacuously, works correctly). If a participant node crashes before the protocol is initiated then nothing bad happens until the proposal message fails to reach the crashed node, so we can deal with that failure later.

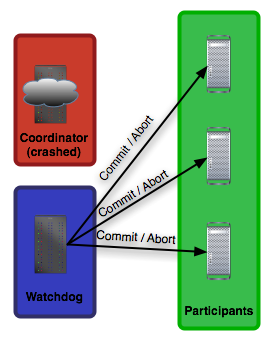
Now consider the protocol after some of the proposal messages have been sent, but not all of them. If the coordinator crashes at this point we’ll have some nodes that have received a proposal and are starting a 2PC round, and some nodes that are unaware that anything is going on. If the coordinator doesn’t recover for a long time, the nodes that received the proposal are going to be blocked waiting for the outcome of a protocol that might never be finished. This can mean that no other instances of the protocol can be successfully executed, as participants might have to take locks on resources when voting ‘commit’. These nodes will have sent back their votes to the coordinator – unaware that it has failed – and therefore can’t simply timeout and abort the protocol since there’s a possibility the coordinator might reawaken, see their ‘commit’ votes and start phase two of the protocol with a commit message.

The protocol is therefore blocked on the coordinator, and can’t make any progress. We can add some mechanisms to deal with this – and we’ll describe these below – but this problem of being stuck waiting for some participant to complete their part of the protocol is something that 2PC will never quite shrug off.

[](http://the-paper-trail.org/blog/wp-content/uploads/2010/01/tpc-coordinator-fails-phase-1.png)

*Figure 3: Two-phase commit, with coordinator failure, phase one.*

What we can do to paper over this crack, however, is to get another participant to take over the job of the coordinator once we realize that it has crashed. When a timeout occurs at some node, we can force that node to finish off the protocol that the coordinator started. This node can contact all the other participants, as in a phase one message, and find out which way they voted. This requires all nodes to keep in persistent storage the results of all 2PC executions, until they know that every other node has committed or aborted – otherwise if all nodes forget what they did for a given execution then the recovery node that takes over from the coordinator will have no way of recovering the state of the transaction. It’s also possible that only one node might know the result of the transaction if the coordinator fails in phase two before all nodes are told of the abort / commit decision.

[](http://the-paper-trail.org/blog/wp-content/uploads/2010/01/tpc-coordinator-fails-phase-2.png)

*Figure 4: Two-phase commit, with coordinator failure, phase two.*

However, if another participant node crashes before the recovery node can finish the protocol, the state of the protocol cannot be recovered. The recovery node can’t distinguish between all nodes having already voted to commit and the failed participant having committed (in which case it’s invalid to default to abort) and all nodes but the failed node having voted to commit and the failed node having voted to abort (in which case it’s invalid to default to commit).

The same kind of arguments apply – as broadly already covered – in phase two. If the coordinator crashes before the commit message has got to all participants, we need a recovery node to take over and safely guide the protocol to its conclusion.

The worst-case scenario is when the coordinator is itself a participant and grants itself a vote on the outcome of the protocol. Then a crash to the coordinator takes out both it and a participant, guaranteeing that the protocol will remain blocked, and as a result of only one failure.

The coordinator typically will log the result of any successful protocol in persistent storage, so that when it recovers it can answer inquiries about whether a transaction committed. This allows periodic garbage collection of the logs at the participant nodes to take place: the coordinator can tell nodes that no-one will try to recover a mutually committed transaction and that they can erase its existence from their log (that said, the log might be kept around to be able to recover a node’s state after a crash).

## Conclusions

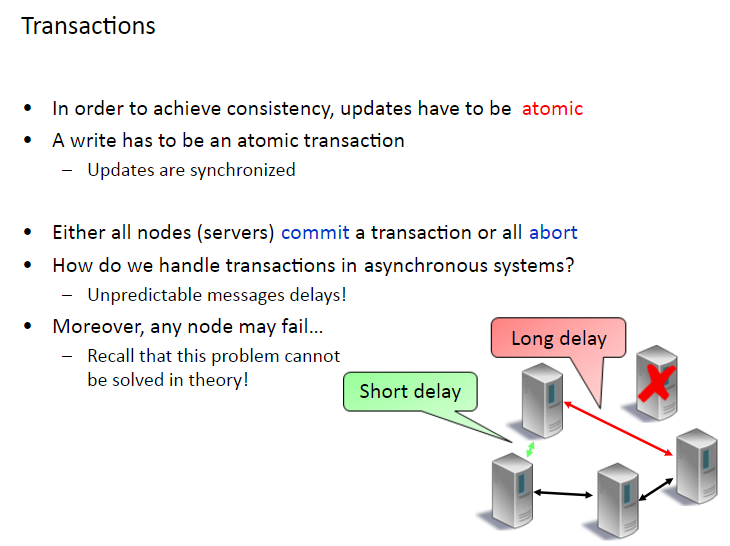
2PC is still a very popular consensus protocol, because it has a low message complexity (although in the failure case, if every node decides to be the recovery node the complexity can go to O(n^2)). A client that talks to the coordinator can have a reply in 3 message delays’ time. This low latency is   
very appealing for some applications.

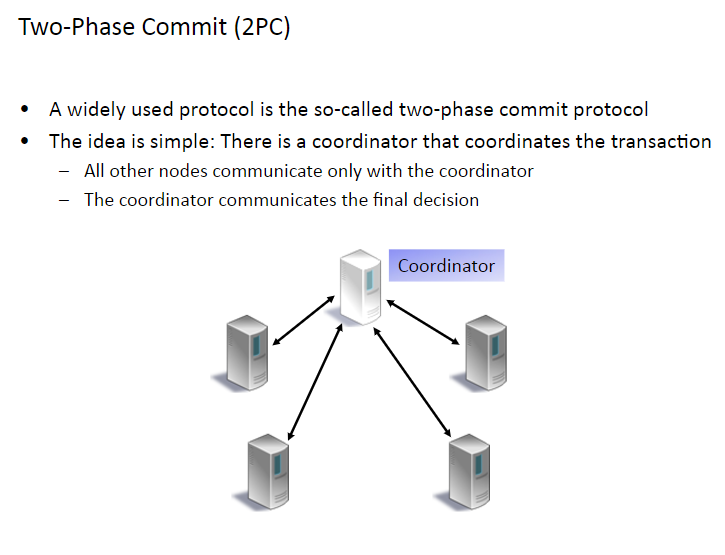
However, the fact the 2PC can block on coordinator failure is a significant problem that dramatically hurts availability. If transactions can be rolled back at any time, then the protocol can recover as nodes time out, but if the protocol has to respect any commit decisions as permanent, the wrong failure can bring the whole thing to a juddering halt.

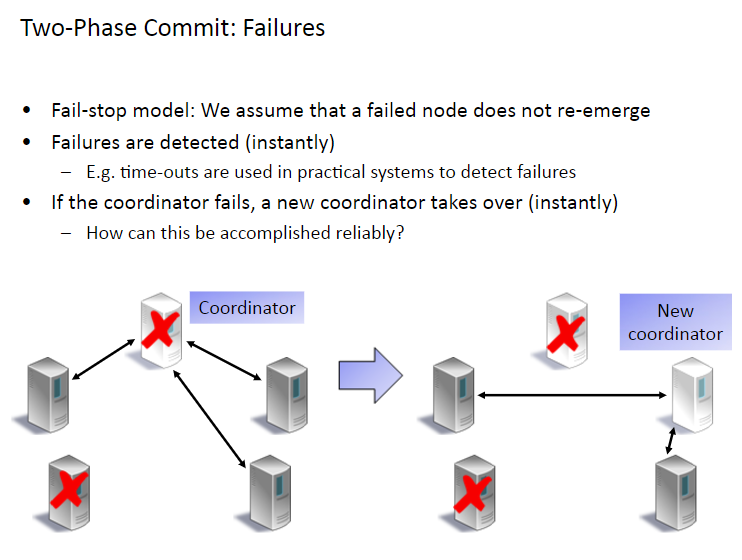
That said, there are still cutting-edge research projects that embrace 2PC. The most significant recently is [Sinfonia](http://research.microsoft.com/users/aguilera/papers/sinfonia-aguilera-sosp2007.pdf) (PDF), which won a best paper award at SOSP 2007. They build a kind of distributed memory on top of 2PC but make coordinator crashes recoverable from (via a time out and the intervention of a recovery node). Memory node crashes are considered ‘game over’ scenarios (but the memory nodes themselves may be highly replicated). It’s a good paper and I recommend a read.

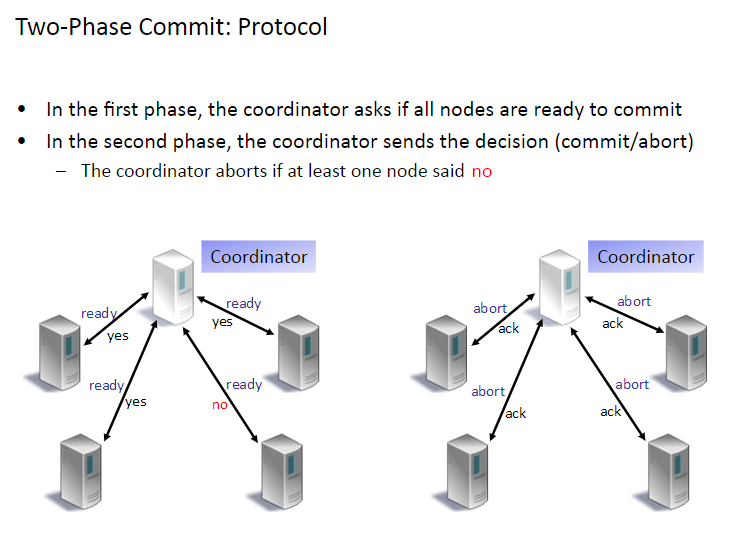
<http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/>

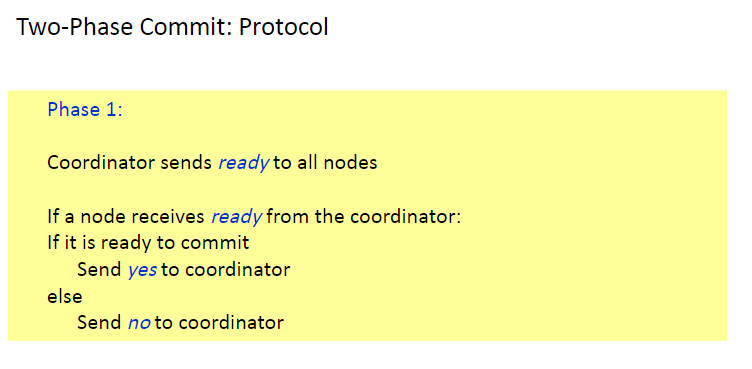
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

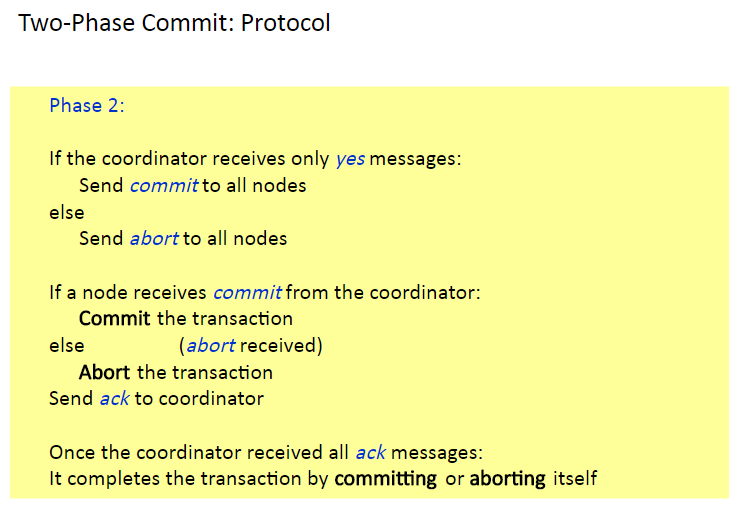


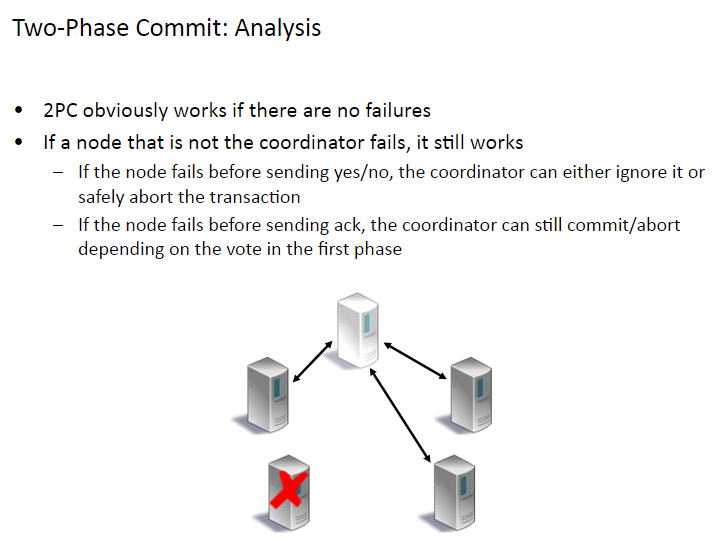


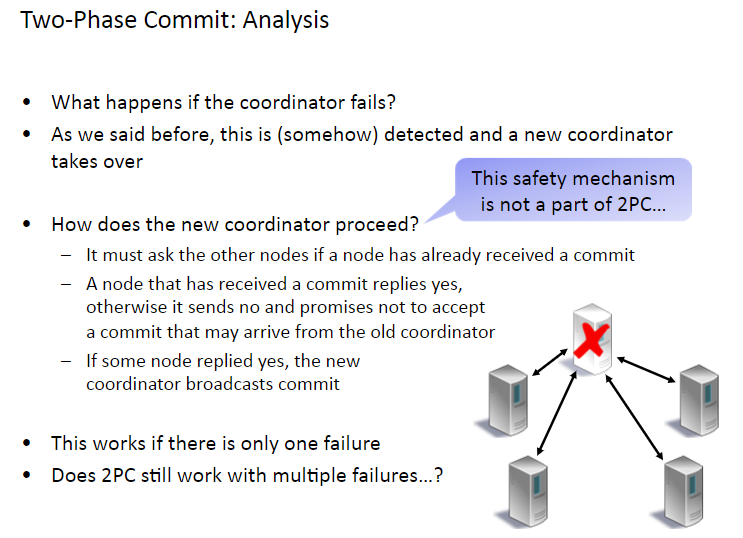


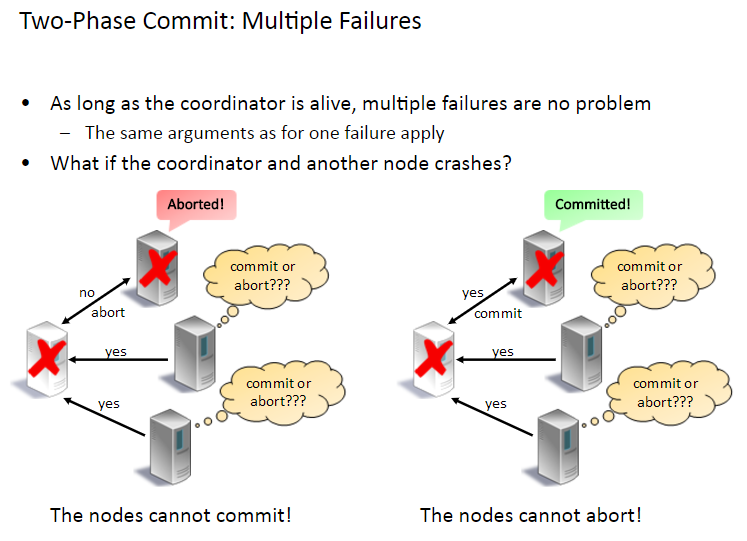


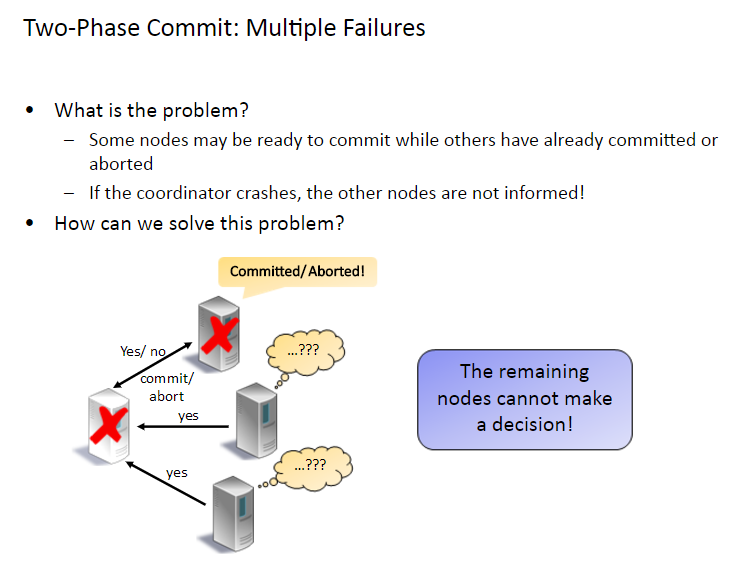




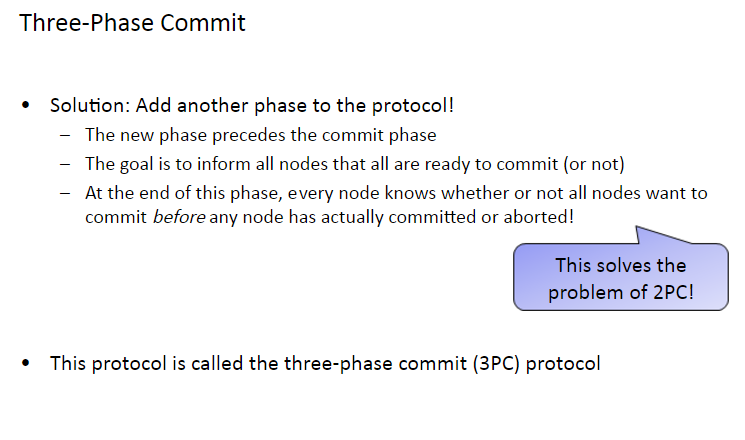


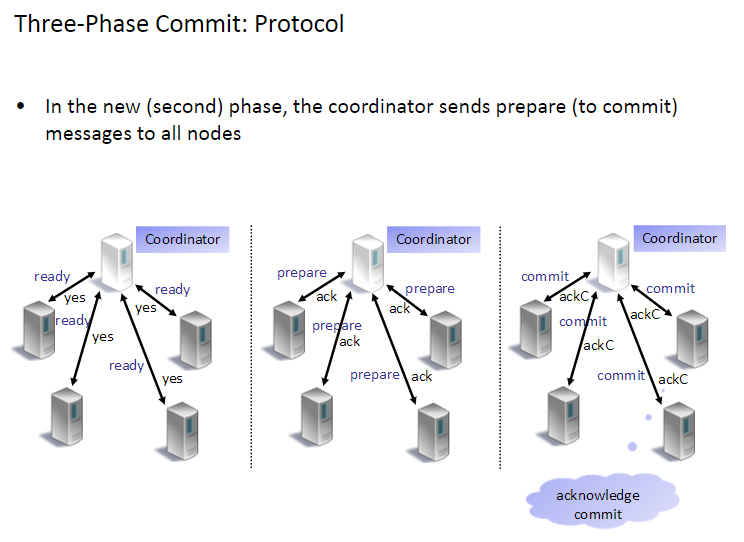


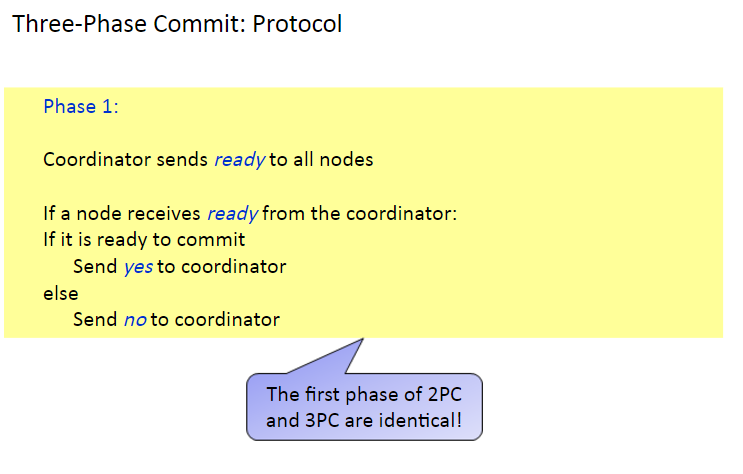




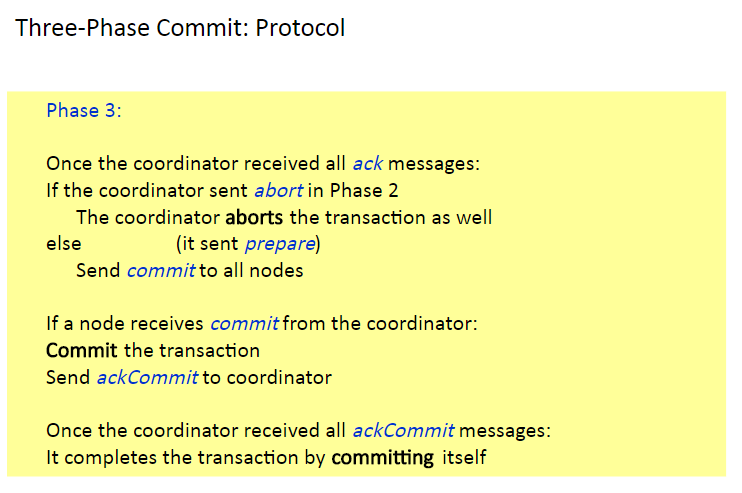
------------------------------------------------------------------------------------------------------------------------------------------

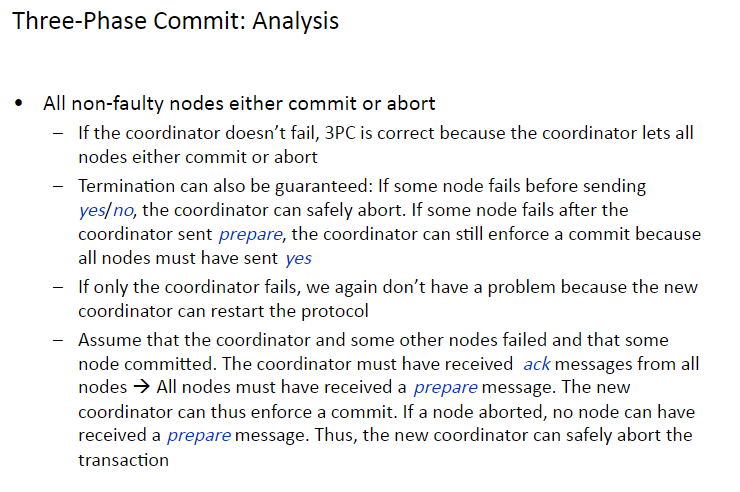


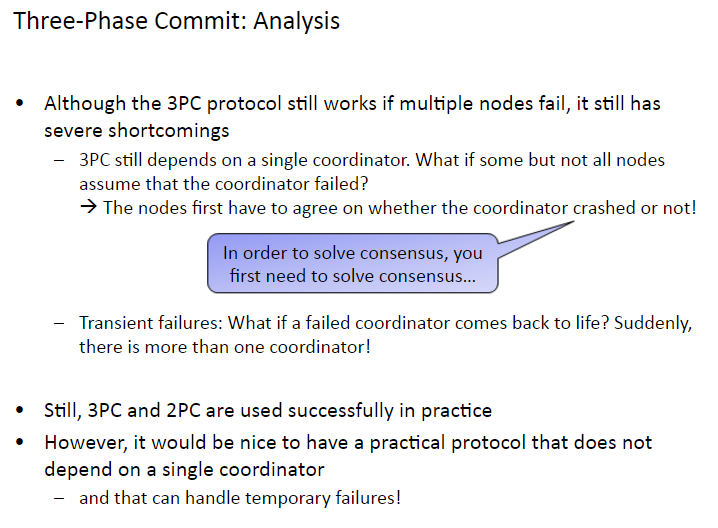












<http://slideplayer.com/slide/5371917/>