[**Two-Phase-Commit for Distributed In-Memory Caches**](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-distributed-in.html)

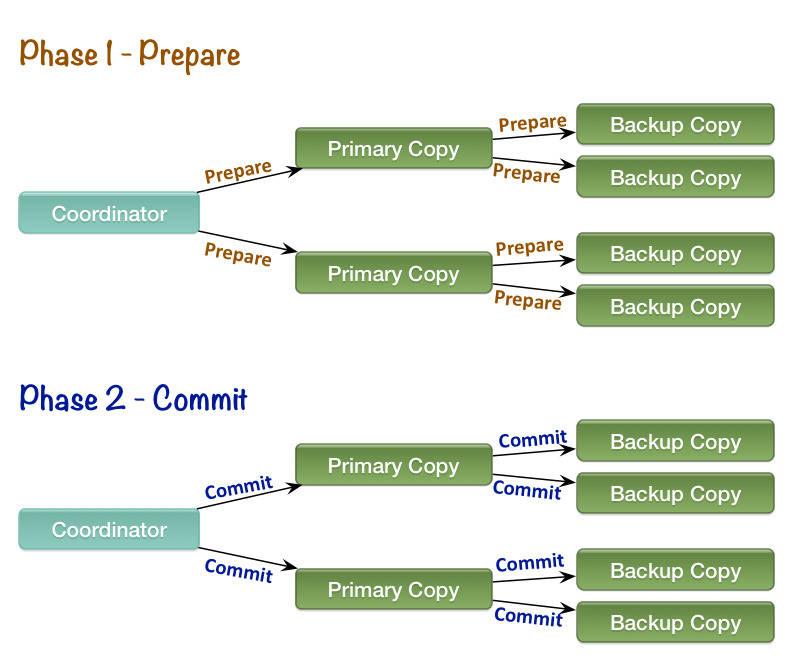
*2-Phase-Commit* is probably one of the oldest consensus protocols and is known for its deficiencies when it comes to handling failures, as it may indefinitely block the servers waiting in *prepare* state. To mitigate this, a *3-Phase-Commit* protocol was introduced which adds better fault tolerance at the expense of extra network round-trip message and higher latencies.   
  
I would like to extend these traditional quorum concepts into distributed in-memory caching, as it is particularly relevant to what we do at [GridGain](http://www.gridgain.org/). GridGain too has *2-phase-commit* transactions, but unlike disk-based persistent systems,***GridGain does not need to add a 3rd phase to the commit protocol in order to preserve data consistency during failures****.* 

*We want to avoid the 3-Phase-Commit protocol because it adds an additional network round-trip and has a negative impact on latencies and performance.*

In GridGain, the data is *partitioned* in memory, which in the purest form means that every key-value pair is assigned to a specific node within the cluster. If for example, we have 100 keys and 4 nodes, then every node will cache *100 / 4 = 25*keys. This way the more nodes we have, the more data we can cache. Of course, in real life scenarios, we also have to worry about failures and redundancy, so for every key-value pair we will have 1 primary copy, and 1 or more backup copies.  
  
Let us now look at the 2-Phase-Commit protocol in more detail to see why we can avoid the 3rd commit phase in GridGain.

### Two-Phase-Commit Protocol

Generally the *2-Phase-Commit* protocol is initiated whenever an application has already made a decision to commit the transaction. The *Coordinator* node sends a ***Prepare*** message to all the participating nodes holding primary copies (***primary nodes***), and the *primary nodes*, after acquiring the necessary locks, synchronously send the *"Prepare"*message to the nodes holding backup copies (***backup nodes***). Once every node votes *"Yes"*, then the *"Commit"* message is sent and the transaction gets committed.

[](http://1.bp.blogspot.com/-sS575_YsGS8/VAEh2o6qPtI/AAAAAAAACTg/F4TYvMRDzgQ/s1600/two-phase-commit.png)

*Figure 1: 2-Phase-Commit Protocol For Distributed In-Memory Cache*

#### Advantages of In-Memory

So, why does in-memory-only processing allow us to avoid the 3rd phase as opposed to persistent disk-based transactions? The main reason is that the in-memory cache is purely volatile and does not need to worry about leaving any state behind in case of a crash. Contrast that to the persistent architectures, where we need to worry whether the state on disk was committed or not after a failure had occurred.  
  
Another advantage of in-memory-only distributed cache is that the only valid vote for the *"Prepare"* message is *"Yes". T*here is really no valid reason for any cluster member to vote *"No"*. This essentially means that the only reason a rollback should happen is if the *Coordinator* node crashed before it was able to send the *"Prepare"* message to all the participating nodes.   
  
Now, that we have the ground rules settled, let's analyze what happens in case of failures: 

#### Backup Node Failures

If a backup node fails during either *"Prepare"* phase or *"Commit"*phase, then no special handling is needed. The data will still be committed on the nodes that are alive. GridGain will then, in the background, designate a new backup node and the data will be copied there outside of the transaction scope. 

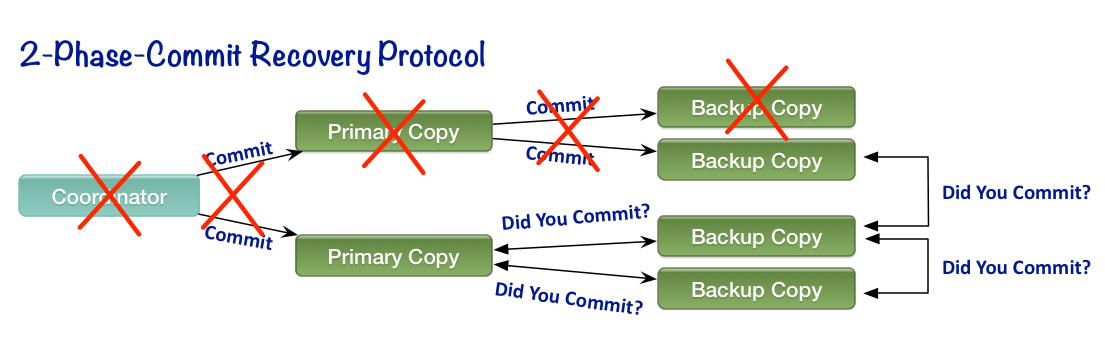
#### Primary Node Failures

#### **If a primary node fails before or during the "Prepare" phase, then the coordinator will designate one of the backup nodes to become primary and retry the "Prepare" phase. If the failure happens before or during the "Commit" phase, then the backup nodes will detect the crash and send a message to the Coordinator node to find out whether to commit or rollback. The transaction still completes and the data within distributed cache remains consistent.**

#### Coordinator Node Failures

#### **Failure of the Coordinator node becomes a little tricky. Without the Coordinator node, neither primary nodes, nor backup nodes know whether to commit the transaction or to rollback. In this case the**"Recovery Protocol"**initiates, and all the nodes participating in the transaction send a message to every other participating node asking whether the"Prepare" message was received. If at least one of the nodes replied "No", then the transaction will be rolled back, otherwise the transaction will be committed.**

Note that after all the nodes received the *"Prepare"* message, we can safely commit, since voting *"No"* is impossible during the *"Prepare"* phase as stated above.  
  
It is possible that some nodes will have already committed the transaction by the time they have received the *Recovery Protocol* message from other nodes. For such cases, every node keeps a backlog of completed transaction IDs for a certain period of time. If there is no ongoing transaction with given ID found, then the backlog is checked. If the transaction was not found in the backlog, then it was never started, which means that the failure had occurred before the *Prepare* phase was completed and it is safe to rollback.  
  
Since the data is volatile, and does not leave any state after the crash, it does not really matter if any of the primary or backup nodes also crashed together with the coordinator node. The *Recovery Protocol* still works the same.

[](http://4.bp.blogspot.com/-l1cZdynJsOg/VAEuw1g5mgI/AAAAAAAACT8/zs2UYwxGjVY/s1600/two-phase-commit-failure.png)

*Figure 2: Recovery Protocol*

#### Conclusion

Note that we are able to fully recover the transaction without introducing the 3rd commit phase mainly because the data in distributed in-memory caches is volatile and node crashes do not leave any state behind.  
  
The important advantage of the *2-Phase-Commit Recovery Protocol*in GridGainover the *3-Phase-Commit* is that, in the absence of failures, the recovery protocol does not introduce any additional overhead, while the *3-phase-commit* adds an additional synchronous network roundtrip to the transaction and, therefore, has negative impact on performance and latencies.   
  
GridGain also supports modes where it works with various persistent stores including transactional databases. In my next blog, I will cover the scenario where a cache sits on top of a transactional persistent store, and how the *2-Phase-Commit* protocol works in that case

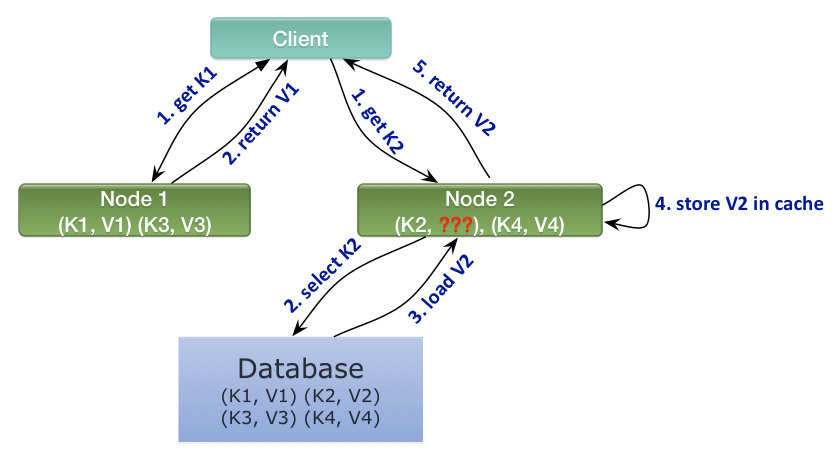
<http://gridgain.blogspot.in/2014/09/two-phase-commit-for-distributed-in.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
[**Two-Phase-Commit for In-Memory Caches - Part II**](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-in-memory-caches.html)

Generally, persistent disk-oriented systems will require the additional *3rd phase* in commit protocol in order to ensure data consistency in case of failures. [In my previous blog](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-distributed-in.html) I covered why the *2-Phase-Commit* protocol (without 3rd phase) is sufficient to handle failures for distributed in-memory caches. The explanation was based on the open source [GridGain](http://www.gridgain.org/) architecture, however it can be applied to any in-memory distributed system.  
  
In this blog we will cover a case when an in-memory cache serves as a layer on top of a persistent database. In this case the database serves as a primary system of records, and distributed in-memory cache is added for performance and scalability reasons to accelerate reads and (sometimes) writes to the data. Cache must be kept consistent with database which means that a cache transaction must merge with the database transaction.

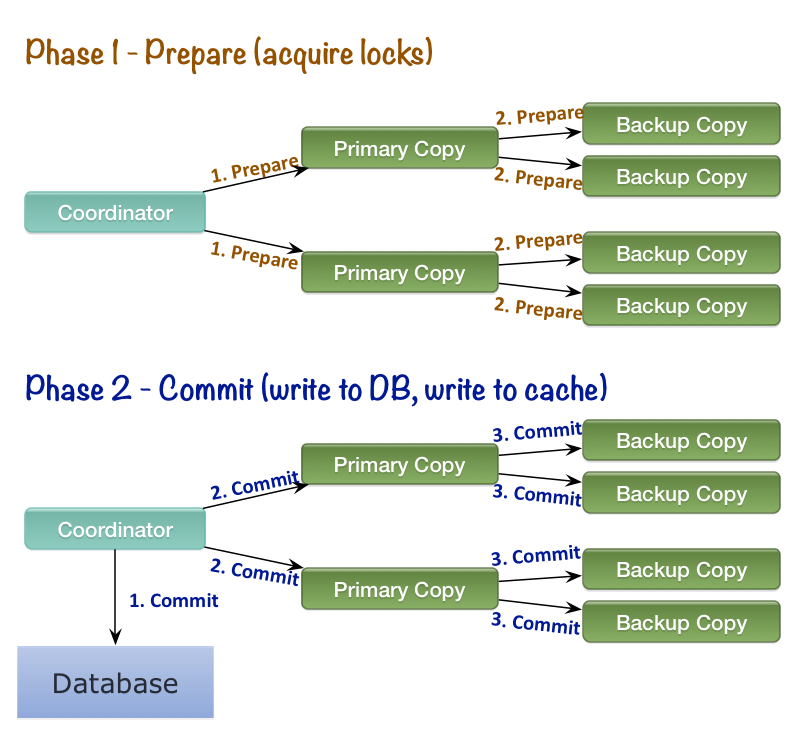
***When we add a persistent store to an in-memory cache, our primary goal is to make sure that the cache will remain consistent with on-disk database at all times.***

In order to keep the data consistent between memory and database, data is automatically loaded on demand whenever a read happens and the data cannot be found in cache. This behavior is called *read-through*. Alternatively, whenever a write operation happens, data is stored in cache and is automatically persisted to the database.  This behavior is called *write-through*. Additionally, there is also a mode called *write-behind* which batches up the writes in memory and flushes them to the database in one bulk operation (we will not be covering this mode here).

[](http://1.bp.blogspot.com/-CXgN_n0aiok/VA1p3B_rJ7I/AAAAAAAACUg/UbUy0zJyMrA/s1600/read-write-through.png)

*Figure 1: Read-through operation for K2*

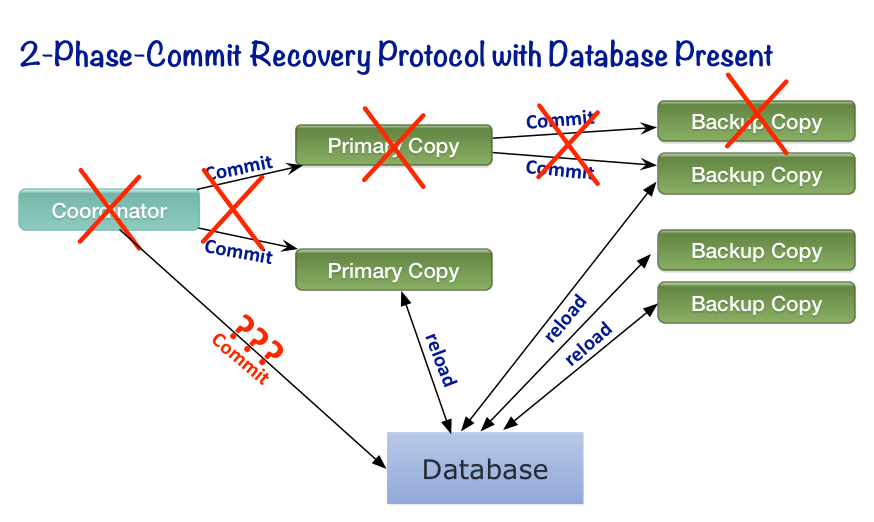
When we add a persistent store to the *2-Phase-Commit* protocol, in order to merge cache and database transactions into one, the *coordinator* will have to write the transactional changes to the database before it sends the *Commit* message to the other participants. This way, if database transaction fails, the coordinator can still send the *Rollback* message to everyone involved, so that the cached data will remain consistent with database. Figure below illustrates this behavior.

[](http://4.bp.blogspot.com/-liRS0bTloMA/VA1v7cpJ7hI/AAAAAAAACUw/Qw9ATGZBCiU/s1600/2pc-store-ok.png)

*Figure 2: Two-Phase-Commit with In-Memory-Cache and Database*

Handling failures is actually more straight forward whenever a database is present rather than when it is not. We always assume that the database must have the utmost up-to-date copy, and it is acceptable to reload data from the database into cache whenever in doubt (see *Figure 1*). Just like in [my previous blog](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-distributed-in.html), the most challenging scenario here is when the *coordinator* node crashes (potentially together with other nodes), because in this case we cannot tell whether it crashed before it was able to commit to the database or not. Other failure scenarios are handled the same way with database present as without.

Whenever we cannot tell whether the database commit had happened or not, we can simply **reload** the relevant data from database into cache upon committing the transaction. This effectively ensures that database and cache always remain in consistent state.

[](http://2.bp.blogspot.com/-8K2VVbGo9s4/VA5rHve-qcI/AAAAAAAACVc/hweFjf4NDZk/s1600/2pc-store-failure.png)

### Conclusion

When working with in-memory caches, we can always manage to keep the data within transactions consistent by slightly enhancing the standard *2-Phase-Commit* protocol. The main advantage of in-memory vs. disk is that failure handling does not introduce any additional overhead, and we do not need to add an expensive *3rd phase* to the *2-Phase-Commit*protocol in order to keep caches consistent with databases in case of failures

http://gridgain.blogspot.in/2014/09/two-phase-commit-for-in-memory-caches.html