A Highly Scalable Transactional Multi-Tier Platform as a Service.

**Deliverable Identifier:** D1.1 Architecture of Transactional Management

**Public Disclosure Date:** 20th November 2011

**Authors:** Ricardo Jiménez-Peris (UPM), Marta Patiño-Martínez (UPM), Iván Brondino (UPM)

**Document version:** 1.0

Project funded by the European Commission under the Information Society and Media DG 7th Framework Programme
# Table of Contents

Executive Summary .......................................................................................................................... 3

1. Overview ...................................................................................................................................... 4
   1.1. Main Contributions and Achievements .............................................................................. 4
   1.2. Progress with Respect to the State of the Art ................................................................. 4
   1.3. Alignment with Project Goals ........................................................................................... 4
   1.4. Structure of the Deliverable ............................................................................................... 4

2. Introduction .................................................................................................................................. 4

3. Global Architecture ..................................................................................................................... 6

4. The Commit Server .................................................................................................................... 6

5. The Snapshot Server ................................................................................................................ 7

6. Conflict Managers ..................................................................................................................... 8

7. Loggers ....................................................................................................................................... 9

8. Transaction Managers .............................................................................................................. 9

9. Transaction Life Cycle ............................................................................................................... 11

10. Conclusions ............................................................................................................................. 13
Executive Summary

This document presents the architecture of the transactional management subsystem of CumuloNimbo. The transactional management subsystem is the core of the innovation made by CumuloNimbo and is the one that enables to attain ultra-scalable transactional processing. It is based on the increasingly popular snapshot isolation correctness criteria in which transactions see a snapshot of the database at starting time of the transaction.

The transactional management system consists of a number of components, namely, commit server, snapshot server, loggers, conflict managers, and transaction managers. The commit server provides commit timestamps and the snapshot server provides the start timestamps. Transaction managers manage a subset of transactions. Loggers provide durability guarantees. The snapshot server determines the visibility of data guaranteeing that only coherent snapshots are visible. Conflict managers take care of aborting conflicting transactions.

The core breakthrough is the separation of durability and visibility of consistent data. This has been achieved by pre-assigning commit timestamps to transaction managers so they can commit transactions in parallel and by regulating the visibility of data by means of the snapshot server that only enables to observe consistent snapshots, that is, snapshots for which data from all previously committed transactions are visible.
1. Overview

1.1. Main Contributions and Achievements

The contribution and achievement has been the definition of the architecture for transactional management that is able to very large update transaction rates.

1.2. Progress with Respect to the State of the Art

The main progress with respect to the state of the art is the removal of the main bottlenecks in transactional processing to attain ultra-scalability of transactional processing.

The core breakthrough lies in the separation of durability and visibility of coherent data. This has been achieved by pre-assigning commit timestamps to transaction managers so they can commit transactions in parallel and by regulating the visibility of data by means of the snapshot server that only enables to observe consistent snapshots, that is, snapshots for which data from all previously committed transactions are visible.

1.3. Alignment with Project Goals

The transactional management subsystem is the core of the CumuloNimbo PaaS providing holistic transactional processing for the whole software stack of the CumuloNimbo PaaS.

1.4. Structure of the Deliverable

The deliverable starts with an introduction to transactional processing in Section 2 that provides some basic background on transactional processing and snapshot isolation. Then, Section 3 positions the transactional processing with respect the global architecture of CumuloNimbo and introduces the different components used for transactional management. Sections 4 to 8 describe the different components. Sections 9 presents the conclusions.

2. Introduction

Information systems are the core component of any business or organization. Information is typically stored in a database, mainly a relational database that is accessed using SQL. Databases are accessed using transactions and guarantee the so called ACID properties: Atomicity, Consistency, Isolation and Durability.

A transaction either succeeds (commits) or fails (aborts), that is, it is atomic. If transactions are correct (the application code has no bugs), after execution of a transaction, the data will remain consistent. The isolation property establishes that if a set of transactions is executed concurrently, their final effect is like if they were executed sequentially in some order (serializability). Concurrency control protocols are in charge of implementing the isolation property.
Serializability is the strongest isolation level. Other isolation criteria have been defined, being today snapshot isolation among the most popular criteria. Snapshot isolation provides similar guarantees to the ones provided by serializability but it allows more concurrency among transactions. Snapshot isolation only disallows concurrent writes, whilst serializability additionally disallows concurrent reads and writes on the same data items. In the context of the CumuloNimbo project the isolation provided by the transactional management is snapshot isolation.

Finally, if a transaction commits, its changes will remain even in the advent of failures. This programming model is easy to understand and relieves programmers of most of the burdens related to concurrency, persistency of data and fault-tolerance. The model has been so successful that it is not only used in the context of databases, for instance application servers offer transactional management at higher levels (e.g., for objects) and it has been recently proposed as the way to program multi-threaded applications to exploit the power of multi-cores.

A key problem in transactional data management is how to scale out the processing. That is, how to use an increasing number of computational resources to process a higher rate of transactions. In this deliverable we describe the transactional management provided in the context of the CumuloNimbo project whose main goal is to process a high number of transactions per second.
3. Global Architecture

The transaction management subsystem (see Figure 1) is holistic and provides coherent transactional semantics across the whole stack independently of which interfaces applications are using, JEE, SQL or HBase.

![Figure 1: The transactional management system in the context of the CumuloNimbo architecture](image)

The transactional management system has been decomposed in different components to enable to scale each component separately in a composable manner. It consists of the following set of components: commit server, snapshot server, transaction manager, logger, conflict manager and data manager. The following sections introduce each of the components.

4. The Commit Server

The commit server is a component that provides timestamps, called commit timestamps that are globally ordered. Commit timestamps are provided to the transaction managers. The commit server can be reactive or proactive. In reactive mode, transaction managers request commit timestamps to the commit server. The commit server can provide batches of commit timestamps to individual transaction managers or to the plurality of them. In proactive mode, the commit server gets notifications about the update transaction rates that each transaction manager processes and uses this information to set the sizes of the batches of timestamps assigned to each transaction manager proportional to the observed or forecast rates.
The commit server does not need to communicate directly with all transaction managers. For instance, in reactive move, it can simply receive periodically a single message with the last update rate of each transaction manager and produce the assignment of commit timestamp batches. This message can be produced by one of the transaction managers or any other component. In this way the amount of work that the commit server performs per update transaction is neglectable. Since the commit server has to do some work per update transaction it can become one of the ultimate bottlenecks of the system. By minimizing this work to the current level, it will be able to reach very high update transaction rates.

The provided commit timestamps are used by the transaction managers to label a committed transaction or they are discarded if unused.

5. The Snapshot Server

Under snapshot isolation a transaction reads the latest committed data at the time the transaction started. That is, the transaction only observes a snapshot of the data at starting time. Writes performed by a transaction become visible only after the transaction commits.

Therefore, when a transaction starts it is assigned a snapshot of the database, that is, the set of committed writes that it will observe. When a read only transaction (transactions that do not perform any write) commits, no special action is taken. When an update transaction (a transaction that has performed at least one write) is committed, it is assigned an order in the sequence of committed transactions. That is, for any two committed update transactions the order in the sequence determines which one committed before the other. Snapshot isolation is implemented by committing update transactions sequentially and assigning new started transactions a snapshot corresponding to the current sequence of committed update transactions.

Additionally, concurrent transactions are not allowed to update common data items. Conflicts are either solved when they happen by aborting one of the conflicting transactions or at commit time by aborting a committing transaction that is concurrent to a conflicting transaction that has committed before.

This kind of transactional processing creates contention due to several reasons. The main reason is that commit processing and the synchronization between the start of new transactions and the commit of completing ones become a bottleneck. In CumuloNimbo, this bottleneck is removed. The ultimate bottleneck that will remain in CumuloNimbo will be able to reach very high rates of update transactions preserving full transactional consistency.

The snapshot server is the component that provides the latest snapshot (set of committed updates) at which transactions can be started with isolation guarantees. This snapshot is represented by the commit timestamp of the latest update transaction included in the
snapshot and is also called snapshot timestamp. The snapshot server keeps track of which are the commit timestamps that have either been used or discarded. To achieve it, the snapshot server receives timestamps from the transaction managers. These timestamps might have been discarded (that is, produced by the commit server but not used to timestamp any update transaction) or committed (that is, the writes of the update transaction are durable and readable from the data manager with a snapshot with a timestamp equal or later to its commit timestamp). The snapshot server provides snapshot timestamps in non-decreasing order. Any snapshot timestamp, \( S \), provided by the snapshot server guarantees that the snapshot server has received all timestamps lower than \( S \). The snapshot server periodically garbage collects a set of timestamps such that the timestamps in the set are contiguous, they are older than the rest of stored timestamps, and there is another timestamp contiguous to the earliest one of them.

6. **Conflict Managers**

The conflict manager is a component that detects and handles conflicting transactions. There are may be several conflict managers. Each conflict manager takes care of handling the conflict of a set of keys. Each key is, therefore, handled by a single conflict manager. A conflict manager receives notifications about: writes over data with keys it manages (update notification), completion of update transactions (either commit or abort), and the oldest start timestamp among active transactions, that is, transactions that have started but not completed. The conflict managers are typically collocated with other components such as the object cache and the No SQL data stores.

An update notification contains the key of the written datum, the identifier of the transaction and the start timestamp of the transaction. Upon an update notification the conflict manager, checks whether there is any update associated to the same key from a concurrent transaction (two transactions are concurrent if both have started and none has completed, or if the start timestamp of one transaction is in between the start and commit timestamp of another transaction). If there is a conflicting update, the conflict manager notifies the abort of one of the two conflicting transactions, keeps the information from the transaction not being aborted, and discards the information of the aborted transaction. If there is no conflicting update, it stores the key and the start timestamp of the update transaction.

A transaction completion notification contains the transaction identifier, the transaction outcome (commit or abort) and if the transaction committed, its commit timestamp. A conflict manager upon receiving a transaction completion notification of the abort of a transaction removes all the updates stored with the received transaction identifier. A conflict manager upon receiving a commit transaction completion notification adds the commit timestamp to all the stored updates with the same transaction identifier.

A notification of the oldest active transaction contains the start timestamp of the oldest active transaction in the system (later it is stated which components issues this
7. Loggers

The logger is a component that is in charge of guaranteeing the durability of updates of committed transactions. A log request contains the set of writes, the transaction identifier and the transaction commit timestamp. The logger might be replicated, that is, a set of loggers that store the same set of updates. The logger can be configured with different durability levels that determine when to answer back from the log request.

If a single logger is used two possible durability levels can be choose: the log request is considered stored once the log request is in memory or when the logger has stored in persistent storage the request.

If the logger is replicated the durability level determines which fraction of loggers should have the log request stored in memory and which fraction of loggers should have the log request stored in persistent storage. When the log request is durable according to the configuration, the logger notifies about the durability of the transaction indicating its transaction identifier. This notification is used by transaction managers to reply back to clients that the transaction has been committed.

8. Transaction Managers

A transaction manager manages the lifecycle of a subset of the transactions. There may be several transaction managers. Each transaction is managed by one transaction manager.

The transaction manager supports the operations: connect, start transaction, read item, write item, read, write, commit and abort. A client application gets associated to one transaction manager. The connect, start, commit transaction operations do not have parameters. The read item operation has as parameter the key of the item to be read. The write item operation has as parameters the key of the item to be written and the item itself. The read operation has as parameter a read-only query. The write operation has as parameter an update query (that is, a query that performs one or more updates: insertions, deletions and modifications, and possibly some reads).

Each application interacts with the transactional data management system via a transaction manager. The client application first connects to one of the transaction managers. The application can perform any number of transactions with the connected transaction manager. Each transaction is performed by first invoking the start operation, then invoking one or more read item, write item, read and write operations and finally invoking either commit or abort.

A transaction manager gets periodically a set of commit timestamps from the commit server, and also gets periodically snapshot timestamps from the snapshot server. When an application connects to a transaction manager, the transaction manager associates to the
application a unique connection identifier that is stored locally and returned as result of
the connect operation. Upon invocation of the start operation (implicit or explicit
depending on the application interface) by an application a transaction manager
associates to it a unique transaction identifier, associates the transaction to one query
engine and selects the latest snapshot timestamp received from the snapshot server, and
stores in a local transaction table the transaction identifier, the connection identifier, the
start timestamp and the query engine identifier.

A transaction manager upon receiving a read item or read operation invocation from an
application forwards the operation to the query engine associated to the transaction. The
read item operation can also be served from any item cache (e.g. the distributed object
cache, or the cache maintained by the No SQL data store). The transaction manager upon
receiving the result from the read operation forwards the results to the application.

A transaction manager upon receiving a write item or write operation invocation from a
client application, it forwards the operation to the query engine associated to the
transaction. Upon receiving the result from the write operation, if it was successful, it
stores the returned updates associated to the transaction identifier and returns the result
to the application. If the operation was unsuccessful due to a conflict detected by a
conflict manager, the transaction manager aborts the transaction, removing all the local
information associated to the transaction and stores the transaction identifier with the
abort status in the transaction table.

A transaction manager upon receiving an abort operation invocation from a client
application forwards it to the associated query engine. Afterwards, it removes all the
associated information to the transaction and stores the transaction identifier with the
abort status in the transaction table.

A transaction manager upon receiving a commit operation invocation from an application
it checks whether the transaction performed any write or not. If the transaction did not
perform any write, the information about the transaction is removed, the commit
operation is invoked on the query engine, and finally the control returned to the
application. If the application performed at least one write, the transaction manager takes
one unused commit timestamp, associates it to the transaction being committed and
stores it in the transaction table. Then, in parallel it sends all the stored updates to one
logger and invokes the commit operation on the associated query engine. When the logger
replies, the transaction manager stores that the transaction is durable in the transaction
table and replies to the application that the transaction has been committed. When the
commit operation returns from the query engine, the transaction manager stores in the
transaction table that the transaction is readable. A readable transaction is a transaction
for which its updates can be read with a timestamp equal or later to its commit
timestamp. That is, it guarantees that all the private versions of the updates of that
transaction have become public versions and tagged with the commit timestamp at all
components in the system. When a transaction is both durable and readable, then the
transaction manager stores in the transaction table that the transaction is committed.
In order to guarantee session consistency, that is, that a client observes all the updates performed by previously submitted and committed transactions by it an additional mechanism is required. A transaction manager upon receiving the start transaction from a client that has executed previously an update transaction delays the start of the transaction until a snapshot timestamp equal or later than its commit timestamp is sent by the snapshot server. This provides session consistency, that is, a client will read previously committed values by itself.

Each transaction manager periodically notifies to the snapshot server the commit timestamps that have been discarded and the commit timestamps of committed transactions. This communication can be made directly or via a third party to reduce the number of components interacting directly with the snapshot server.

Periodically, one of the transaction managers collects from all other transaction managers the local oldest start timestamp for all the transactions they manage. Upon receiving these timestamps, it obtains the oldest one and notifies all conflict managers and all data managers that this timestamp is the oldest snapshot of an active transaction. This mechanism enables all tiers to purge versions that will not be needed by any active transaction.

9. Transaction Life Cycle

In this section we show how transactions are processed cooperatively by all components of the transaction manager. The transaction starts due to an action of the application that leads to an explicit or implicit transaction start (see Figure 2) invocation in the transaction manager. Then, the transaction manager gets a start timestamp for the transaction from the snapshot server (or has obtained previously). The start timestamp is stored in the transaction context. Finally, the transaction start is registered in the local JBoss transaction manager.

![Figure 2: Transaction start](image-url)
During transaction execution read operations use the start timestamp to determine from which snapshot to read data. During execution of write operations the transaction identifier is used to track to which transaction private versions belong. Also each write invokes the corresponding conflict manager and if a conflict is detected the transaction is rolled back.

The transaction commit (see Figure 3) is also triggered by the application directly or indirectly that results in an invocation of the commit operation on the transaction manager. The transaction manager gets a commit timestamp for the transaction (or has obtained it in advance) and stores it in the transactional context. The cache stub propagates the commit operation on all involved cache nodes. The transaction manager also propagates the commit operation to the SQL engine that performed the transaction operations. The SQL engine propagates it to the involved NoSQL data store nodes.

![Figure 3: Transaction commit](image)

The transaction abort is invoked explicitly by the application or is triggered by a conflict manager upon detecting a conflict. The abort operation is run by the transaction manager (see Figure 4) that basically propagates the abort operation to the cache stub and the SQL engine. The cache stub propagates the abort operation to all involved cache nodes. The SQL engine does so over all involved NoSQL data store nodes.
10. Conclusions

The architecture of the transactional management subsystem has successfully removed the main bottlenecks of transactional processing. With the developed architecture it is expected that CumuloNimbo will be able to reach a million update transactions per second definitively solving the scalability of transactional systems. It does with full transparency to applications, without introducing any constraints, neither requiring any a priori knowledge. This will enable a very wide adoption of the system.