CoherentPaaS

Coherent and Rich PaaS with a Common Programming Model

ICT FP7-611068

CoherentPaaS Global Architecture

D10.1

March 2014

Document Information

Scheduled delivery	31.03.2014
Actual delivery	30. 04.2014
Version	1.0
Responsible Partner	UPM

Dissemination Level:

PU	Public
PP	Restricted to other programme participants (including the Commission)
RE	Restricted to a group specified by the consortium (including the Commission)
CO	Confidential, only for members of the consortium (including the Commission)

Revision History

Date	Editor	Status	Version	Changes
15.02.2014	Ricardo Jiménez	Draft	0.1	TOC
21.02.2014	Ricardo Jiménez	Draft	0.2	First complete draft
28.02.2014	Marta Patiño	Draft	0.3	Second draft
8.03.2014	Marta Patiño	Revised	0.4	First peer review
15.03.2014	Marta Patiño	Revised	0.5	Second peer review
20.03.2014	Ricardo Jiménez	Final	1.0	Final version

Contributors

Ricardo Jiménez Peris (UPM)
Marta Patiño (UPM)
Iván Brondino (UPM)
Valerio Vianello (UPM)

Internal Reviewers

Pavlos Kranas NTUA
Martin Kersten MonetDB

Acknowledgements

Research partially funded by EC 7th Framework Programme FP7/2007-2013 under grant agreement n° 611068.

More information

Additional information and public deliverables of CoherentPaaS can be found at: http://coherentpaas.eu

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DoW	Description of Work
EC	European Commission
PM	Project Manager
PO	Project Officer
WP	Work Package

Table of Contents

List of Figures

1. Executive Summary

This deliverable provides the global architecture of CoherentPaaS. It provides a summary of the vision of the project, its subsystems, and how they are integrated and interact among each other.

2. Global view

The Project ambition is to remedy the current situation for building scalable cloud applications. Application developers today have to use a combination of different cloud data management technologies that are totally disparate. For instance, a graph database, a key-value data store together with a relational database. This totally lack of coordination across cloud data stores creates many difficulties. The main three issues are lack of data coherence across different data stores, the large development efforts to develop queries across cloud data stores that should be programmed manually, and the difficulty of performing performance debugging in applications across several cloud data stores.

The CoherentPaaS project addresses these difficulties with three contributions. The data coherence issue is addressed by providing holistic transactions for all data stores. Depending on each data store, this might mean some might simply integrate their local transactional processing with the holistic one, and for others to fully implement transactional processing. The result is a single transactional manager that enables transactions across any combination of data stores providing full ACID semantics. This transactional manager leverages the ultra-scalable transactional processing from CumuloNimbo guaranteeing that it can reach any required scale without creating a bottleneck in transactional processing.

The issue of the effort for performing queries across data stores is addressed by developing a global query language and its engine able to run queries written in (a subset of) SQL across any set of data stores. Since some of the data stores have a specialized API or query language that is crucial to attain high performance, as for instance, graph databases, the proposed query language can also embed subqueries fully written in the native query language/API of the cloud data store. In this way, it is possible to combine the full power of the native query languages with the full expressivity of SQL to query across data stores and get the best of the two worlds. The common query language engine is also integrated with the holistic transactions to guarantee full transactional consistency for transactions involving global queries.

The issue of the difficulty of doing performance debugging in complex applications involving multiple cloud data stores is addressed by a combination of contributions. First of all, we will provide a framework to instrument transparently applications to do fine grain monitoring of the time spent on each application method and on each invocation to cloud data stores. In this way, it will become possible to find out where the performance bottlenecks are happening at the application level. Additionally, the cloud data stores themselves will be monitored at fine grain level as well. This monitoring will be correlated with the application performance. The monitoring capabilities will enable application developers to perform fine tuning of the cloud data stores and find out what configuration is the most adequate for their workloads.

The different subsystems are depicted in Figure 1. The holistic transaction management includes all the components for the ultra-scalable transactional processing. It interfaces with all cloud data stores, NoSQL, SQL and CEP. The common query language engine is depicted close to the top between the applications and the cloud data stores. It interfaces with all cloud data stores and also with the holistic transactional management. The x-ray monitoring subsystem provides the monitoring framework that is integrated with all subsystems and the deployed applications. There is a component in charge of platform management taking care of elastic reconfiguration decisions and the deployment of the platform in a cloud environment. Finally, on top of the figure the addressed use cases are depicted. These use cases have been selected because they have the potential to take advantage of the use of multiple cloud data management technologies integrated in the project.

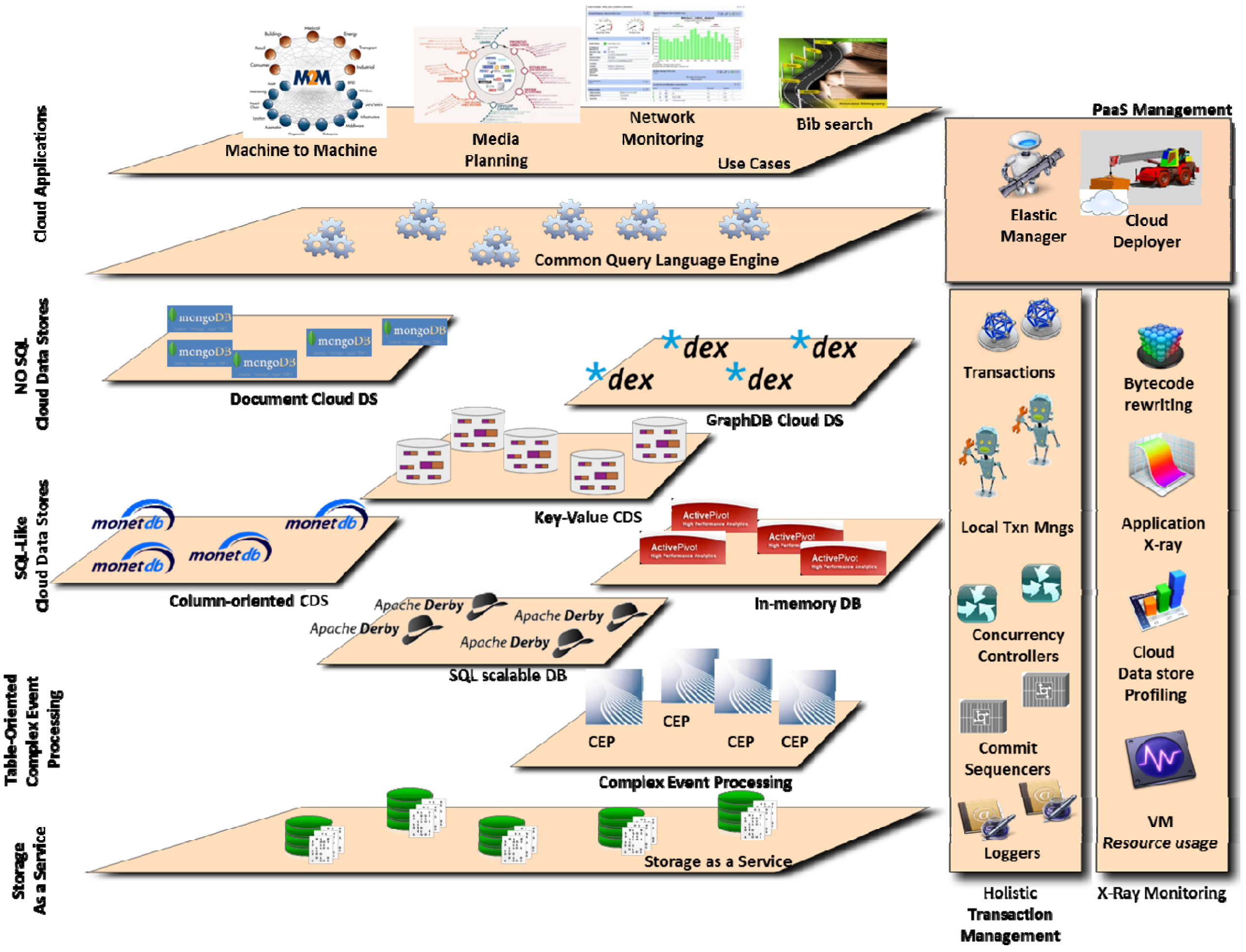

Figure 1: CoherentPaaS Global Architecture

3. CoherentPaaS Subsystems

3.1. Holistic Transactions

One of the three major goals of the project is to provide transactional coherence for updates across cloud data stores. For this purpose, the project plans to leverage the CumuloNimbo ultra-scalable transactional processing and extend it to support multiple cloud data stores.

The planned design for this extension was homogeneous, more concretely, it was planned that all the transactional processing would be performed by the holistic transactional manager. However, some of the cloud data store vendors participating as partners in the project have started to see value in this offering and are willing to develop the transactional functionality themselves in order to be able to exploit this feature also individually for their cloud data store. For this reason the design of the holistic transactions has been extended beyond the initial conception at proposal time to be flexible an allow both the case in which transactional processing is fully delegated to the holistic transactional manager and the case in which the transactional processing is performed locally and only interacts with the holistic transactional processing when required.

The holistic transactional manager provides snapshot isolation as consistency criteria [Berenson95]. The functionalities from the holistic transactional manager that need to be integrated with the cloud data stores are the following:

- Conflict management. This functionality lies in checking whether there are updates by two different concurrent transactions over the same key.
- Logging. The updates of a transaction (writeset) should be made durable before notifying the client application about the commit of the transaction.
- Making public the transaction updates. Once the commit of the transaction is successful the results of the transaction should be made public so they become potentially readable.
- Recovery. In the advent of a failure, it has to be found out which transactions are committed and which aborted. Resources kept by aborted transactions should be released. Committed transactions that are in doubt should be redone from the log.
- Removal of obsolete versions. Some versions of the data will never be read again. In order to release its space, the transactional manager finds out which is the oldest transaction and its start timestamp. Any version for which another version exists that equal or lower than that timestamp can be removed since it cannot be read by any transaction.
- Regulating which version to read by means of a start timestamp.
- Regulating the version with which to label committed updates when making them public.

The holistic transactional manager consists of a set of different roles and many of these roles are distributed servers themselves. In this deliverable we do not detail how the holistic transactional manager is architected internally, since this is done in the corresponding deliverable. In here, we treat as an abstract server that is accessed via a proxy that we call local transactional manager. Cloud data stores are provided with a local transactional manager instance that is collocated with all instances of the cloud data store that are capable of bracketing transactions. This is done typically at the client proxy of the data store, but it can also be done at the server side of the cloud data store. It should be noted that the life cycle of a transaction should be managed at the side where there is the notion of the session for the transaction and all interactions with the holistic transactional manager should be performed by the same instance of the component on the cloud data store side.

3.1.1. Conflict Management

Conflicts that have to check are only write-write conflicts, since transactional isolation is based on snapshot isolation [Berenson95]. A CoherentPaaS cloud data store can handle conflict management in two ways. The first way is fully delegating conflict management to the CoherentPaaS holistic transactional manager. In this case, the cloud data store has to check periodically by means of the conflict manager service all the keys that have been updated by all the active transactions. Each active transaction is checked independently. The keys should be unique for a given data store so they should be qualified with the table name and the key in the case of table oriented data stores and in general with any sequence of qualifiers to guarantee that they are unique.

For cloud data stores that are handling transactional semantics themselves, they might decide to handle the conflict management within the cloud data store. The cloud data store should guarantee that the transaction does not have any conflict before starting the commit phase of the transaction. If there is a conflict, the cloud data store should rollback the transaction.

3.1.2. Logging & Recovery

In order to guarantee durability the holistic transactional manager performs logging. The holistic transactional recovery is redo only, and therefore, it only stores postimages of the updated data to execute redo actions during recovery. Again there are two ways to handle logging, it can be fully delegated to the holistic transactional manager, or it can be handled by both the cloud data store and the holistic transactional manager.

If the logging is fully delegated to the holistic transactional manager, then when the client triggers the commit of a transaction the cloud data store should provide the writeset (all the updates performed by the transaction in the cloud data store) to the holistic transactional manager. The writeset can have any representation since it will only be handled by the cloud data store and it is opaque for the holistic transaction manager. However, it should be self-sufficient, it should contain all the information for the cloud data store so it can redo the transaction upon recovery.

If the logging is not fully delegated to the holistic transactional manager because the cloud data store is performing transactional management, it should be taken into account that the definitive logging will only be the one of the holistic transactional manager. The cloud data store can log itself locally the writeset, but then it has still to provide as writeset to the holistic transactional manager the minimal information so it can perform recovery. Typically, in this case, the information logged at the holistic transactional manager will be a handle that upon recovery will enable the cloud data store to find locally the redo record and apply it. It should be noted that the log record locally stored at the cloud data store in this case will behave as a prepare record in a traditional two-phase commit, therefore, not definitive till the holistic transaction manager reports that the transaction is durable at holistic level.

Upon recovery a cloud data manager will first perform local recovery to the extent it is handling it and then, it will perform the recovery with the holistic transaction manager. After the holistic recovery, it will be able to start giving service to clients.

3.1.3. Versioning

After making durable the updates at holistic level, a local transaction manager will instruct all data stores involved in the transaction to make the updates public. Till then, versions will be kept as private versions (invisible to other transactions). This will be done providing the commit timestamp associated to the versions. This commit timestamp should be used for labelling versions.

When a transaction starts it is provided with the start timestamp that indicates the version of the data that a transaction should read. More concretely, it should read the version with the highest timestamp such that is lower or equal than the start timestamp. Data stores should be able to use the externally provided timestamp and compare it with the one used to label versions to find efficiently the versions to be read.

Another crucial issue for the transactional integration is garbage collection of obsolete versions (not to be confused with Java Garbage collection that happens at the level of JVM runtime system). A version of a data item that is not the last one can only be read by a transaction, if the transaction start timestamp is equal or higher than the commit timestamp of the data item and the last version of the data item is higher than the start timestamp of that transaction. If all active transactions have a start timestamp that is strictly higher than the commit timestamp of a later version, this version will never be read. Since obsolete versions occupy space, they should be removed.

The integration with the holistic transactional manager is materialized by periodically reporting to the cloud data stores about the lowest start timestamp among active transactions. With this information, cloud data stores should be able to remove obsolete versions in an efficient way. This is very important since obsolete version removal can become a heavy process.

3.2. Common Query Engine

The common query language MdbQL is designed to be capable of querying multiple heterogeneous databases (relational and NoSQL) within a single query that can potentially contain nested sub-queries written in the native query languages of the queried data stores.

MdbQL is based on the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations. MdbQL keeps its common data model schema-less in order to simplify data integration and avoid the trouble of describing multiple evolving data models. Thanks to its common data model all the datasets retrieved from the data sources match this common model.

MdbQL is able to embed subqueries in the native query language/API of the queried data store. In order to support functional transformations, MdbQL queries can contain constructs of the programming language Python. Python is quite rich in data types, easy to use, and rich in standard libraries and widely used.

MdbQL introduces the notion of "table expression" an expression that returns a table (relation – a structure, compliant with the common data model). Table expressions are used to represent nested queries and typically query a particular data store. Table expression can be SQL, Python code snippets producing relations and native expressions querying a data store in its native query language. A table expression can be assigned a name and a signature, thus, becoming a "named table expression", which can then be used in the FROM clause of the query as any other regular relation.

MdbQL also introduces the notion of "action expressions", which can contain data operations on a data store. They are instantiated by an EXECUTE clause. For an SQL data store an action expression contains SQL DML command, while for a NoSQL data store an action expression contains invocations to the data store's native query API. An action expression can instantiate named table expressions which gives the flexibility for a single query to retrieve data from one (or more) data store, perform transformations on it and then use it to update another data store. A single MdbQL command can perform data manipulation against several data stores. In order to guarantee ACID consistency the common query engine is also integrated into the holistic transactional support, providing all-or-nothing semantics for updates across multiple data stores.

The common query engine is integrated with the data stores by means of a mediator/wrapper architectural model. Wrappers transform queries expressed in the common query language into native queries of the data store, and also transform the result sets of queries into results of the common data model. Wrappers also provide information about the data store schemas. The mediator transforms queries expressed in the common language into queries for the wrappers and integrates the wrapper queries' results. The mediator also centralizes the information provided by wrappers in a global schema. The interface between the common query engine and the data stores if through the wrappers.

The common query engine is integrated with the holistic transactions as any other CoherentPaaS data store. The main difference between the common query engine and the rest of the data stores is that the common query engine itself does not store data and acts as a kind of delegated transactional manager with respect to holistic transactions.

3.3. X-Ray Monitoring

The X-Ray monitoring subsystem provides a fine-grained analysis and real-time monitoring of applications deployed on CoherentPaaS. This subsystem will enable to transparently instrument cloud applications to perform a detailed analysis of the cost of each request performed by the application. This detailed analysis will include a detailed analysis of where the time is spent within the cloud application. But it will also include a detailed analysis of the cost of each query performed against any of the CoherentPaaS data stores. This support will provide an invaluable tool to CoherentPaaS application developers enabling developers to tune and improve the performance of applications. X-Ray will also be crucial to enable database administrators to tune the configuration of each of the CoherentPaaS data stores and measure in a very detailed manner the performance impact of each configuration change. The profile information will include detailed information about the query cost, selectivity, number of updates, transactional cost, etc.

The integration of the X-Ray monitoring subsystem with the CoherentPaaS data stores is performed via agents that collect performance metrics and detailed cost information from the data stores. The agents also collect resources usage metrics from the underlying operating system/hypervisor via the sigar library [Sigar].

3.4. Data Stores and CEP

One of the core components of CoherentPaaS architecture are the cloud data stores and CEP system. Two kinds of data stores are dealt with: SQL-like data stores and NoSQL data stores. Among the SQL-like data stores there is a SQL database, providing a SQL OLTP engine (the SQL query engine has been extracted from Derby DB [Derby]), a column-oriented SQL data store, MonetDB [MonetDB] and, an in-memory MDX active database, ActivePivot [ActivePivot]. On the NoSQL side another three technologies are supported: a graph database, DEX [DEX], a document-oriented data store, MongoDB [MongoDB] and, a key-value data store, HBase [HBase]. Finally, as CEP technology to be integrated into CoherentPaaS, Storm [Storm] has been selected.

Data stores are integrated with the cloud application via their proxy clients. For instance, the client proxy of the Derby SQL database is a JDBC driver. For HBase, there is an HBase client proxy. It is worth noting, that the common query engine also has its client proxy. This proxy acts as any other data store proxy for the application. Again, it is a special data store since it does not contain data itself but enables to access other data stores.

CEP is a special case. The CEP does not store persistent data. Also CEP queries are very different in nature since they are continuous, while for persistent data stores they are point in time queries. This means that there is a big impedance mismatch between both kinds of technologies. Solving this impedance mismatch is another of the challenges that CoherentPaaS aims to solve. CoherentPaaS provides a bi-directional integration. On one hand, CEP queries may access the persistent data stores, and on the other hand we want to enable applications to use the output of CEP queries in the same way as any other data store. The CEP queries will be able to access the persistent data stores via a new kind of CEP data operators that will enable to correlate events with the persistent data stores via the common query engine. CoherentPaaS cloud applications will be able to access the output of CEP queries without having to use the CEP API, thanks to the CEP materialization operators that allow materializing the output of a CEP query as a SQL table that can be accessed by applications as any other SQL table.

The CEP is integrated with the holistic transactions in a peculiar way due to the lack of explicit transaction bracketing. On one hand, events are guaranteed to observe a consistent snapshot as they are transformed through a CEP query. This consistency guarantees that an event will observe the same snapshot along the whole CEP query.

Also batches of events will be dealt transactionally when they update persistent data stores or when they are materialized at the output of a CEP query.

3.5. PaaS Manager

The PaaS manager provides a centralized management for the whole platform. It enables to make deployments of the platform in an automated manner. It provides a dashboard that provides monitoring information of the deployment and that it is connected with the x-ray monitoring system to provide detailed profiling information of deployed applications. It also provides a console that enables to administer all the deployed subsystems in a particular deployment. It also provides holistic elastic management decisions for those subsystems that delegate the elasticity management support, a subset of all the data stores technologies, namely, Derby, HBase and the transactional subsystems.

4. References

[ActivePivot] http://quartetfs.com/en/products/activepivot/high-performance-analytics
[Berenson95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10.
[Derby] http://db.apache.org/derby/
[DEX] http://www.sparsity-technologies.com
[HBase] https://hbase.apache.org
[MonetDB] http:// https://www.monetdb.org
[MongoDB] https://www.mongodb.org
[Sigar] http://www.hyperic.com/products/sigar
[Storm] http://storm.incubator.apache.org