PROTEUS

PROTEUS

Scalable online machine learning for predictive analytics and real-time interactive visualization

687691

D3.4 Updateable-state management requirement definition

Lead Author: Daniel Higuero, Alvaro Agea
With contributions from: Juan Manuel Tirado
Reviewer: David Piris

Deliverable nature:	R
Dissemination level: (Confidentiality)	CO
Contractual delivery date:	1/12/2016
Actual delivery date:	25/11/2016
Version:	1.1
Total number of pages:	29
Keywords:	Cache, Distributed Databases, big data

Abstract

The current document describes a proposal for a distributed key-value caching mechanism named PEaCH that can be leveraged by the different components of the Proteus project. The cache design supports several clients accessing the same key and provides eventual consistency guarantees on the update of the state among all involved clients. The cache uses Conflict-free replicated data types as the underlying structures to support the values stored by the clients, and as such provides method to define how a value should be updated and how conflicts should be resolved. Additionally, the document provides a list of formal requirements on the solution to be developed.

Document Information

IST Project Number	687691	Acronym	PROTEUS
Full Title	Updateable-state management requirement definition		
Project URL	http://www.proteus-bigdata.com/		
EU Project Officer	Martina EYDNER		

Deliverable	Number	D3.4	Title	Updateable-state management requirement definition
Work Package	Number	WP3	Title	Scalable Architectures for both batch data and data streams

Date of Delivery	Contractual	M12	Actual	M12
Status	version 1.1		final x	
Nature	report x demonstrator □ other □			
Dissemination level	public □ restricted x			

Authors (Partner)				
Responsible Author	Name	Alvaro Agea	E-mail	alvaro@novelti.io
	Partner	LMBDP	Phone	(+34) 687 472 135

Abstract (for dissemination)	
Keywords	Cache, Distributed Databases, big data

Version Log			
Issue Date	Rev. No.	Author	Change
16/11/2016	1.0	Alvaro Agea	First Version.
25/11/2016	1.1	Daniel Higuero	Revisited version.

Table of Contents

List of figures

Abbreviations

PEACH: Proteus ElAstic caCHe

ACID: Atomicity, Consistency, Isolation, Durability

ANSI: American National Standards Institute

API: Application programming interface

BSD: Berkeley Software Distribution

CmRDT: Operation-based CRDT

CRDT: Conflict-free replicated data type

CvRDT: State-based CRDT

HDFS: Hadoop Distributed File System

LRU: Less recently used

NFS: Network File System

SQL: Structured Query Language

SEC: Strong Eventual Consistency

TTL: Time-to-live

P2P: Peer-to-peer

1 Introduction

The Proteus project defines a technological framework that provides scalable ready-to-use online machine learning algorithms of general use in streaming applications. Considering that in order to scale the computation capabilities the algorithms should be able to be executed in a distributed environment, the need for shared structures become a common point when facing new algorithm implementations.

For this reason, as part of Proteus it has been decided to design and develop a caching system that satisfies the requirements of a distributed processing framework such as Apache Flink [FLINK] when executing the SOLMA library developed also as part of Proteus. While the cache offers a generic and extensible solution for caching in distributed system, the main usage within Proteus is the storage of intermediate results produced by the machine learning algorithms. For example, consider an iterative training process of a model that is executed with data being ingested in streaming. In this scenario, a cache may maintain the updated model allowing each distributed node that participates in building the model to read and update it. Once the model has been trained, it may be also maintained in cache to be applied to new incoming data.

The next paragraphs describe the main objectives of the current deliverable and the document structure.

1.1 Document objectives

The objective of the present document is to describe the proposal for a distribute cache mechanism that can be leveraged by the different technical components of the Proteus project. The main objective can be divided into:

- Introduce the state-of-the-art with respect to distributed caching solutions.
- Introduce the concept of Conflict-free replicated data types or CRDT.
- Provide an overview of the proposed caching solution and the associated requirements.

1.2 Document structure

The document is structured as follows. First, Section [2] overviews the current state-of-the-art in the context of caching solutions in distributed systems. Next, Section [3] describes the main features of the proposed cache and shows different use cases that may benefit for such a solution. Section [4] lists the cache requirements. Finally, Section [5] summarizes the conclusions and Section [6] contains the list of bibliographical references.

2 State of the art

This section describes the different supported caches in the market and an overview of the CRDT applications.

2.1 Distributed cache systems

The problem of efficient caching in distributed system is an ongoing challenge. The following paragraphs overview different solutions that address this problem using different technological approaches, in particular we analyze Memcached [2.1.1], Redis [2.1.2], Alluxio [2.1.3], Apache Ignite [2.1.4], and Oracle Coherence [2.1.5]

2.1.1 Memcached

Memcached [1] is a distributed in-memory key-value solution. Each piece of data is composed of a key, an expiration time, optional flags and raw data. Memcached is designed as a distributed hash where data is stored in a node according to the hash value of their key. Queries can be solved in O(1) by simply computing the hash of the queried key. The simplicity of the queries makes possible to serve millions of queries per second. Memcached is designed as an LRU cache discarding data after their time has expired.
Memcached does not provide any I/O mechanism to ensure data persistence. This means that the amount of data that can be stored is limited by the amount of available memory. When a node is running out of memory it simply discards the oldest entry following the LRU policy. Extensions such as MemcachedDB [2] provides storage solutions to persist data onto disks. Memcached has revealed itself as a lightweight and easy-to-deploy distributed cache solution. Some memory allocation issues have been pointed out particularly for homogeneously size distributed entries. Solutions such as Cachelot [3] propose similar implementations that try to improve memory allocation.

2.1.2 Redis

Redis (REmote DIctionary Server)[4] is an open source (BSD licensed), in-memory data structure store solution. Redis is designed as a distributed key-value store solution that supports several data structures such as strings, hashes, lists, sets, etc. This solution is written in ANSI C with a small system footprint and a remarkable performance and scalability.
Redis offers two persistence solutions. The first solution periodically dumps the key-value dictionary to a secondary storage device. A second solution logs incoming commands to a file as soon as possible. This approach makes possible to restore values by reproducing the set of executed commands. However, both solutions may lead to unstable solutions if no replication is used. For this purpose, Redis supports replication using a master-slave schema. In this schema, a master is in charge of guaranteeing the coherence for a set of slaves. The simplicity of this approach makes possible to bring Redis to a large scale deployments while maintaining resilience and performance.
Redis offers several wrappers in different programming languages and is particularly well-maintained for Linux environments. This project has become an extremely popular solution for the deployment of large scale distributed caches and is the cornerstone of other projects such as Celery [5] and well-known platforms such as Twitter or Facebook among others.

2.1.3 Alluxio (former Tachyon)

Alluxio [6] is an open-source distributed memory-centric filesystem. The solution itself is designed to improve performance on large scale analytics platforms and intensive computing solutions based on platforms such as HDFS or Spark. The main advantage of Alluxio is a reduction of the computation time avoiding access to secondary disks through an in-memory computation. Additionally, Alluxio offers a common interface that permits different systems to access data in the same way. Alluxio has a layering design where existing applications can be connected to read or write data. It offers a traditional filesystem API and key-value storage. The open design supports several platforms such as Spark, Hadoop, Flink, Hbase or NFS.
The architecture is based on a master-slave paradigm. A single master (or two masters in case high availability is activated) is in charge of a set of workers. The master, maintains the system metadata and the coherence of the system. Workers manage local resources such as memory or hard disks, serve requests to clients and read/write data into blocks. However, only the master can map blocks to their corresponding files. Finally,

clients expose the filesystem API and connect the final user with the master node and the read/write operations performed by workers.

2.1.4 Apache Ignite

Apache Ignite [7] is a high-performance distributed in-memory platform designed for transactions on large-scale data sets. The solution offers a homogeneous platform for applications to access data in the same manner independently of the employed backend storage. The system allocates a key-value storage distributed across the nodes of the cluster. Data is stored in a key-value fashion that can be transparently accessed by clients independently of where the keys are allocated. Apache Ignite offers fault tolerance with support for SQL queries and ACID transactions. Queries are distributed across the nodes containing copies of the keys involved in the query and returning partial results that are merged afterwards. Ignite can be used as an intermediate memory caching tier for Hadoop and Spark based solutions without modifying existing code. From an architectonic point of view, Ignite permits to add cluster nodes on the fly facilitating horizontal scalability.

2.1.5 Oracle Coherence

Oracle coherence [8] is a Java-based in-memory data grid designed to have larger performance than traditional databases. This platform is designed as a full peer-to-peer solution with nodes creating a fully connected mesh among them offering linear scalability with fault tolerance. The designed connectivity among nodes ensures no single point of failure and no single points of bottleneck. Data vision is globally the same. This simplifies the system access which can be done using any entry point reducing latencies. Data can be accessed using a specific query language that distributes queries across the nodes with data involved into the query.

2.2 CRDT

The main challenge of proposing a caching mechanism for a distributed system environment is related to the consistency guarantees of the proposed solutions. Several consistency levels [9] are defined in the literature:

- **Strong consistency.** After an update is completed to a shared element, all subsequent accesses to that element must return the updated value.
- **Weak consistency.** They system does not guarantee that subsequent accesses to a shared element that has been updated return the last value for that object. There is usually a period of time where the system may return old values to some read request, and the new value to newest ones.
- **Eventual consistency.** The eventual consistency is a specific class of weak consistency, where the system assures that if no further updates are performed to the shared element, all replicas will converge to the same updated value. As a basic example of this type of mechanism, consider a classic TTL caching policy where entries are assigned a TTL value until they are discarded the new values are read.

Other consistency levels such as casual consistency, session consistency and others are specific subtypes of the aforementioned ones and remain outside the scope of this section. Given the fact that a caching distributed system that provides strong consistency guarantees requires some type of centralized transactor component, and that this transactor will in fact limit the scalability of the system, alternative solutions have been defined. In particular, Conflict-free replicated data types [CRDT] define a new paradigm to implement distributed system with shared elements under strong eventual consistency guarantees. The CRDT is provides Strong Eventual Consistency (SEC) guarantees with is a combination of an eventually consistent design with the fact that all replicas that have received the same updates must have converged to the same state.

The next subsections overview the different types of CRDT classes and methods. Additionally, a set of implementations of CRDT are analyzed as part of the state-of-the-art.

2.2.1 CRDT Classes

As described in the previous Section, the objective behind CRDTs is to define a method that supports the existence of n replicas of a shared elements in a distributed system. In order to do that, it is necessary to define the update mechanism of a single replica and the mechanism that manages the update of the remainder replicas in the system. Conflict-free replicated data types can be categorized into two main classes [10, 11]: operation-based CRDTs and state-based CRDTs. Both types differ in the methods used to update the information, and where the update is executed.

- **State-based CRDTs (CvRDTs)** define a replication mechanism where and update always occurs at the source. Once the new state of the element is known, it is then propagated to the other nodes of the system in order to updated any other replicas.
- **Operation-based CRDTs (CmRDTs)** differs from CvRDTs in the way that the update and propagation mechanisms are implemented. Instead of updating the value of the shared element locally and forward the result to the remainder replicas, in this approach an operation contains the required semantics. The update operation receives the function to be applied to transform the existing replica into an updated replica. Notice that a reliable broadcast mechanism among all replicas should exist potentially with ordering guarantees depending on the type of object to be implement.

While both approaches provide guarantees on the eventual convergence of the state, the performance of both differs. CvRDTs are simpler in design as the new state contains all the information required to update or replace the old value with the new one. CmRDTs on the other hand, while they may require a complex design impose less constraints on the network used to transmit updates. This situation is especially interesting if the state to be updated is a large object by itself. For example, consider the difference in design to implement a shared set of objects. With CvRDTs the add/remove operation will be executed locally, and the new set send through the network; with a CmRDTs design, only the differential add/remove operations with the affected element of the set will be transmitted, nor the whole set itself.
Implementations

2.2.2 CRDT Implementations

The CRDTs have been used in large scale projects in the recent years. Soundcloud (http://soundcloud.com) engineers developed Roshi [13, 14] in 2014 as a CRDT system for timestamped events. Roshi provides a CRDT set type on top of Redis providing partition tolerant, high availability and eventual consistency guarantees. The idea behind the system is to provide a high-performance index for timestamped data for a use case where an updated on one particular user must reach all its followers in the Soundcloud platform.
Eventuate [12] developed and published by RedBull in 2015 is a toolkit for building applications based on the existence of event streams. Eventuate provides an abstraction to code event-driven applications with a stack based on Scala with Akka actors.
Apart from specific frameworks built around the concept of CRDTs, other technologies make use of CDRTs to provide specific capabilites. Riak [15] is a distributed NoSQL key-value data store that provides among others a specific set of operations built on top of CRDTs such as flags, registers, counters, sets, or maps.
Cassandra [16]another NoSQL database implements the internal consistency and replication mechanism using CRDT-like sets to retrieve, compare, and merge the result of a query, or to evaluate which informations needs to be replicated in other nodes.
Other public uses of a CRDT based solution for industrial applications can be found in the TomTom [17] NavCloud where CRDTs are used to merge user information updated from multiple connected devices.

3 General Overview

The mission of Proteus is to propose and develop ready-to-use scalable online machine learning algorithms. Within this context, two main components are found, the Apache Flink distributed processing framework as the infrastructure foundation for the algorithms, and the SOLMA library developed as part of Proteus where scalable machine learning algorithms are going to be developed. As a distributed system, one of the components that affects the overall performance is the ability to cache results along the processing processes to speed up the computation. In this section, we overview the design of a generic distributed key-value cache that can be used both inside and outside of the Proteus scope. We have named this cache as PEaCH (Proteus Elastic Cache).

The proposed cache aims to provide low latency responses on a distributed elastic deployment with fault-tolerance capabilities. As a generic design, the cache could be integrated within Flink to speedup computing processes, the SOLMA library itself to cache intermediate models, and in any other part of Proteus where a distributed cache is required. The next sections describe the main features of the proposed distributed key-value cache and some of the main use cases where having such a caching infrastructure may improve the overall performance of the system.

3.1 Main features

The design of PEaCH aims to provide the following features.

3.1.1 Distributed cache with support for key-value data structures

From the different types of entity storage paradigms such as document oriented, object oriented, or row oriented, we have selected the key-value approach as the one to be implemented for a caching solution. The key-value approach provides a natural fit for a caching solution as the requesting application only needs to query a key to retrieve the associated value. The key-value approach requires the specification of a key for any value to be queried and stored in the cache. For this purpose, several options are commonly used: use the row key if the value is stored in a database or apply a hashing function to the fields that would be part of the key. From the point of view of the value to be stored no restriction should exist a priori for the type, length and content of the value.

Existing implementations of key-value datastores/caches include: Redis [4], Aerospike [AEROSPIKE] or memcached [MEMCACHED]. However, none of the existing approaches support CRDT operations.

3.1.2 P2P deploy

Peer-to-Peer (P2P) architectures are composed by a set of nodes where each node is assigned the same responsibility as its peers. In the case of a caching solution, every node in the network will support client requests, and will also store user data in the form of a key-value pair. Typical P2P architectures are based either on the existence of lookup tables if the requests send to the network contain the target node identifier, or in the existence of a Distributed Hash Table (DHT) logical overlay when the request target content identifiers and not physical node identifiers. The Akka [20] framework offers the possibility of creating highly concurrent, distributed and resilient message-driven applications with Actors on Scala. Chord [18] and DynamoDB [19] are examples of DHT based solutions.

3.1.3 Support different data stores

Providing an efficient persistency mechanism to back the cached results is important for scenarios where fault-tolerance is required at cache level. For example, consider the case where several clients are updating a value that whose lifecycle is limited by the duration of the distributed job. Such cases benefit from a persistence data store to provide recovery mechanism in case of a failure in the case service. The design of PEaCH considers the underlying persistent storage as an optional element which may not be deployed if not necessary. Additionally, a common interface is defined in order to be able to employ different persistence technologies depending on the expected guarantees in terms of fault-tolerance and availability. In this context, we consider technologies such as Redis [4] as a viable option to store persistently cached data.

3.1.4 Support for invalidation mechanisms

In relation with the aforementioned scenario where several clients may be accessing the same content, and considering the possibility of caching values both at the caching service and at the client side, an efficient invalidation mechanism will be required as to assure that all client will receive the last updated value. A simplistic approach for such mechanism could be implemented using any of the well-known eviction policies (e.g., less frequently used) or by means of a publish-subscribe mechanism where clients inform the caching service of the keys they are observing and the caching service is responsible of updating client values whenever a new update is being observed in a particular key. Whether an invalidation mechanism or client side caching is appropriate depends on the specific scenario where the cache is deployed, the size of each value, and the update rate of the keys.

3.1.5 Multi-data types support

The underlying data type to be stored in the cache should not be limited as to facilitate the integration with the native application data types. The key for a given value is represented by a String data type to provide a consistent method to refer to data. The value is not limited in data types and we plan to support the following data types:

- **Primitive data types.** Including Long, Double, Boolean, Integer, and String
- **Complex data types.** Complex data types are composed by the usual data structures: list, set, queue, stack and map.
- **User defined types**. In order to provide further extensibility capabilities and improve the integration with the native application data types, the cache should provide a mechanism to define and store UDT (User-defined types).

3.1.6 Support CRDT based data structures

Given that the cache foundations are based on the use of CRDT data structures, the cache should provide a method to perform the classical CRDT operations.

- **Query.** Read the value associated with a particular key.
- **Update.** Update the value associated with a particular key considering a specific update function. Notice that for simple data types, standard update functions may be included in the cache definition. Further improvements will allow users to specify their own update mechanisms, especially in the case of user-defined data types.
- **Merge.** Whenever the value of a given key may be in a conflicting state, a merge function should be applied. In this sense following the same approach as with the update function, the cache will provide a mechanism to specify the merge function to be used for a particular data type. Custom merge functions are also planned to be supported as to support user-defined data types.

3.1.7 Java/Scala API

The existence of an easy-to-use and well documented API is required for any service to be widely used and adopted. Given that Apache Flink and the SOLMA library are coded in JVM-based languages, the cache will offer a native Scala/Java API that allows the developers to easily integrate the cache within the requiring applications.

3.2 Use cases

A caching mechanism can be used in multiple use cases. The following paragraph describes common situations where a caching mechanism such as PEaCH may be used.

3.2.1 Client cache

Before entering in the distributed system scope, one of the most common usages of a cache appears on applications that use a cache to store intermediate results independently of whether they come from an underlying datastore, or they are created as intermediate results on the application processing. This approach permits applications to speedup their computations and request latency by means of using an internal cache. Figure [1] depicts the components of this scenario.

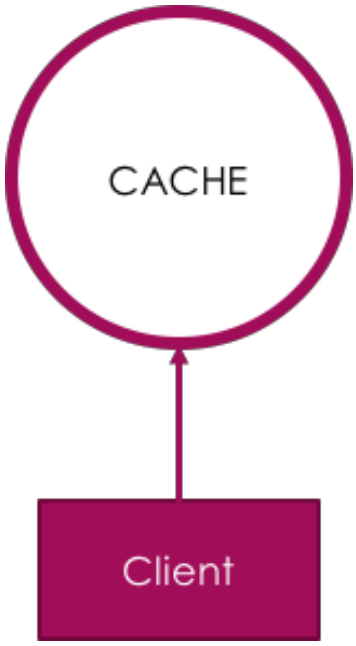

Figure 1 Client cache

3.2.2 Datastore-backend client cache

The second scenario is when an application uses a cache mechanism to speedup the access to an underlying datastore. A classic example of this scenario is a "row cache". In this scenario two approaches appear. In the first approach the cache is used by the application to cache results from the underlying datastore, but the cache does not request data automatically to the datastore as request go from the application to the datastore and the retrieved data is stored on cache for further requests. The second scenario considers a direct link functionality on the cache that allows to retrieve data from the underlying datastore in case of a cache miss. Both scenarios are depicted in Figure [2] Future work on this respect includes the definition of a miss-function defined by the users so that a cache will automatically request data from the underlying datastore.

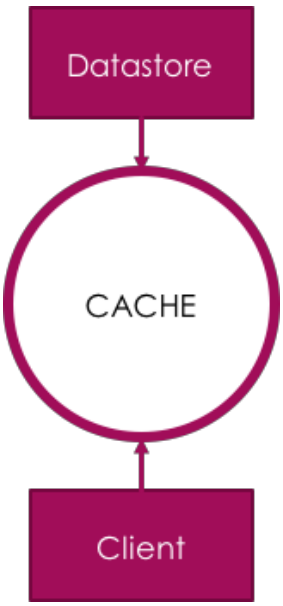

Figure 2 Datastore backend client cache

3.2.3 Publish-subscribe scenarios

The third scenario considers the existence of a distributed system where multiple clients access a specify key. The idea behind a publish-subscribe scenario is that elements with the producer role will create or update values for a given topic (in this case a particular key). The values are stored in a broker, which in this case will be the caching solution, and the elements with the consumer role should be informed whenever a value for a key they have subscribed to is changed or updated. By allowing client-side caches to subscribe to a key, the value will be updated on client-caches whenever each of them updates the key. Notice that in this scenario there are no guarantees on the consistency of the value on each client in the event of an update. Depending on the approach, clients may lose updates, or use out-of-date values in some situations. Figure [3] depicts the aforementioned scenario.

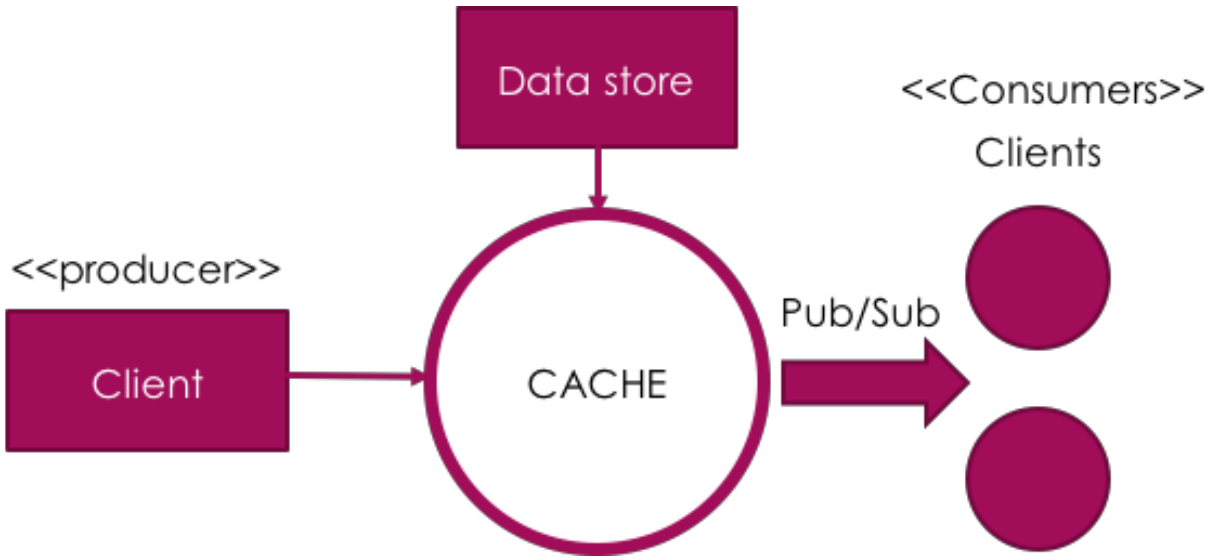

Figure 3 Publish subscribe scenario

3.2.4 CRDT functions

The final scenario assumes the existence of a distributed system where several clients access the same key and any client may read or update the value associated to it. While this scenario may seem similar to the previous one, the main difference is the use of CRDT functions that assure the eventual consistency of the values being used by any client of the cache. Notice that mixed scenarios where a publish-subscribe mechanism coexist with a CRDT cache are also considered.

4 Requirements

4.1 Architectural requirements

Architectural Requirements	
ID: PCH-1.1	Distributed architecture
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must deploy in different computers. The components interact with each other in order to achieve common goal.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.2	Horizontal scalable architecture
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must be able to improve the performance adding new computers in the deploy.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.3	Fault tolerant
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must continue operating properly in the event of the failure of some of its components.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.4	Fault tolerant
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must continue operating properly in the event of the failure of some of its components.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements

ID: PCH-1.5	Low latency
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must guarantee read and write operation with low latency (<1s)
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.6	In memory storage
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must cache the information in memory of each data nodes.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.7	Persistence
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system must persist the information in second level of persistence.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-1.8	Support of different storages
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system should be able to support different storages.
Priority	SHOULD
Status	UNCOVERED
Comments	

Architectural Requirements

ID: PCH-1.9	Client cache
Version	0.0.1 2016-10-26
Subsystem	System
Dependencies	
Description	The system should support client cache
Priority	SHOULD
Status	UNCOVERED
Comments	

4.2 API requirements

Architectural Requirements	
ID: PCH-2.1	Get method
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API must have a method to recover items from cache using the key.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-2.2	Put method
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API must have a method to put values in the cache.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-2.3	Merge method
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API must have a method to put values in the cache with a merge functions associated.
Priority	MUST
Status	UNCOVERED
Comments	

Architectural Requirements

ID: PCH-2.4	Atomic operations
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The operation GET, PUT and MERGE should be atomic.
Priority	SHOULD
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-2.5	Sequence of commands
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API should support sequence of commands. The operation should execute in order without interruption with other commands.
Priority	SHOULD
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-2.6	Set expiration date
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API should include operation to handle expiration dates. The date is associate to each key in the cache.
Priority	SHOULD
Status	UNCOVERED
Comments	

Architectural Requirements	
ID: PCH-2.6	Client cache
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API should support configuration command for create a client cache.
Priority	SHOULD
Status	UNCOVERED
Comments	

Architectural Requirements

ID: PCH-2.6	Client cache invalidation commands
Version	0.0.1 2016-10-26
Subsystem	API
Dependencies	
Description	The API should support configuration command for create a client cache.
Priority	SHOULD
Status	UNCOVERED
Comments	

API Requirements	
ID: PCH-3.1	Support CRDT
Version	0.0.1 2016-10-26
Subsystem	Other
Dependencies	
Description	The system should support conflict-free replicated data types.
Priority	SHOULD
Status	UNCOVERED
Comments	

4.3 Documentation requirements.

Documentation Requirements

ID: PCH-3.1	API Documentation
Version	0.0.1 2016-10-26
Subsystem	Documentation
Dependencies	
Description	The API must be documented.
Priority	MUST
Status	UNCOVERED
Comments	

Documentation Requirements	
ID: PCH-3.1	Installation guide
Version	0.0.1 2016-10-26
Subsystem	Documentation
Dependencies	
Description	The system must include an installation guide
Priority	MUST
Status	UNCOVERED
Comments	

Documentation Requirements	
ID: PCH-3.1	User guide

Version	0.0.1 2016-10-26
Subsystem	Documentation
Dependencies	
Description	The system must include a user guide, with use case examples.
Priority	MUST
Status	UNCOVERED
Comments	

Documentation Requirements	
ID: PCH-3.1	Integration guide
Version	0.0.1 2016-10-26
Subsystem	Documentation
Dependencies	
Description	The system should include a integration guide with other data stores.
Priority	SHOULD
Status	UNCOVERED
Comments	

Documentation Requirements

ID: PCH-3.1	Architecture overview
Version	0.0.1 2016-10-26
Subsystem	Documentation
Dependencies	
Description	The system should include architecture overview.
Priority	SHOULD
Status	UNCOVERED
Comments	

4.4 Other Requirements

Documentation Requirements	
ID: PCH-3.1	Integration with Apache Flink
Version	0.0.1 2016-10-26
Subsystem	Other
Dependencies	
Description	The system must integrate with Apache Flink.
Priority	MUST
Status	UNCOVERED
Comments	

5 Conclusions

This deliverable presents the catalogue of requirements for PEACH to be developed in the context of PROTEUS. The requirements are derived from the need of the online machine learning techniques. However, PEACH is able to resolve other problems associated with the big data projects and it can be used in a variety of use cases.

The final aim of PROTEUS is to obtain a system that is interactive and fulfils the requirements associated with scalable online machine learning, hybrid data processing and interactive real-time visual analytics. PEaCH allows to use complex machine learning models using a low latency access.

.

6 References

[1] http://www.memcached.org *Memcached web site.*

[2] http://memcachedb.org/ *MemCached DB web site.*

[3] http://cachelot.io *Cachelot web site.*

[4] http://redis.io Redis web site.

[5] http://www.celeryproject.org/ Celery web site.

[6] http://www.alluxio.org/ Alluxio web site.

[7] http://ignite.apache.org Apache ignite web site.

[8] http://www.oracle.com/technetwork/middleware/coherence/overview/index.html Oracle Coherence website.

[9] Vogels, Werner. "Eventually consistent." Communications of the ACM 52.1 (2009): 40-44.

[10] Shapiro, Marc, et al. "Conflict-free replicated data types." Symposium on Self-Stabilizing Systems. Springer Berlin Heidelberg, 2011.

[11] Shapiro, Marc, et al. A comprehensive study of convergent and commutative replicated data types. Diss. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[12] http://rbmhtechnology.github.io/eventuate/ Eventatu

[13] https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events

[14] https://github.com/soundcloud/roshi

[15] http://basho.com/products/#riak

[16] http://cassandra.apache.org/

[17] [17] Practical demystification of CRDT. Dmitry Ivanov, Nami Naserazad, Lambda Days 2016, http://www.esi.nl/innovation-support/documents/symposium-2016/2-SA-Demystification-of-CRDT_TomTom-NavCloud-CRDT.pdf

[18] [18] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.

[19] Sivasubramanian, Swaminathan. "Amazon dynamoDB: a seamlessly scalable non-relational database service." Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.

[20] http://akka.io/ Akka Website.