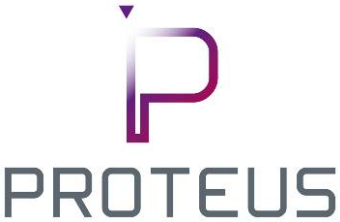

PROTEUS

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

687691

D3.5 Updateable-state management prototype implementation

Lead Author: Álvaro Agea, Daniel Higuero
With contributions from: Juan Manuel Tirado
Reviewer: Jeyhun Karimov

Deliverable nature:	D
Dissemination level: (Confidentiality)	Public
Contractual delivery date:	1/6/2017
Actual delivery date:	17/05/2017
Version:	1.0
Total number of pages:	18
Keywords:	Cache, Distributed Databases, big data

Abstract

The current document describes the prototype implementation for a distributed key-value caching mechanism named PEACH that can be leveraged by the different components of the PROTEUS project. The prototype implementation of PEACH is a distributed cache with support for key-value data which will support the integration within the Apache Flink [4] environment. The code associated to this deliverable is the PROTEUS GitHub repository [1].

Executive summary

This deliverable describes the current implementation of PEACH, a distributable cache designed and develop for distributed systems. The cache is used by Apache Flink to allow the sharing of information among distributed tasks. In particular, this is specially useful for sharing models and managing their updates in a distributed machine learning environment.

The document describes the main features of the cache that have been covered in the prototype version including the peer-to-peer architecture, the support for different storage backends for the cache and the design and definition of the client modules. The peer-to-peer implementation is supported by the AKKA framework and the storage backend used for the prototype is Redis. Both technologies have a proven record in terms of performance and are well known in the Big Data ecosystem.

Additionally, the document includes the list of requirements that have been covered in the prototype and introduces the concepts that will be implemented in future versions.

Document Information

IST Project Number	687691	Acronym	PROTEUS
Full Title	Scalable online machine learning for predictive analytics and real-time interactive visualization		
Project URL	http://www.proteus-bigdata.com/		
EU Project Officer	Martina EYDNER		

Deliverable	Number	D3.5	Title	Updateable-state management prototype implementation
Work Package	Number	WP3	Title	Scalable Architectures for both batch data and data streams

Date of Delivery	Contractual	M18	Actual	M18
Status	version 1.0		final □	
Nature	report x demonstrator □ other □			
Dissemination level	public x restricted □			

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

Abstract (for dissemination)	The current document describes the prototype implementation for a distributed key-value caching mechanism named PEACH that can be leveraged by the different components of the PROTEUS project.
Keywords	Cache, Distributed Databases, big data

Version Log			
Issue Date	Rev. No.	Author	Change
17/5/2017	1.0	Alvaro Agea	First Version.

Table of Contents

List of figures

Abbreviations

PEACH: PROTEUS **E**l**A**stic ca**CH**e

API: Application programming interface

CRDT: Conflict-free replicated data type

TTL: Time-to-live

P2P: Peer-to-peer

UDT: User defined types

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 ensure scalability algorithms should 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 [4] when executing the SOLMA library also developed as part of Proteus. While the cache offers a generic and extensible solution for caching in distributed systems, 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 also be maintained in cache to be applied to new incoming data.

The Proteus deliverable D3.4 lists the requirements of this cache and overviews the architecture of the project. The focus of this document is to describe the implementation of the first version of PEACH and review which requirements are addressed in this version of the deliverable.

1.1 Document objectives

The objective of this document is to present the PEACH design and implementation focusing on the following areas:

- Architecture overview and code structure
- Installation and minimal requirements.
- List of covered requirements
- Roadmap and future work.

1.2 Document structure

The first part of the document presents an overview of the internal architecture of PEACH and the code structure [page 9]. After that, we present a guide describing how to build and install the project [page 14]. Finally, this document will list the covered requirements [page 15] and present our conclusions [page 17].

1.3 External references

The main repository of the project is in Github [1] and the source code is licensed under the Apache License 2.0 [2].

2 Architecture overview

This section describes the architecture of PEACH and the internal code structure.

2.1 Main components and communication

The PEACH architecture is composed of two types of elements: PEACH Server and Peach Client. Every element is deployed using a peer-to-peer architecture and communications are carried out using the AKKA technology to send messages among the components of the system. Figure 1 depicts the main components of the architecture.

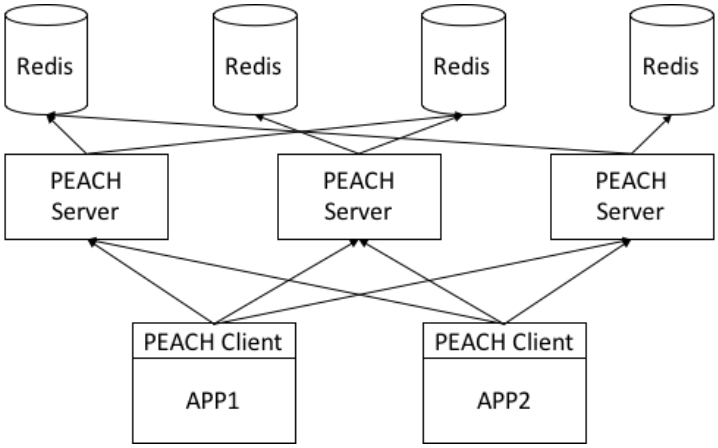

Figure 1 Main components of PEACH

- **PEACH server.** It is the main piece in the PEACH architecture, receives all client requests, coordinates the execution of them, and sends back the results to the clients. The architecture follows a P2P approach, so all the PEACH server instances are configured identically and are able to perform the same functions.
- **PEACH client.** It is a library that offers an API for clients to interact with the underlying cache. It manages the internal communication with the required PEACH servers. In terms of configuration, each client must know at least one server to connect to with the PEACH server. Other points of contact are automatically managed within the P2P framework. No support for secure communication is available in the current version.
- **Redis.** It is the distributed storage used by PEACH to persist user data. Redis was choosen because it is a scalable in-memory data store with support to key/value models.
- **AKKA.** It is used to simplify the communication between the different components of the architecture. It is also the backbone required to create a distributed and scalable environment for PEACH. For the implementation of the prototype, a specific fork of Akka provided by Flink is used to assure compatibility with the Apache Flink requirements.

2.2 PEACH vs Redis

The design of PEACH requires an external data store to persist information. The selected data store for the current version is Redis whose features fit with the requirements of the project (e. g., low latency read/write, atomic operations, fault tolerance). However, PEACH offers a layered design that permits to use other platforms as long as they satisfy the needs of PEACH server and client components. Other data store solutions such as Cassandra or Memcache could be easily employed maintaining the design presented in this document.

In this sense, PEACH is layer on top of a data store (Redis in this case) that allows clients to support new data types, publish/subscribe methods, a client cache or support to distributed ML models.

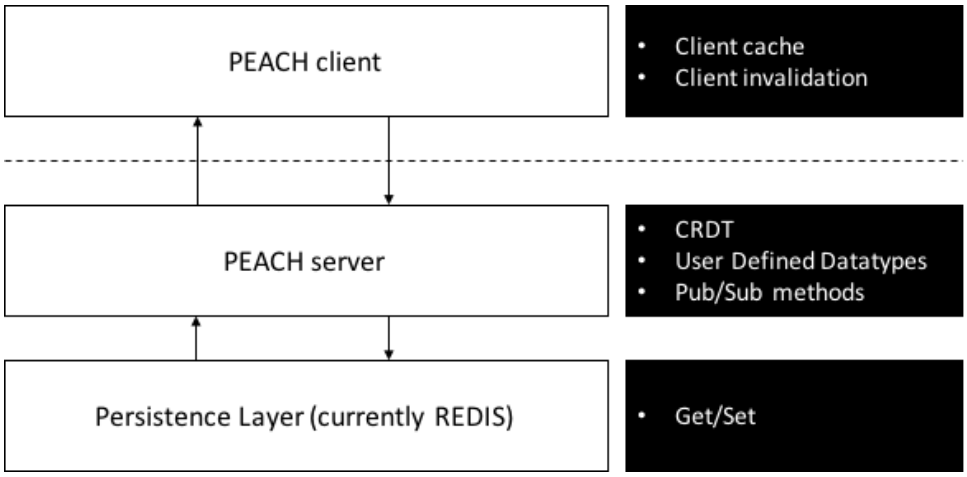

Figure 2 Functional architecture of PEACH

2.3 Code structure

Figure 3 describes the component structure of the different modules inside PEACH.

- **Common.** This module contains the communication messages and some utilities that are used through the remainder of the modules. Client and Server depend on this module. This module is a library.
- **Client.** It is the module that contains the API for the connection with the PEACH server instances. This module is a library and does not have a deployment script.
- **Server.** The Server module contains the logic required to connect with the distributed storage to persist user data. It also contains the communication logic among server instances. This module is a library.
- **RedisServer.** This module is an application that implements the PEACH server abstraction using a Redis storage backend. This module is an application and requires configuration files.
- **RedisServerDist.** This module creates the deployment files for the Redis Server module.

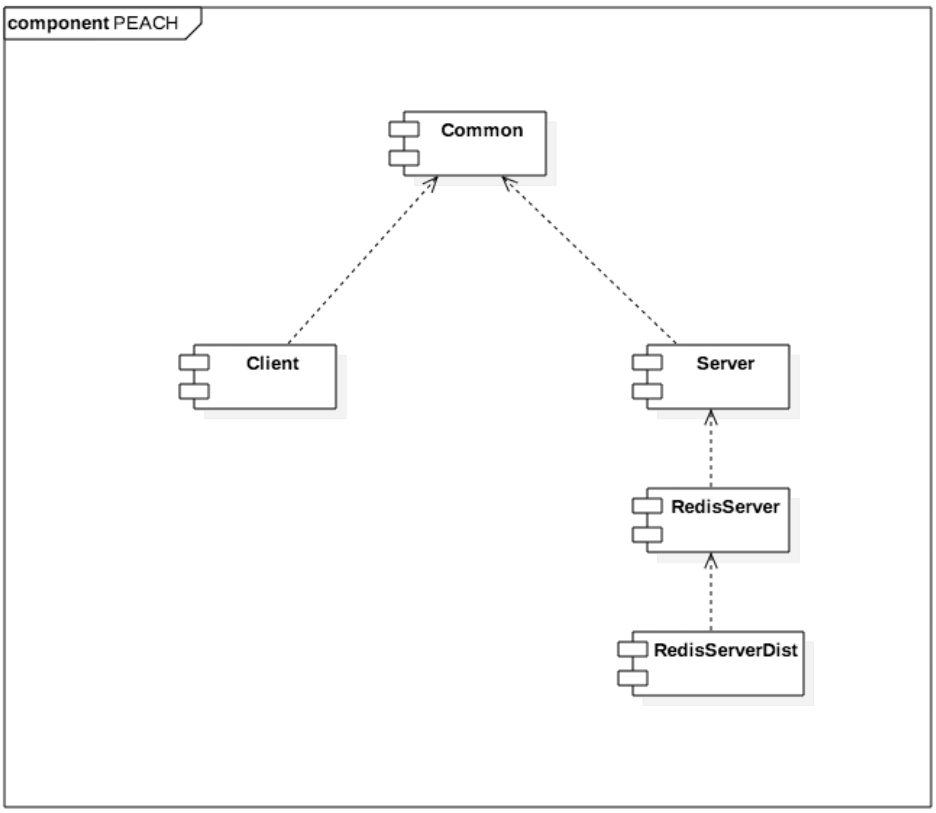

Figure 3 PEACH library structure.

2.4 PEACH server structure

Figure 4 depicts the class model diagram for the Server module.

- **PeachServer.** This class is responsible for initializing the PEACH server and the AKKA actor system.
- **AkkaPeachServerReceptor.** It handles the communication with the clients, receives messages and connects with the data storage.
- **PeachServerConfig.** The interface is the basic configuration for the PEACH server. It contains all the necessary data to configure a PEACH server.
- **ExternalServerCache.** It is the interface used to abstract the data stores for the server. This class contains all the methods required to persist and recover data from the data store.
- **MockupExternalServerCache.** It is a mockup that simulates an in-memory data store. The mockup has been created for testing purposes.
- **Service.** It is an interface that defines new services in an application.
- **AbstractService.** It is a helper to create new services.

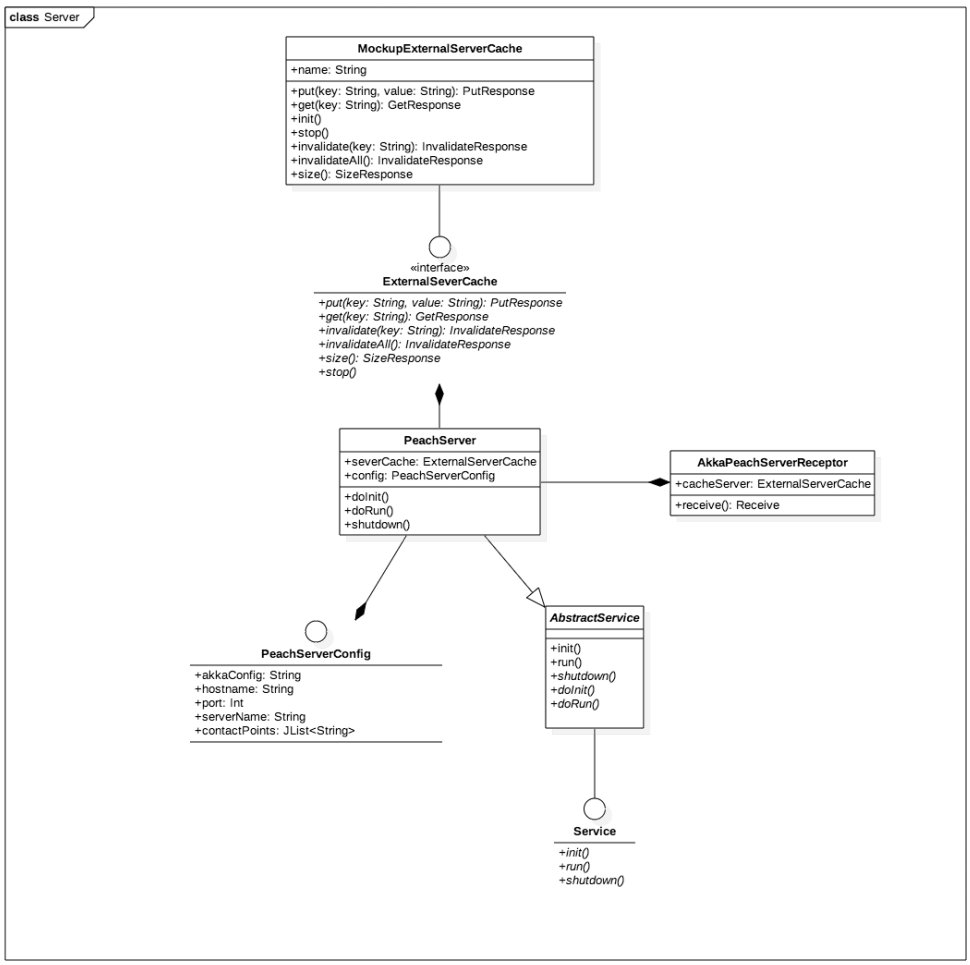

Figure 4 Server module structure.

2.5 PEACH client library

Figure 5 shows the class diagram for the client library.

- **PeachClient.** It is an interface with the available methods in the distributed cache, this is the main access for an external application to connect with the PEACH servers.
- **PeachAkkaClient.** This implementation of the PeachClient communicates with the PEACH server instances using the AKKA protocol.
- **MockupPeachClient.** This mockup is an abstraction of the PeachClient. This mockup is used for unit and integration testing purposes.
- **PeachClientConfig.** It is interface with the basic parameters required to configure the library.

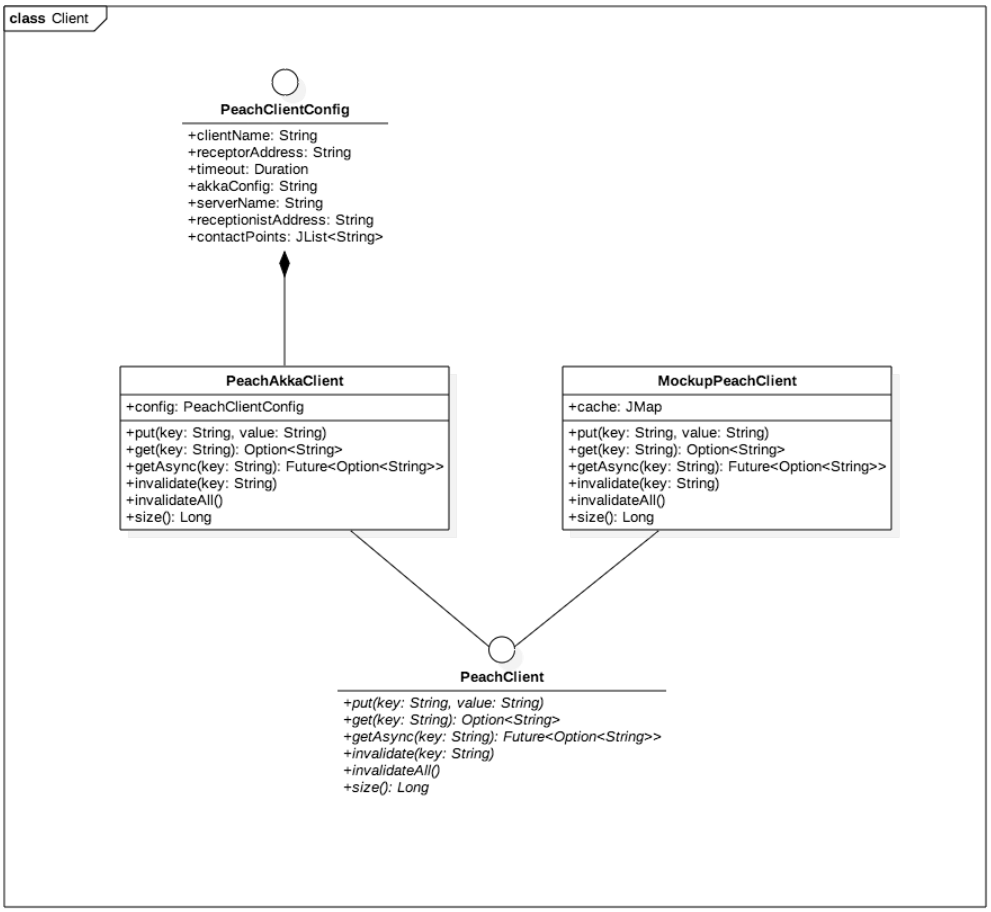

Figure 5 Client module structure.

3 Installation and deployment guide

This section describes the building procedure of PEACH and how to deploy the system in a real environment. Section 3.1 demonstrates how to build the code from the repository and Section 3.2 how to deploy a PEACH servers.

3.1 Building from source code

This guide describes how to build the PEACH project so that it can be used and deployed. The following steps are required to generate both the client library and the server deployment scripts. Before going into the steps, the following pre-requirements are needed:

- Maven 3.3.X[3] is needed to build the project.
- Deploy a Redis cluster that the servers can connect to. For testing purposes, a local instance is enough.

Steps required to build the project:

1. Clone the peach repository from the GitHub [1] repository.
2. Execute the command `mvn clean install` to build the project and produce the library and deployment scripts. This process will compile the code and run all the tests.
3. The code is compiled and the libraries installed into the local maven repository. To deploy the servers, refer to Section 3.2

3.2 Basic deploy

The results of the compilation process are the client library installed into the local repository and the PEACH server distributable package. The distributable package is located into the module peach-redis-server-dist inside the “target” directory and it has the next structure:

- **Bin folder.** Contains all the executable scripts.
  - **peach-redis-server-app.** Script to launch the server like an application.
  - **peach-redis-server-daemon.** Script to launch the server as a Linux daemon.
- **Conf folder.** Contains all the configuration files.
- **LIB folder.** Contains all the required JARs to launch the server.

4 List of requirements

This section overviews the requirements covered by the prototype version of PEACH and defines a roadmap with the next steps of this project. The current version of PEACH is v0.1.0, the next version, v0.2.0, will be available in M21 and the last version, v0.3.0 will be available at the end of the project.

4.1 Covered requirements

This section lists the covered requirements and explains how this version have met them.

4.1.1 PCH-1.1 Distributed architecture

The PEACH project uses the AKKA framework to distribute the process across different machines. The project uses two internal AKKA's patterns to share the information between the nodes. The PEACH server uses the Cluster Specification [5] to create a cluster abstraction for a group of nodes with the same cluster name. The connection with the PEACH server uses the Cluster Client pattern [6]. This model allows the communication between two different cluster systems using the AKKA remoting protocol.

These two methods are included into the core of the PEACH project and guarantee a distributed architecture.

4.1.2 PCH-1.2 Horizontal scalable architecture

Horizontal scaling is defined as "the ability to scale connecting multiple hardware or software entities so that they work as a single logical unit" [7]. The PEACH project has two scalability points, the PEACH server, and the data store module.

The distributed architecture of the PEACH [PCH-1.1] server makes possible to increase the number of operations, adding nodes to the cluster.

The capability of the data store to support horizontal scaling depends on the selected technology. The prototype version supports REDIS [8] as storage, and this technology raises methods to scale out the system [9].

4.1.3 PCH-1.3 P2P

Peer-to-Peer (P2P) architectures are composed of a set of nodes where each node is assigned the same responsibility as its peers. In the case of PEACH, every node in the network will support client requests, and will also communicate with the storage server to store the user data in the form of a key-value pair.

Because PEACH server uses the Akka Cluster [5] specification as part of the core [PCH-1.1], it provides a fault-tolerant decentralized peer-to-peer based cluster with no single point of failure.

4.1.4 PCH-1.4 Fault tolerant

PEACH uses the AKKA framework to guarantee the failure tolerance. The AKKA Cluster Specification [5] uses an automatic failure detector. It is responsible to guarantee that all the nodes are reachable by the rest. In the case that some node is down, the rest of the nodes receive information about the new situation. If a client is connected to this unreachable node, it changes the connection with another server in the cluster. The data store selected by the prototype version, REDIS [8], supports fault tolerance methods [10] as replication.

4.1.5 PCH-1.8 Support of different storages

The PEACH server has an abstraction of the data store model that allows changing the underlying data store adding a class that implements the ExternalServerCache trait and creating a final class using the AbstractService helper.

4.2 Architectural requirements overview

ID	Name	Priority	Status	Version
PCH-1.1	Distributed architecture	Must	COVERED	0.1.0
PCH-1.2	Horizontal scalable architecture	Must	COVERED	0.1.0

PCH-1.3	P2P	Must	COVERED	0.1.0
PCH-1.4	Fault tolerant	Must	COVERED	0.1.0
PCH-1.5	Low latency	Must	UNCOVERED	0.3.0
PCH-1.6	In memory storage	Must	UNCOVERED	0.3.0
PCH-1.7	Persistence	Must	UNCOVERED	0.3.0
PCH-1.8	Support of different storages	Should	COVERED	0.1.0
PCH-1.9	Client cache	Should	UNCOVERED	0.3.0
PCH-4.1	Integration with Apache Flink	Must	UNCOVERED	0.2.0

4.3 API requirements

ID	Name	Priority	Status	Version
PCH-2.1	Get method	Must	COVERED	0.1.0
PCH-2.2	Put method	Must	COVERED	0.1.0
PCH-2.3	Merge method	Must	UNCOVERED	0.2.0
PCH-2.4	Atomic operations	Should	UNCOVERED	0.2.0
PCH-2.5	Sequence of commands	Should	UNCOVERED	0.2.0
PCH-2.6	Set expiration date	Should	UNCOVERED	0.2.0
PCH-2.7	Client cache	Should	UNCOVERED	0.3.0
PCH-2.8	Client cache invalidation commands	Should	UNCOVERED	0.3.0
PCH-2.9	Support CRDT	Should	UNCOVERED	0.2.0

4.4 Documentation requirements.

ID	Name	Priority	Status	Version
PCH-3.1	API Documentation	Must	UNCOVERED	0.3.0
PCH-3.2	Installation guide	Must	UNCOVERED	0.3.0
PCH-3.3	User guide	Must	UNCOVERED	0.3.0
PCH-3.4	Integration guide	Should	UNCOVERED	0.3.0
PCH-3.5	Architecture overview	Should	UNCOVERED	0.3.0

5 Conclusions

This deliverable presents the first prototype of PEACH and the current status of the project. Currently, the most important requirements are covered by this first version. The next version will address the support of CRDT structures and advanced client cache features.

5.1 Future works

This section describes the features to be included into the next versions of PEACH.

5.1.1 CRDT structures

The support of CRDT data structures is a challenge for the PEACH module, the cache should provide a method to perform basic 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.

This functionality will solve the limitation of the cache to update part of the value in a specific key.

5.1.2 Client cache

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 data store, or they are created as intermediate results on the application processing. This approach permits applications to speed up their computations and request latency by means of using an internal cache. PEACH will provide a client cache to execute operations without sending all these operations to the PEACH server.

5.1.3 Support for distributed ML models

To provide further capabilities and improve the integration with native application data types, PEACH will provide a mechanism to define and store User Defined Types (UDT). This mechanism will make possible the creation of more complex structures and will add support for distributed machine learning models. PEACH will add two datatypes to support large matrices in the distributed cache.

- **RowMatrix** This datatype provides an abstraction of a cached matrix where each row of the matrix is stored as a key/value element into the persistence system.
- **CellMatrix.** This datatype provides the same abstraction that RowMatrix, although each cell is stored into the cache as an independent register in the key/value data store.

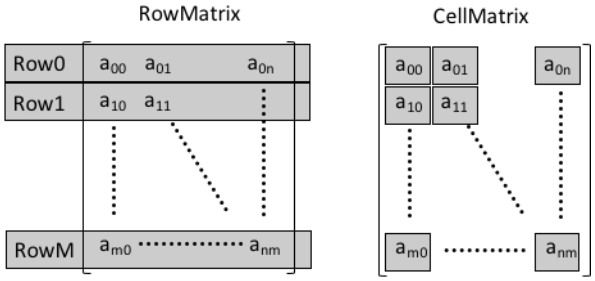

Figure 6 RowMatrix and CellMatrix

6 References

[1] https://github.com/proteus-h2020/peach *Proteus PEACH GitHub repository*

[2] https://www.apache.org/licenses/LICENSE-2.0.html *Apache 2.0 License*

[3] https://maven.apache.org/ *Apache Maven Web Page*

[4] https://flink.apache.org/ *Apache Flink Web Page*

[5] *http://doc.akka.io/docs/akka/2.3.16/common/cluster.html Akka Cluster Specification*

[6] *http://doc.akka.io/docs/akka/2.3.16/contrib/cluster-client.html Akka Cluster Client*

[7] *https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/ How to explain vertical and horizontal scaling in the cloud*

[8] *https://redis.io/ REDIS web site*

[9] *https://redis.io/topics/partitioning Redis partitioning*

[10] *https://redis.io/topics/replication Redis replication*