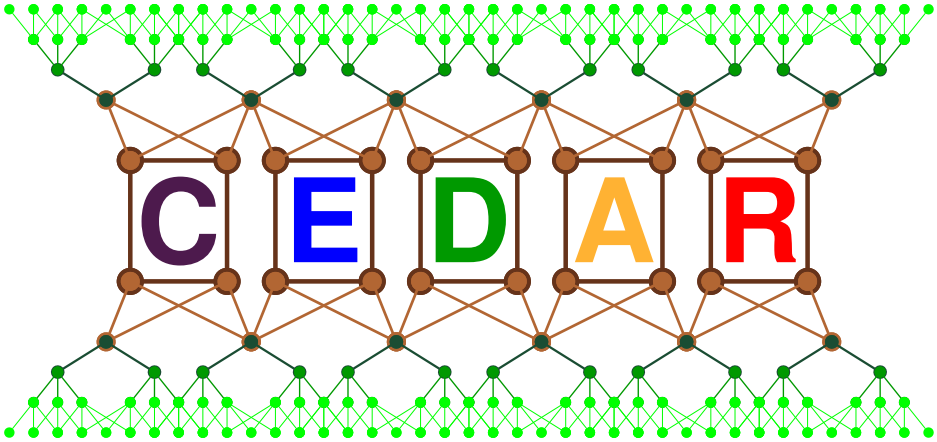


Technical Report Number 8

A Cache Only Memory Architecture for Big Data Applications

Tanguy Raynaud and Rafiqul Haque

July 2014

Publication Note

This report is based on the work done by Tanguy Raynaud during his internship in the $\mathcal{CEDAR}$ Project toward the obtention of his MSc degree at the Université Claude Bernard Lyon 1, on a topic proposed by, and under the supervision of, Dr. Rafiqul Haque [11].

Corresponding Author:

Akm Rafiqul Haque

LIRIS - UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08
Email: akm-rafiqul.haque@univ-lyon1.fr
tanguy.raynaud-gallonet@etu.univ-lyon1.fr

$\mathcal{CEDAR}$ Project's Web Site: cedar.liris.cnrs.fr

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 8**

A Cache Only Memory Architecture for Big Data Applications

Tanguy Raynaud and Rafiqul Haque

tanguy.raynaud-gallonet@etu.univ-lyon1.fr, akm-rafiqul.haque@univ-lyon1.fr

July 2014

Abstract

Distributed architecture is widely used for storing and processing Big Data. Operations on Big Data need first,locating the required data blocks and then, read them. Reading data from secondary storage to process Big Data jobs is not an ideal approach especially for high performance applications. Because, the processors cannot access data faster if they are stored in secondary devices. In addition, fetching data from main memory is time consuming due to limited I/O bandwidth. Therefore, to optimize the application performance, it is not sufficient to have efficient algorithms only, an efficient architecture is needed to provide faster data access to the processors. The need for such an architecture has been a research issue for a long-time, however, the state-of-the-art is still missing one. This paper develops a promising architecture which caches data in main memory. It essentially transforms a main memory into a *attraction memory* which enables high-speed data access. Also, it enables automatic migration of data blocks and computations across the nodes contained in the clusters. It offers an exchange protocol for fast transfer of data blocks between the different physical nodes and speeds up job processing. The proposed architecture combines the power of Cache-Only Memory Architectures (COMAs) and the structural principle of Hadoop.

Keywords: Cache Only Memory Architecture, Big Data, Attraction Memory

Table of Contents

1 Introduction

Data is becoming bigger everyday. Today, its size ranges from Gigabytes (GBs) to Petabytes (PBs). It has been predicted by many such as Zikopoulos *et al.* [14] that the data size will reach to *Yottabytes* in future. The rapid increase of data has given the rise to several problems related to computation (*e.g.* processing a job), predominantly, *efficient access to mammoth size datasets*. *Time* more specifically *time to access data* is the critical attribute because it determines the efficiency in terms of processing jobs on Big Data. In Big Data researches specifically in literature, importance has given to the underlying algorithms such as *query processing algorithms*. They have been heavily investigated and improved for querying Big Data efficiently. The system level issues such as high-speed data access and fetching them to CPU cache in case of *cache miss* are important as well, yet overlooked in existing technologies. A cache miss refers to an unsuccessful attempt by a CPU to read or write a data block in its cache.

In a distributed environment, efficient management of memory that stores data as blocks is of critical importance. In this environment, different variants of times are critical and as such determine the performances of applications. For instance, *access time* to a desired data block is a critical attribute. It is composed of *time to locate data blocks* and *time to load data to main memory from the hard disk*. These two in addition to others such as *query execution time* determine the efficiency of Big Data applications. The delay to locate data blocks or load data can degrade the performance significantly.

Furthermore, accessing data stored on secondary devices is time-consuming. In fact, even the disks with the fastest RPM (rotation per minutes) consume a significant amount of time to read or write data to the secondary devices. Therefore, it is highly unlikely that a high-performance application would be able to perform jobs efficiently using disk-based system architecture. The conventional *in-memory* based architecture could be an option to accelerate processing time. However, it is not a suitable option for the applications which process real-time queries. The key reason is the limited *I/O bandwidth*. In addition, the size of main memory available in the market is an obvious limitation. Although, the technologies such as Virtual Shared Memory [13] are available to deal with the size problem, the communication overhead and complexity of cache coherence can never be ignored. Taking all these facets into account, we conclude that *an efficient system architecture for supporting the Big Data applications is missing.*

Our main interest in this research paper lies at the system layer. Our objective is to *develop a high-performance architecture called 'CedCoM' (CEDAR Cache Only Memory), which will enable efficient data processing by making data access faster for the processors and therefore, by increasing the cache hit ratio*. The proposed architecture combines the power of Cache-Only Memory Architectures (COMAs) and the structural principle of Hadoop. It aims to solves two major problems. First, avoiding a systematic access to secondary storage when the processors have to execute a job and finding an effective and efficient way to provide access to all necessary data blocks to a machine. Second, enabling migration of data blocks from one node to another dynamically when required. It is worth noting that, the ultimate goal of the architecture

is to support high-performance query processing on Big Data.

The remainder of this technical report is organized as follows. In Section 2 we describe the motivation of this research. Section 3 presents the related works that have been done so far. A high-level overview of the CedCom architecture is presented in Section 4. Section 5 describes the development of the architecture. The experiment and results are presented in Section 6. The final section provides a conclusion and the outlook of this research.

2 Motivation

There are many efficient query processing algorithms. In distributed databases, efficient algorithms play an important role to identify and efficiently query a specific block of data. However, reducing time to access physical data is not within the scope of these algorithms. In addition, these algorithms are not concerned of *time to fetch data* from secondary storage or memory to CPU cache. However, as said in Section 1, these temporal attributes heavily influence the overall performance of Big Data applications. Since applications are mostly I/O bound and processors are more powerful than memory, guaranteeing high-speed access to data is of paramount importance. If *access time* could be reduced, the application could be able to process jobs *e.g.* queries on Big Data with high-speed. This would essentially solve the problem related to processing realtime queries efficiently on Big Data.

3 Related Work

This section discusses the works related to system architecture and memory management which are mainly used in the area of distributed, parallel, and high performance computing.

3.1 Shared memory abstraction

There are different ways to share data between multiple computers. They are described as follows:

- *Virtual shared Memory (VSM)*: This term is commonly used to describe the systems which provide a shared address space by using hardware assistance [10]. The virtual memory is implemented on top of this shared address space.
- *Shared Virtual Memory (SVM)*: Unlike VSM, SVM describes a system which provides a shared memory with a software implementation on top of the operating systems (OSs) [10]. This architecture employs a *MMU (Management Memory Unit)* to provide coherent shared address spaces. Unfortunately, this architecture is not OS transparent because it uses specific operating system functions to share memory with other processors.

- *Distributed Shared Memory (DSM)*: It is another memory architecture that allows accessing shared data, without replication. Since replication is beyond the scope of this architecture, it is not concerned of data coherence. Consequently, the memory address spaces have to be explicitly managed by the user. This is one of the reasons this architecture has not been adopted in many systems.
- *Cache Only Memory Architecture (COMA)*: In COMA the memory organization is similar to Non-Uniform Memory Access (discussed in the next subsection). However, instead of storing data in a fixed location, COMA uses the storage spaces of different processors as a large cache [4] called *attraction memory*. It enables accessing these blocks by processors faster than other memory architecture such as SVM. Additionally, compared to NUMA (which enables storing a block of data using a unique address and copying it in all the processors' caches), COMA stores the blocks only once and migrates them dynamically whenever a processor require them. COMA reduces data copies to many processors. However, in COMA, a block can be duplicated in some specific cases for instance, data required by two or more nodes at the same time. COMA caches data in main memory, which can significantly promote the performance of data retrieval because the access time required to load data from secondary storage is eliminated in this architecture [9].

3.2 Memory access

Uniform Memory Access (UMA) and *Non-Uniform Memory Access (NUMA)* are the two widely known techniques for accessing data in the memory. They are described in the following:

- *Uniform Memory Access*: In UMA, accessing data depends on a single bus and all the processors share the physical memory uniformly [4]. This essentially means that the data is stored in a location (often a centralized server) that is accessible by all processors uniformly. This architecture is effective when many clients need to share data, yet it is insufficient for Big Data applications. The key reasons are two-fold: the data size often exceeds the capacity of a single server and the nodes need a permanent access to data, which may cause network congestion.
- *Non-Uniform Memory Access*: In NUMA, the memory is divided between the different processors, and each block has a fixed location. The processors will access their local memories, which is faster than a remote access [4]. This architecture does not help in data migration; more specifically, the data blocks do not move even if needed. The only way making data available to the processors is to have a copy in the local cache of the processors. This architecture needs an efficient cache coherence protocol to guarantee consistency of the data blocks.

3.3 Cache coherence

A distributed system comprises one or more clusters. Typically, each cluster consists of multiple physical nodes that can contain several processors. While running, if a processor needs a block of data, it is copied to its cache. Consequently, a large number of copies of data blocks are diffused within and across the physical nodes of a cluster. These copies must be consistent to avoid any *dirty read* (incorrect data). If no *write* operation is performed, the blocks remain consistent without requiring any management tasks. However, in case a data block should be changed, the copies of the corresponding block needs to be updated to keep them consistent to the latest version of the data block.

Different methods to maintain data consistency in a distributed architecture already exist. This section briefly describes some notable protocols used for preserving data consistency.

The *Write-invalidate* protocol relies on *write-once read-many* principle. It allows only one *write* operation at a time, however, it allows multiple *read* operations. When a node updates a block, it sends an *invalidate signal* to all the nodes that have the copy of this block and write new data in the block. The other nodes then flag their copies as obsolete and remove them. If the copy is needed for computation, the requesting nodes request a copy of the new block directly to the one that performed the update [10]. The main advantage of this protocol is that the data is not broadcasted when an update occurs. Additionally, it allows recovering data progressively.

In *Write-update* protocol, the node which performs an update sends the data to all nodes which have the copy of that block [10]. This instantly guarantees usability of the data blocks, because all blocks residing in the nodes are essentially up-to-date. This protocol is hard to apply in large networks because it generates heavy traffic, which can quickly saturate the connection.

Between these two protocols *Write-invalidate* is more suitable for a large architecture where data is cached by a wide number of processors simultaneously. Whereas, the *Write-update* is preferable to those where data are not frequently cached, yet faster access is required. The systems which rely on central memories (as opposed to attraction memories) use one of these two protocols. Essentially, they can be integrated to complement the previous protocols to determine the updates in the central memory.

The *write-through* approach updates the central memory each time an update occurs. In *write through* systems, the main memory is always up to date [10]. The other approach is called *write-back* which does not synchronise the main memory of a node *at once* [10]. Rather, the other processors contact the node having the latest copy of the block. Additionally, this protocol updates main memory if the data block is removed from the cache. This protocol avoids unnecessary accesses to central memory, which is particularly useful for the architectures where data is often changed.

There are some advanced protocols that optimize updates during write operations. *Write-once* is one of them. This protocol introduces several states (namely *not modified*, *possibly shared*, and *reserved*) of operations performed on memory pages. It reduces the overall bus traffic by performing the *write-update* operation during the first write, and then it carries out *write-invalidate* operation [3] [10].

However, the protocols presented above are not suitable for multi-bus architectures. Thus, a special type of cache coherence protocol called *directory based cache coherence* [1] was introduced for these architectures. This protocol employs specifics directories for maintaining coherence between caches. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

In the *full map directory* architecture, the directories contain a list of processors present. A single bit is used for each of them to know whether or not a processor contains different blocks [1]. In this architecture, when a *cache miss* (it refers to unavailability of data in cache) occurs, the processor can contact any directory to locate the data block. Additionally, for each update, all the directories must be updated. For this architecture, the size of directories are very important, because each of them stores the information about all the processors.

Another approach called *limited directory* is nearly the same as full-map directories, except for the fact that, in this architecture, the directories are not storing all the system entries. Rather, only a limited number of parts are stored. This solution reduces the storage space that is required by the directories, while it limits the number of blocks which can be cached simultaneously.

Finally, the *chained directory* distributes the directory between the different caches. This approach addresses the size problem of directories without restricting the number of shared block copies. Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers.

3.4 The Data Diffusion Machine (DDM)

The *Data diffusion machine* is a multiprocessor memory architecture based on the COMA principle. The data is stored in different *attraction memories*, each of them is associated with a processor. When a block of data is needed by a processor, it is migrated from the source attraction memory location to the target location [10].

In DDM architecture, nodes are not explicitly interconnected, however, communicate with each other using a hierarchical system of directories. Figure 1 shows the architecture of DDM.

The directories store information about the data blocks that are stored in the leaf nodes [4]. The data is stored in set-associative memories. This approach enables the directories to locate data blocks efficiently.

Various directories of this architecture contain input/output buffers, which enable storing intermediate results while transferring data from source to target locations.

DDM provides a shared memory abstraction that encapsulates the underlying mechanisms and provides users a subset of the total distributed data in timely manner.

The main challenge of classical *hierarchical DDM* architecture is high-traffic, which may go out of control when too many data blocks are migrated at the same time. It can cause read and write buffers overflow, and can lead to a congestion and slow the block transfer significantly.

Several solutions have been proposed to resolve this problems. One of them is to use routers for an optimized transfer of data block between different directories [8].

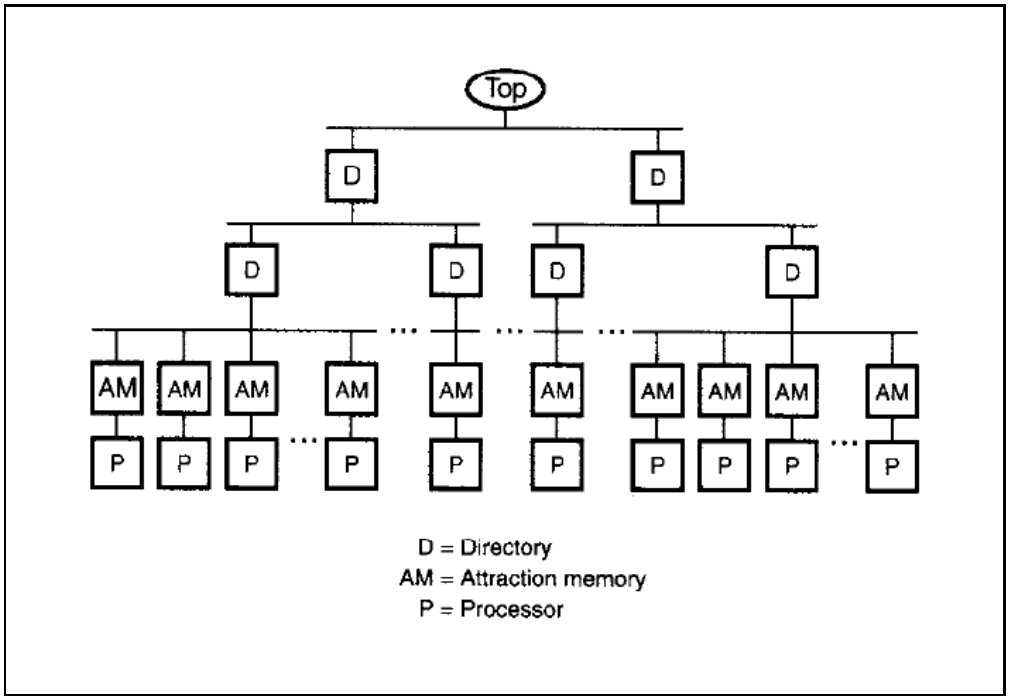


Figure 1: A hierarchical 3-level DDM

Another approach is called **Horn DDM** and essentially connects the physical nodes using a bus based point-to-point communication system [6]. This reduces congestion because the data can use different paths to be transferred between the nodes, however this approach increases the number of intermediate nodes.

3.5 The Hadoop Distributed Filesystem (HDFS)

The *Hadoop Distributed filesystem (HDFS)* is a distributed file system based on a principle of splitting a large-scale data set into many blocks and then distributing them among hundreds of thousands of nodes. Each node contains a number of blocks that represent parts of the data [12]. The file system is distributed, scalable and portable. It is able to store a huge amount of large files ranging from Gigabytes or Petabytes to a distributed style machines [13].

Hadoop comprises two layers. At the lowest layer, the *data/slave nodes* reside to store data files whereas the top layer contains a node, called *NameNode*. This node contains mainly the Metadata.

The main role of the *Namenode* is to manage data within a cluster. It stores the complete list of files and the list of data blocks. In addition, it creates a hash-map index to store information about the data blocks contained in specific nodes.

In HDFS, the *data nodes* (also called *slave nodes* or *compute-nodes*) store actual data and perform computational tasks. In order to inform the Namenode about their status, the data nodes regularly (by default, every 3 seconds) send a report containing

meta-information about their blocks and their memory space [12]. The data-nodes can communicate between them to balance data load, move a copy of a data block, or replicate it.

In order to increase the reliability, HDFS rely on **fault tolerant** technique. This technique avoids the risk of data loss which can be a consequence of crashing a node in clusters. In addition, to guarantee fault tolerance, in HDFS, the data blocks are replicated on different data nodes. If a data node crashes, the data contained within it would remain available in the cluster. In the case of namenode crashing, it starts a new replication from another node which contains a copy of the lock blocks. Furthermore, HDFS includes a *secondary namenode* as a *backup node* [13]. Its role is to create a snapshot of the namemode with an interval, which essentially supplies the information of the old namenode to the new one.

In HDFS, there is a component called *HDFS client* that enables users to load their data to the file system. The client contacts the namenode to obtain information about a storage location. Then the client opens connections with data nodes and sends data blocks. To ensure the fault tolerance, the blocks are replicated to data nodes.

4 The Design of CedCoM Architecture

This section presents our CedCoM architecture which combines the feature of COMA and Hadoop. It is worth mentioning that, it adopts the structure of Flat COMA (COMA-F). We adopt COMA-F because unlike the conventional hierarchical COMA (*e.g.*, the HORN DDM [4]), in this the data nodes are interconnected and can potentially communicate with each other directly using point-to-point network [6]. Additionally, like Hadoop, our architecture is capable of distributing data across the nodes which comprise clusters. However, unlike Hadoop, data is stored in attraction memories. These memories are physically located in main memory and logically implemented as cache memories. In CedCoM, there is no notion of main memory because it is transformed manly into attraction memories. Figure 2 shows the CedCom architecture.

The architecture comprises *directory-nodes* and *compute-nodes* (Note that, we use the terms compute-node and data node interchangeably). A compute-node consists of *processors*, *conventional cache memory*, and *main memory*. Figure 2 shows that the larger portion of the main memory of the compute-nodes is transformed into *attraction memory*. The size of attraction memory is predefined by users. In addition, the remaining portion is called *transit area*. Furthermore, the CedCom architecture allocates a certain amount of memory for storing a *directory* (*Dir*, shown in the above figure). It is worth noting that the CedCom architecture provides flexible space management for transit areas and directories.

The data blocks that are needed to execute a job by a compute-node are not necessarily stored on the same node. Since the CedCoM architecture relies on COMA, data blocks do not need a particular *home node*. Any compute-node can contain any data block. If a block is required by node but stored in another node, the target node contacts source node and then the block is migrated from the source to target node. However, in case of large size data blocks, our architecture enables transferring computations to the

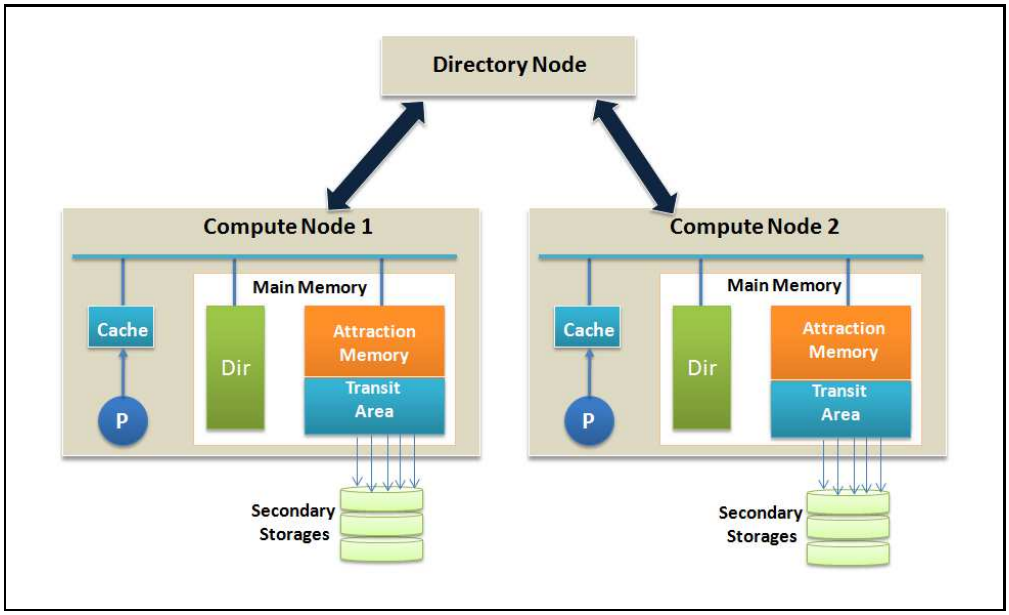


Figure 2: The Architecture of CedCom

target node, in lieu of the required blocks. This is a substantial feature of the proposed architecture. The key idea is to reduce the *delay* which can be the result of high traffic that would be created due to an exhaustive number of interactions between the nodes. Transferring computation to the data hosts would avoid communication between the nodes and eventually would ensure faster job processing.

The architecture contains a *local directory* which indexes the data blocks stored in the local attraction memory. The attraction memory contains the data blocks. The transit area contains the *least recently used (LRU)* data blocks that are moved out from the attraction memory. Also, it temporarily stores the data blocks that are about to be migrated or copied to other compute-nodes.

In addition to main memory, the CedCom architecture enables storing data blocks onto *secondary storage* (*e.g.*, Hard Disk) on special conditions. For instance, there is no space in the attraction memory of any of the nodes which belong to a cluster. The CedCom architecture allocates secondary storage automatically for the data blocks which cannot be stored in attraction memory or transit area. However, the architecture would load the data blocks automatically in attraction memories or transit areas as soon as it finds required spaces for the blocks.

The *directory-node* is essentially a metadata server which provide information such as locations and state of the data blocks. Like Hadoop, the compute-nodes are tightly linked with the directory-nodes whereas the connection between compute-nodes are kept open for a temporary session. The closing of the session should terminate all connections between nodes.

5 Development of CedCoM Architecture

This section describes the implementation of the CedCoM architecture. We used C++ programming language for the implementation. It is a suitable language for developing high-performance computing (HPC) applications with a real control of the memory used for computation. In addition, several external libraries are available to avoid reinventing the wheel. The solution implemented in this paper is independent of any specific operating system platform. The following subsections describe the development of the CedCom architecture.

The implementation is described in three parts. In the first part, we describe the development of the basic concepts. In the subsequent part, we describe how the core components: *the compute-node* and *the directory-node* have been implemented. The final part presents the details of implementation of some advanced operations.

5.1 Basic concepts

The following subsections provide details on how we developed the basic concepts of the CedCom architecture.

5.1.1 Data structure

We started the development of the CedCom architecture by defining the structure of data blocks. Since the data blocks will be frequently migrated within clusters, it is important to define the size of data blocks. To create a generic and portable structure, the data is divided according to our own file system. The size of data blocks are predefined. Each block has a unique identifier which is a reference key for the blocks and used to store the blocks in processing nodes. The unique key of a data block must not be altered. It must be the same for all processing nodes storing that data block. This identifier is assigned by the directory-node whenever a node registers a new incoming data. A node can register as many blocks as it can store and thus, can obtain a range of identifiers.

Data is stored in the 'data' part of the nodes. It is the smallest unit of a large file. The size of the data stored in one slot may vary, but cannot exceed the maximum size that was predefined. To identify the source of the data, a field name called 'filename' has been added. It enables a node to find different parts of a file by using their filenames. A specific directory located in the directory-node enables to find the data blocks by their filename.

5.1.2 Network communication

The CedCom architecture is specially designed for Big Data applications and therefore, it handles a large quantity of data that is distributed within and across clusters. The data can be stored and migrated over the network if needed. Thus, an efficient communication protocol had to be implemented to enable faster communication between the nodes.

Since C++ provides a basic network implementation, we decide to use a high-level tool provided by the boost suite called *'Boost Asio'*.[1] This tool was selected because it provides pre-built functions that are easy to use at the network layer. However, *'OPEN MPI'* was initially selected.[2]. But because it facilitates only the parallelization of the task processing and not the transfer of data, the tool was not used.

Boost Asio provides a function to open *synchronous* or *asynchronous* connections between servers. It simplifies network utilisation by overloading the basic C++ functions. Combining 'Boost Asio' with basic functions of C++, the CedCom architecture provides functions to open a *socket* between a client and a server. In addition, it provides functions to write or read binary streams to the socket. The main advantage of using 'Boost Asio' is the portability.

5.1.3 Connection management

The CedCom architecture handles distributed data that are supplied by an external client. Currently, the architecture does not have any native client for loading data; rather, it relies on external clients more specifically, the application clients. The connection between client and nodes is established in two steps. First, the client requires the meta-information about the data nodes included in the cluster. To do so, it contacts the directory-node and fetches information such as the IP address, the TCP port, the storage space available, and the utilization ratio of memory. Based on the collected information, the client produces a data distribution plan. Then, the client opens connections with different compute-nodes and sends the data blocks to those nodes.

The size of the data packets is critical. Typically, it depends on the network bandwidth. The CedCom architecture enables receiving data with small blocks or a large blocks (e.g., 500 MB or more). For the small size blocks, the data received by the compute-nodes are aggregated immediately in the registration queue. For the large size blocks, the nodes use a specific *input buffer* which temporarily stores the incoming data blocks. In order to confirm the correct order of arriving data blocks, a unique number is assigned with each block which can be deemed as an *identifier* or a *tag number*. Upon receiving a block, the compute-node verifies the tag number to ensure the correctness. If the verification is not successful, the compute-node sends an '`error`' message which contains the block's tag number, to the client. The client uses it as a reference number and can find easily the invalid block and resend it to the node.

The compute-node automatically detects when the client leaves or closes the connection and therefore closes the socket, clear the input buffer, and start waiting for the next data blocks. The CedCom architecture does not use any close protocol. The client opens the connection with the node, send data blocks, and then disconnects itself.

The compute-node uses a 'registration queue' to add data blocks received by the node. This queue temporarily stores the blocks before adding them in the attraction memory of the nodes. The new blocks are stored in this queue, because they do not have their unique identifier when they were created. The directory-node assign unique identifiers

[1] http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html
[2] http://www.open-mpi.org/

to the blocks. They are then added to the attraction memory.

5.1.4 Replication management

Like Hadoop, the CedCom architecture uses the notion of data replication. However, the architecture would not allow creating replicas in attraction memories. Since, data can be migrated dynamically from node X to node Y, we argue that replication would not be necessary. However, replicas could be created in attraction memories dynamically to deal with a specific condition which is—two or more nodes need the same data blocks for processing tasks.

Typically, the CedCom architecture enables creating (maximum) one replica in secondary storages. In this case, it differs the replication policy of Hadoop. The notion of replication has been adopted in this architecture to make the system fault tolerant. For instance, if a node does not send heartbeat signal during a predefined timeout period, the directory-node automatically considers it a *inactive node*, and issue a command to a relative free compute-node to load the replicas of the inactive node into its attraction memory. The compute-node is allowed to store only one replica of a data block. When the restoration of data blocks are completed, the directory-node automatically update its own global index. If an update occurs, the replications of this block has to stay up to date. This update employs the principle of the *write-update* protocol.

5.1.5 Serialization

Serialization is critical to the CedCom architecture. The reasons are two-fold. First, it enables saving data in a known format. Second, it enables sending data easily from one node to another by merely sending the binary streams in the sockets.

To simplify the serialization process, we use the technology called *'Boost Serialization'*. It enables transforming a C++ class in a binary format or parse a binary stream in a specified class. More importantly, it enables choosing the variables of a class to be serialized and also the variables not to be serialized. This 'Boost Serialization' enables the serialization of a great part of the *std* container, like *vector* or *map*. Another advantage of this library is that it recursively serializes the classes. This means, if a variable is an instance of another serializable class, it will be easily serialized like a *primitive type* in the source code.

5.1.6 Configuration

The CedCom architecture is complex. Therefore, initializing parameters in a command line interface is non-trivial task. To simplify, a parameter file has been integrated with CedCom. The parameter file contains all the information required at the initialization phase. It will greatly help users (e.g., *administrators*) to initialize the compute-nodes and the directory-node. The parameters stored in those files are as follows.

- *Parameters for compute-nodes:*
  - The path of hard drive where the data is stored

  - The IP address of the directory-node
  - The communication port of the directory-node
  - The Heartbeat port of the directory-node
  - The transfer port use to establish a connection with an other compute-node
  - The port the client has to use to establish a connection
  - The Heartbeat interval delay
  - The Replication port

- *Parameters for directory-node:*

  - The path on hard drive where the data is stored
  - The communication port
  - The Heartbeat port
  - The port the client has to use to establish a connection

5.1.7 Client connection

The users will use the client applications to distribute data to the compute-nodes. Since currently no client component has been implemented in this paper, the users need to use application's client for data distribution. The application client contacts the directory-node to obtain the information required to establish connections with the compute-nodes. The directory-node has an asynchronous server that is used only to communicate with the client on a predefined port. This client receives various information from the directory-node such as:

- IP Address and port to open the communication;
- free-space storage;
- size of the block;
- memory utilization ratio.

This helps the client to partition the file into blocks of specific size expected by the compute-nodes. When the partition is completed, the data blocks are sent to compute-nodes.

5.2 Core components

The subsections below describes the details how the core components have been implemented.

5.2.1 The compute-nodes

The compute-nodes are important elements of the CedCom architecture. These nodes store data and execute jobs. Figure 3 shows the structure of a compute-node.

The CedCom architecture organizes memory in an unique style. In the subsection below, we describe the memory organization of the CedCom architecture.

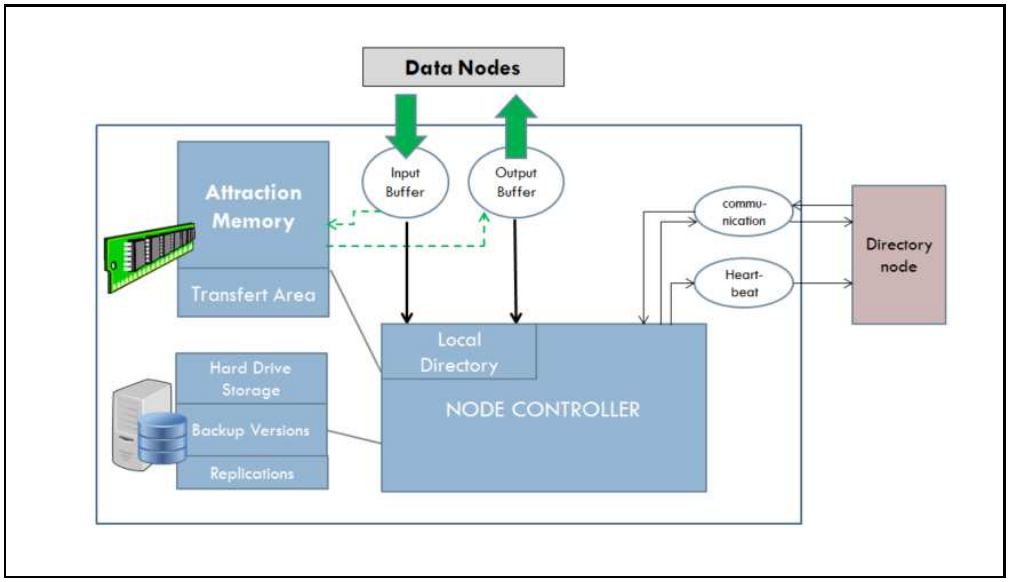


Figure 3: Structure of a compute-node

A. Memory organization Each node has its own independent memory. In this architecture, data is stored exclusively in main memory to improve the performance by reducing time needed to access the data. The CedCom architecture organizes different types of storage specifically, *Attraction Memory*, *Transit Area*, and *Secondary Storage* as a single unit. A new data block will first, attempt to be stored in attraction memory. However, as explained in Section 4, it will be stored in secondary storage if and only if the attraction memory of all the nodes in the cluster is filled with data blocks.

The transit area will store the *least recently used* and *ready to be migrated* blocks. This area is considered as the transit point for data migration. In fact, this is the reason we named this *Transit Area*. The implementation of these kinds of storage in the CedTMart system [2] are discussed in the following subsections:

A.I. Attraction memory The main purpose of attraction memory is to store the data blocks that have higher *cache hit* ratio. Cache hit refers to a successful attempt made by CPU to read or write a data block in the CPU cache. These data blocks are the much needed ones for processing tasks such as a *query*. The data is stored in attraction memory to enable faster access by processors.

A.I.a. Associative cache To implement the attraction memory, we used a technique called *multi-way associative cache*. This technique divides the memory slots into subsets and uses the *hash keys* to store and find the data. The term 'way' is used to define the maximum number of items each set can contain. This technique is also called *n-way associative cache* which can be written as follows, 2^N slots of the memory into x subsets, each one having 2^n slots, with $n < N$ and $x = \frac{2^N}{2^n}$. Figure 4 depicts an example of a two-way associative cache.

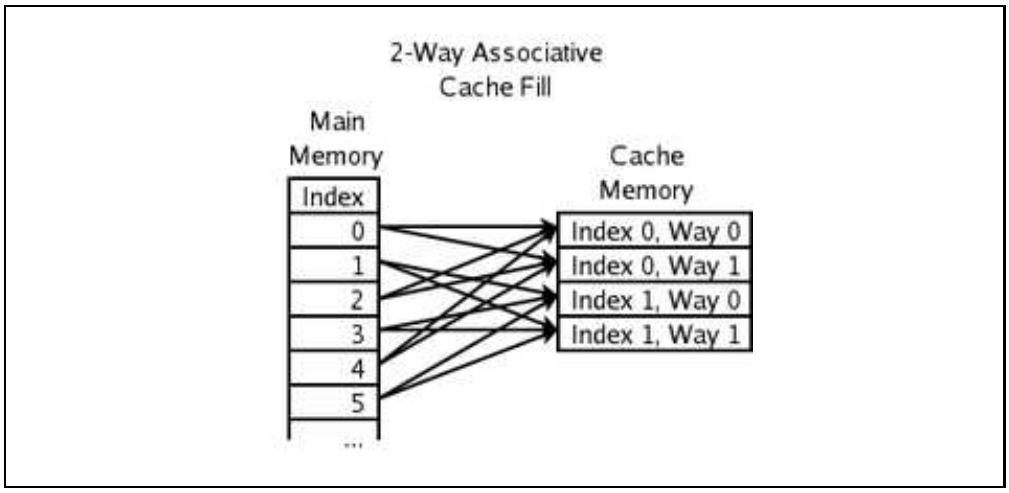


Figure 4: Two-way associative cache

We choose this technique because it provides a fast access to the data and a better hit ratio than a usual cache [7]. With the use of this approach, the complexity of finding a block in $O(n)$ reduces to $O(log(n))$ where n is the number of elements stored in the memory. A standard *list structure* was a potential candidate technology as well but it is not a feasible option because it results a systematic shift of all elements in the set ($O(n)$) which requires a lock during the execution. The associative cache uses the technology called *Map* that is more efficient. It organizes the data table comprises two columns: *key* and *value*. This enables faster data access. The CedCom architecture promotes the operational complexity as follows, find or insert an element $O(log(n))$ and delete $O(n)$ (without lock).

A.II. Transit area The size of attraction memory is limited. Thus, it is likely that an excessive number of blocks with the same hash key stored in the same node cause a buffer overflow. Since migrating data blocks to attraction memory of another node would generate high-traffic and storing blocks in hard drive is not a viable option, the blocks are moved to the transit area. The key purpose of implementing transit area is to increase the availability of attraction memory for the data blocks with high cache hit ratio.

The transit area is implemented with a single map which contains the blocks and uses their identifiers as *keys*. This enables users to find blocks, delete blocks, or add a new block with a complexity of $O(log(n))$. A specific class: `secondary_memory.h` gives some pre-built optimized functions to access, update, and delete different blocks that are available in the *Map*.

A.III. Secondary storage The CedCom architecture includes the secondary storage. The key purpose of this storage is to complement the attraction memory. The data blocks are sent to secondary storage when the attraction memories are full or above a predefined threshold. Since reading data blocks from secondary storage is time consuming, there is a trade-off using secondary storage. More specifically, data blocks will be stored in secondary storage at the expense of performances.

B. Local directory The local directory has been introduced in the CedCom architecture to index the locations and status of the data blocks contained in the compute-nodes. The locations are: *Attraction Memory*, *Transit Area*, and *Secondary Storage*. The local directory reduces time to read, find or write data blocks. Without indexation, these operations are time consuming because the system must scan all different locations sequentially. The directory provides exact locations of data blocks and hence, accelerate the speed to finding data.

C. Versioning system A version management system has been integrated in the CedCom architecture to avoid data loss. The architecture enables a node creating the *snapshots* of itself and storing them in its secondary storage. These snapshots are called *versions*. Each snapshot corresponds to a specific restoration point. However, storing the snapshot of a node can be expensive in terms of storage space. The storage can be overflowed. In order to avoid overflow of local storage, a few (e.g., currently 3) versions of nodes are kept in the secondary device. In case the storage is full, the oldest version will be removed.

D. Operating principle The operating principle of a compute-node is complex, because a number of parallel operations will be carried out by the node at runtime, especially maintaining the data coherency is non-trivial. Various *operational components* have been implemented for parallelising operations that a compute-node performs at runtime. They are described as follows:

D.I. ComLoader This is an important component of the CedCom architecture. It performs the primary operations such as starting or stopping the other components of a compute-node. It process instructions received from the directory-node. Also, it performs saving the snapshot of the recent state of compute-nodes. For instance, if a compute-node is *shutdown* normally, the *save* on *exit* option is popped up on the screen. Then, the ComLoader (stands for *Component Loader*) saves all the contents and node information as a file in secondary storage and reloads it when the node is restarted. The node information file is saved on the secondary storage usual way except for the fact that the new versions of the node will be created by the ComLoader.

D.II. Block manager To keep the data blocks contained in the compute-nodes consistent, we have implemented a *Block Manager*. The main tasks of this component are, moving or adding data blocks in memories. In CedCom, when data blocks are arrived, they are added in the input queue which store the blocks temporarily. The block manager calculates the hash keys of the incoming blocks and stores them in appropriate sets of attraction memories. If the set is full, the block manager automatically moves the oldest blocks of the set into *Transit Area*. This frees attraction memory to load new data blocks. Finally, the block manager updates the local directory to maintain a coherence between the block identifiers and their locations.

D.III. Transit-area manager The *Transit Area Manager* has been implemented to manage the blocks stored in the *Transit Area*. The main purpose of developing this component is to manage the transit area efficiently to make attraction memory available for the data blocks with high hit ratio. The key role of this component is managing the transit storage and also finding a host to migrate the blocks. However, the migration of a data block is performed on two conditions: *the data block is the least recently used one* and *the potential host node has sufficient storage available*. The transit area manager contacts the directory-node to find a compute-node that can host data blocks located in transit area. If the directory-node finds host then it returns the information to the requester node. Then the transit area manager transfers data blocks to the host node. The block stays in transit areas until a host node is found.

D.IV. Heartbeat manager This component is responsible for managing one-way messages from compute-node to directory-node. It triggers messages at a regular interval and sends to the directory-node. This signal is an indication to the directory-node that the compute-node is active. The message body contains various information such as *availability of free spaces* in the nodes and *the memory usage ratio*. Note that, the manager will not initiate connection with the directory-node on the Heartbeat port. This task is done at the node initialization phase.

D.V. Replication manager Two major functionalities of the replication manager are: (i) it enables storing replicas of data blocks in the secondary storage of a node, which are received from another node and (ii) it enables moving a replica of a data block to make it available to the attraction memory of a node where it is required to process a

job. The manager enables to access data in the replication area of secondary storage. No other data is allowed in this restricted area. The strict access control guarantees data consistency. In CedCom architecture, each node has a specific port to create an asynchronous server specifically for the replication.

The replications are managed by the directory-node and therefore, this particular component is deployed on this node. We developed a replication directory in our architecture. This directory contains information such as data blocks that need to be replicated. The manager contacts the host compute-node and requests to create replicas of the data blocks. The compute-node creates the replica and stores its in the *replication area*. It is worth mentioning that only one replica is allowed in the attraction memory of one compute-node.

D.VI. Backup manager A compute-node may need to be shut down. Since the CedCom is an attraction memory based architecture, the information of compute-nodes (such as the information about *Attraction Memory*, the *Transit Area*, and the *Local Directory*) and the data blocks which it contains must be persisted. CedCom stores the snapshot of compute-nodes in their secondary storages. The node information and data blocks are stored at a regular interval to save the most recent changes in the attraction memory. Also, the directory and the transit area are stored. Essentially, once the information are saved, the Backup Manager updates the information periodically rather than repeating the *write* of the same information on secondary storage. Since writing node information and data blocks are time consuming, the 'Block Manager' launch multiple threads to parallelize the task.

D.VII. Block-registration manager The 'Block Registration Manager' is responsible for registering the inbound data blocks from the clients. It contacts the directory-node and assigns a unique identifier to the new blocks. The CedCom architecture enables generating several threads to perform this task in parallel.

5.2.2 The directory-node

This subsection describes the directory-node of the CedCom architecture. This is essentially a shared metadata server in our architecture. In CedCom, each compute-node can act as a local metadata server because, each of them has a local directory that supplies a limited amount of meta-information. However, the information is purely about the data blocks stored in local node. Thus, a global metadata server has been developed in our architecture to deal with some specific situations. For instance, the local metadata servers usually do not have sufficient information about the data blocks stored in other nodes in the cluster. Conventional COMA does not have any directory-node, rather, it relies on local metadata. In this case CedCom architecture adopts Hadoop's architectural principle *a global metadata server that receives requests coming from the compute-nodes.*

Big Data applications can generate billions of blocks distributed to thousands of nodes. Thus, storing global metadata in compute-nodes may not be a viable option for two

reasons: it will consume the attraction memory or transit area and maintaining an updated distributed metadata server is a non-trivial task and computationally expensive. The former is a well-understood problem whereas the latter is communication network related problem. For each write, all metadata servers have to be updated, which will promote an exhaustive number of messages exchanged between the nodes. This will cause a high traffic in the network. Consequently, the job processing time will be increased. In some case, processing time can be increased increased dramatically.

Considering these issues, the CedCom architecture introduces a specific node to take the role of storing and managing a *Big Directory* which stores the location of all the blocks distributed across the compute-nodes. We called this node *'Directory-Node'*. This node is aware of both locations of the different blocks and active blocks which is a similar approach to Hadoop [12] but more simplified. Additionally, this node also manages the block replication system. Figure 5 shows the directory-nodes.

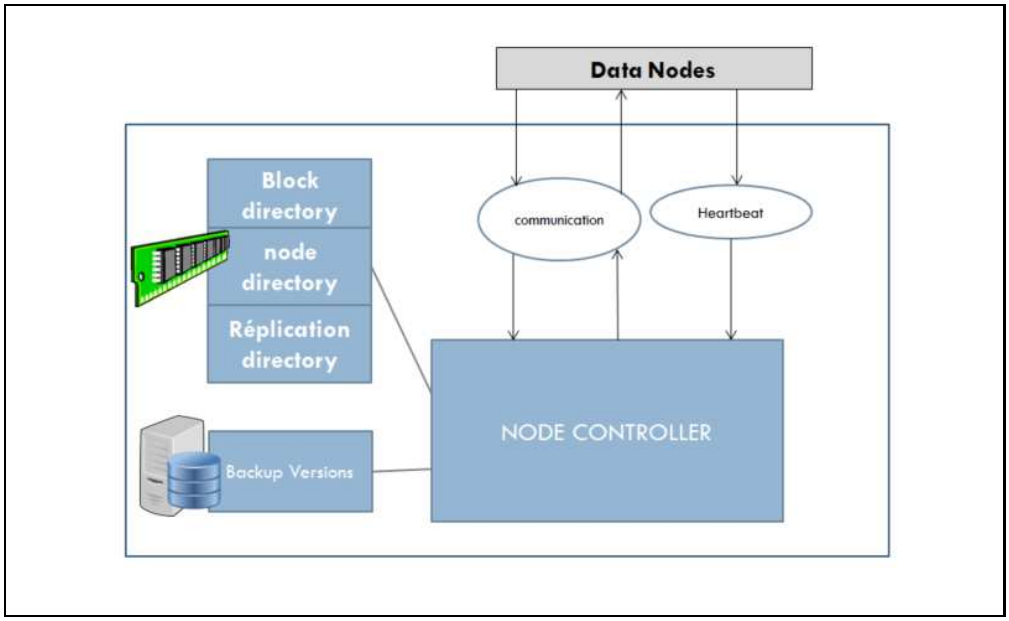


Figure 5: Structure of the directory-node

Below, we describe different aspects of the directory-node. They are organized as follows.

A. Server operations

A.I. Communication connection

A.I.a. Command requests
A.I.b. Data requests
A.I.c. Communication protocol

A.II. Heartbeat connection

A.II.a. Heartbeat protocol

A.III. Operating principle

A. Server operations The directory-node acts as a metadata server. The compute-nodes are connected with them over the peer-to-peer technology. The connection between the directory-node and the compute-nodes are permanent. The connection has been classified into *Communication Connection* and the *Heartbeat Connection*. Note that, the directory-node communicate with compute-nodes asynchronously. Therefore, the node is able to receive and manage hundreds of requests from hundreds of nodes simultaneously, using multi-threading techniques.

A.I. Communication connection The compute-nodes may contact the directory-node in many occasions such as to know about the location of a data block. The *Communication Connection* has been implemented to establish connections efficiently between the compute and directory-nodes. In order to establish connection, first the directory-node checks *ip address* of a compute-node in the directory. It is worth noting that the ip addresses are stored in the directory-node through a registration process that is done when a compute-node is connected with directory-node for the first time. If an ip address is not found in the directory, the directory-node starts registration process. It creates a new unique node identifier and gives it to the compute-node which opens the new connection. If an error occurs during this phase, the compute-node is immediately turned off. Once the connection is successfully established, the directory-node uses this connection to perform the following tasks:

- turning off the node (with or without save)
- duplicating data blocks
- restoring data blocks in the attraction memory or transit area
- creating restoration point of compute-nodes
- restoring a compute-node by reading the node information snapshot which was taken while switching off the compute-node

On the other hand, the compute-node uses the socket to do the following actions:

- requesting information about a block location
- requesting the IP address of a specific node
- register a new block, or a group of blocks
- disconnect the node from the directory-node
- sending an update signal to notify the move of a block in the local attraction memory

We implemented a protocol to handle the requests from compute-nodes to the directory-node. The request messages must comply the protocol. It is worth noting that in CedCom architecture the requests messages are categorised into *command requests* and *data requests*. They are briefly explained in the following.

A.I.a. Command requests The commands requests are concerned with requesting information or operations. The size of the body of the command request messages is small. It does not exceed fifty characters. The format used for this type of requests is as: *a four characters header* for writing the operations and the remaining are for instructions. *Regex* is used to identify and extract information from the request messages.

A.I.b. Data requests The data requests are concerned with sending the data blocks from one node to another. Unfortunately, the Regex technology is slow for data requests because of the size of messages' body is large. Thus, analysis of the message takes longer period. Note that, upon receiving a request message, it is analysed. We cleverly avoid analysing the body of data request messages by separating the it from the header. The CedCom architecture allows only the header to be analysed. Currently, the separator used is '`||-||`'. The frequency of its occurrence is very close to zero.

A.I.c. Communication protocol The CedCom architecture has its own low-level communication protocol. The protocol presented below has been implemented so far:

- The compute-nodes open the connection with the directory-node by using the asynchronous connection provided by '`boost.asio`'.
- The directory-node registers the connection by assigning a unique node identifier corresponding to the IP address of the compute-nodes
- The directory-node sends the registration notification to the compute-node:

  - If successful—the identifier of the node:

  ```
  RSTR:N<(node_identifier)>;
  ```

  - If failing—an error message:

  ```
  RSTR:ERR:(error_message);
  ```

- The compute-node will use this connection to request for information about the location of a block:

  - Request:

  ```
  BLCK:BLOCK<(block_identifier)>;
  ```

  - Response:

  ```
  BLCK:BLOCK<(block_identifier)>,
  NODE<(node_identifier)>,
  ADDRESS<(ip_address:port)>;
  ```

- The compute-node will also use this connection, to obtain a group of unique identifiers for the new blocks received from an external client:

  - Request:

  ```
  BLRG:RANGE<(number)>;
  ```

  (Range is used to determinate the number of identifiers requested.)

  - Response:

  ```
  BLRG:MIN<(min_identifier)>,MAX<(max_identifier)>;
  ```

- The directory-node can use this connection to turn off a compute-node with or without previously saved contents:

  ```
  STOP:SAVE<true/false>;
  ```

A.II. Heartbeat connection In our architecture, a separate connection type has been implemented to send heartbeats from the computing nodes to the directory-node. Heartbeats are the signals to the directory-node that the compute-nodes are *alive* or *dead*. As mentioned earlier, the compute-nodes send *heartbeat* signal at a regular interval.

To register a heartbeat connection, the nodes must be already registered with the directory-node. The registration will result in opening a socket between the nodes (a computing node and the directory-node). Using this socket the compute-nodes send signals to the directory node. If the registration of a compute-node is unsuccessful, it will be turned off and an error message will pop up.

As mentioned in the beginning of this section, in addition to the heartbeat signal, the compute-nodes send other information of itself. In particular, the following information are transmitted in each heartbeat:

- Free Space of the nodes
- Memory utilisation ratio of the nodes

The directory-node generates and persists metadata and the time-stamp of a compute-node. For instance, it tracks the latest time-stamps of heartbeats and stores it as metadata An arbitrary time is defined as *timeout* which allows the directory-node to determine whether the node is offline. This timeout is currently set to *5 seconds*.

A.II.a. Heartbeat protocol We implemented a protocol for the communication concerning the heartbeats between compute-nodes and the directory-node. This protocol is used for communicating with the heartbeat port which is implemented specifically for the communication about heartbeats . The protocol is briefly explained in the following:

- The compute-nodes open the connection with the directory-node by using the asynchronous connection provided by 'boost.asio'.
- The directory-node registers the *Heartbeat* by associating the IP address of the node with its unique identifier
- The compute-nodes send an initialization message to the directory-node to inform about the port which the (external) client should use to establish a connection:

  ```
  INIT:C_PORT<(Port_number)>;
  ```

- The compute-node sends the heartbeat message at a regular interval along with the information of its memory use:

  ```
  BEAT:FREE_SPACE<(size_in\Mo)>,RATIO<(percentage_number);
  ```

- The directory-node receives heartbeats, stores information, and updates the latest heartbeat time-stamps.
- The directory-node triggers an error message upon occurrence on an error:

  ```
  ERRO:(error_message);
  ```

- If needed, the directory-node uses the *Heartbeat* connection to turn off a node, with or without previously save its content:

  ```
  STOP:SAVE<true/false>;
  ```

A.III. Operating principle The operating components and principles of the directory-node are explained below.

- Like compute-nodes, the directory-node has a component called *ComLoader* that starts and stops the other components of the directory-node. In addition, this component is used to save the content of the directory-node on secondary storage if it is to be turned off. Compare to the *ComLoader* of the compute-nodes, the process of storing the directory-node information on secondary devices is more systematic.
- The directory-node has a component for checking whether or not the computing nodes are alive. It checks the status by comparing the latest time-stamp with the predefined timeout. If a node is not alive, this component puts it in the *restoration node queue*. The nodes should be restored efficiently. To do so, the CedCom architecture initiates a new thread to speed up the restoration process.
- The CedCom architecture has a component for managing the reload of the blocks. If a compute-node stops responding, this component will restore all of its blocks in memory. The reloading process is composed of four steps. In the first step, the component identifies the data blocks that were stored in the failed compute-node. Then, it identifies the compute-nodes that have the copies of the blocks. In the third step, it sends a request to these nodes to transfer the blocks to the failed compute-node. Finally, it updates the directory.

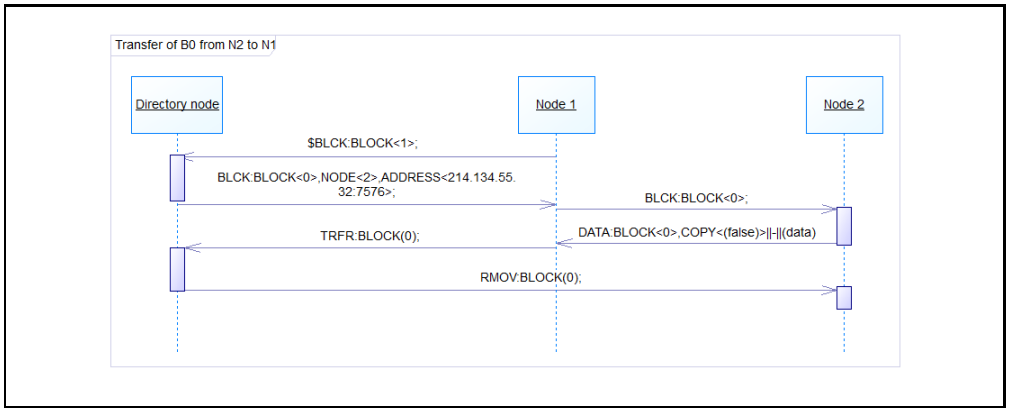


Figure 6: Transfer of a block of data between two nodes

5.3 Advanced operations

This section presents a list of advanced operations performed by the components which we have developed in this paper. The CedCom architecture enables performing these operations during transferring and creating data blocks. They are briefly explained in the subsections below.

5.3.1 Block transfer

This section summaries the steps of transferring blocks from the source compute-node to the target compute-node. Figure 6 shows an example of block transfer between two nodes.

The steps are listed below:

- The compute-nodes use the communication connection with the directory-node to request the location of the block needed:

  ```
  BLCK:BLOCK<(block_identifier)>;
  ```

- The directory-node uses the block directory to locate the specified block, and then sends the response to the compute-node:

  ```
  BLCK:BLOCK<(block_identifier)>,
  NODE<(node_identifier)>,
  ADDRESS<(ip_address:port)>;
  ```

- A compute-node opens a temporary connection with the compute-node that is hosting the required data blocks, and requests for the blocks by using its identifier:

  ```
  BLCK:BLOCK<(block_identifier)>;
  ```

- Alternatively, a compute-node requests for processing a job to the compute-nodes which have the copy of the data blocks needed to process the job. However, in this case the host compute-nodes must be free. The requester and the host compute-nodes will process the transmission of the computations instead of transferring the data packets.

  ```
  BLCK:TRANS_COMP_REQ;
  ```

- If a block is transferred or just copied, the source nodes send the block by using the standard data package format and specify in the header of the package.

  ```
  DATA:BLOCK<(block_identifier)>,COPY<(true/false)>||-|||(data)
  ```

- Upon arrival, a (requester) compute-node stores the data blocks in its attraction memory and then sends a notification of the transfer to the directory-node which updates the latest location of the blocks.

  ```
  TRFR:BLOCK(block_identifier);
  ```

- The directory-node updates its block directory and sends a *invalidation* request to the compute-nodes which have the old copies of the data blocks. Upon receiving the request, the source nodes invalidate the copies. This avoid the data inconsistency and hence, the *dirty read*.

  ```
  RMOV:BLOCK(block_identifier);
  ```

5.3.2 Block creation

This section summaries different steps which are carried out when an external client sends data blocks to the compute-nodes. Figure 7 data transfer from an external client.

- The client establishes a connection with the directory-node which sends information of how to contact the compute-nodes

  ```
  B_SIZE<(size)>,[{"address":"(ip_address)"
                  ,"port":"(port)"
                  ,free_space":"(size)"
                  ,"ratio":"(percentage)"}
                  ]
  ```

- The client establishes a connection with the compute-nodes contained in the clusters. It partition the input files into different packages. Each package can contain a maximum data size that is suggested by the directory-node. The partition may result in a single package or many smaller packages, which depend on the size of the input files and the predefined maximum size of the package.

  - If data is sent as a single part:

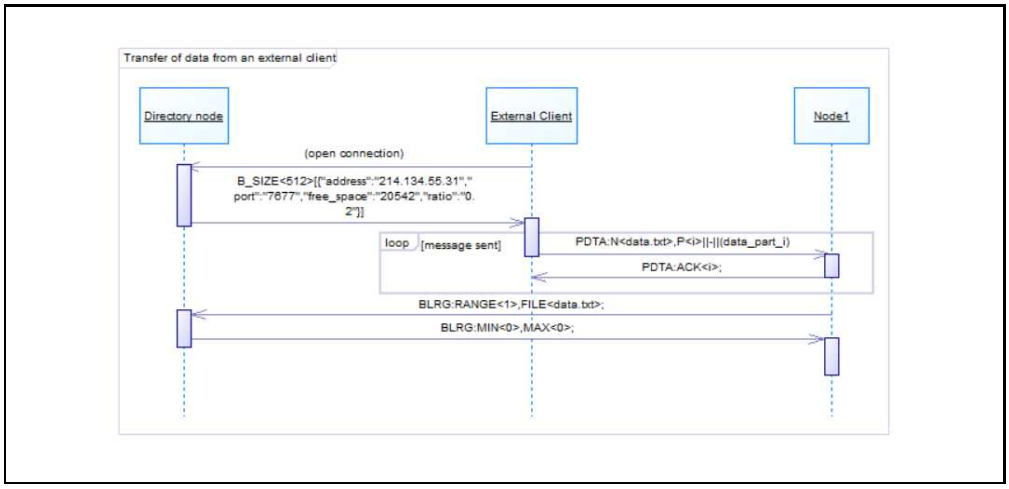


Figure 7: Transfer of data from an external client

`DATA:N<(filename)>||-|||(data)`

- If data is divided into many smaller packages:

`PDTA:N<(filename)>,P<(part_number)>(LAST)||-|||(data)`

- To ensure data integrity and flow control, the compute-nodes send an ACK signal to the client for each message received.

  `PDTA:ACK<(part_number)>;`

- The data blocks received by the compute-nodes must be registered with the directory-node. By using a system of range, a node can obtain successive identifiers. The input filenames are transferred as well to associate the filenames with the data blocks in the internal directory of the compute-nodes. This essentially indicates that the node will always group them by filename during block transfer.

  `BLRG:RANGE<(nb_block)>,FILE<(filename)>;`

- Upon receiving the registration request from the compute-nodes, the directory-node provides the block identifiers to the requesting compute-nodes.

  `BLRG:MIN<(identifier)>,MAX<(identifier)>;`

6 Experimentation

In this section, we describe the experiment that we conducted with the CedCom architecture. Note that, we conducted experiment on a single-node cluster, although the CedCom architecture was developed specifically for large cluster.

6.1 Experiment setup

We conducted experiment to test only two functionalities of the CedCom architecture. We tested its ability to a) host Big or Blinked Data applications and b) store the data blocks efficiently [5].

The cluster contains one physical node which acts as both directory-node and compute-nodes. The specification of the physical node are given below:

- *Specification*
  - Processor: Intel i7
  - Memory: 8GB
  - Hard Disk: 1 Terabyte
  - Operating System: Windows 7

We used a 10 GB dataset (containing approximately 15000 Notation 3 (N3) files) in our experiment. [3] During the experiment, the following steps were carried out:

Step 1: The data distribution client opens a connection with directory-node (DN). The steps listed below were performed after the connection had established.

1. The client contacts the directory-node on a specified address and a specified port. The client message contains the amount of data it wants to distribute to the compute-node.
2. The directory-node receives the message scan the metadata table to find available space and sends a pre-formatted string to the client containing the following information:
   - Maximum size of one data block (the default value is 512 MB)
   - For each node: free space size, IP address and port

   In our experiment, the directory-node sends an JSONArray (to the client) containing multiple JSONObjects. The JSONObjects are of the following form:

```
[{"ip":"192.168.0.1"
 ,"port":"8888"
 ,"free_space":"20480"
 , "ratio":"0.5"
 }
,{"ip":"192.168.0.2"
 ,"port":"8686"
 ,"free_space":"200000"
 ,"ratio":"0.3"
 }
]
```

[3] Notation 3: [4]

The IP and PORT are used to establish the data connection between client and the compute-node. The client's distributor sends data to compute-nodes according to the value given for "`free_space`" (in MB) and "`ratio`" (that are shown in the above).

3. The client stops the connection with directory-node

Step - 2: The client decides the strategy to distribute the data with the previous information

Step - 3: Then the client opens a connection with the (only) compute-node and distribute the data blocks.

Figure 8 shows the data distribution.

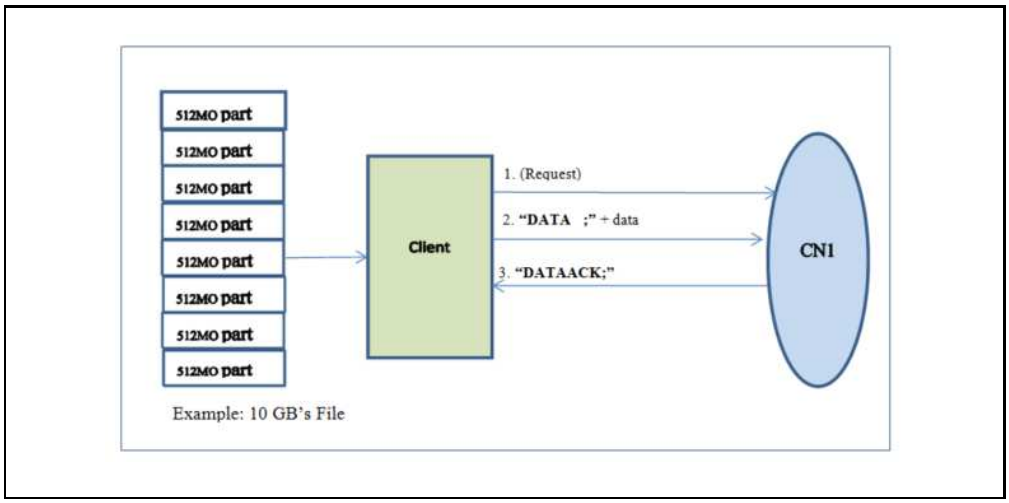


Figure 8: Data distibution

The following steps are performed by the nodes to distribute the data-blocks data.

1. The client opens the socket with the (only) node it wants to contact using the same request header.

2. Then the client sends data to the compute-node. A loop was defined to distribute data iteratively until all the blocks are sent to the compute-node. Two important issues regarding data distribution are listed below.

   - The client partitions input files into many parts according to the maximum size 512 MB suggested by the directory-node. The client sends the data packets begin with a header "`DATA; (8 bytes)`."
   - The compute-node receives the data, stores it and sends a message begins with "`DATAACK; (8 bytes)`" to the client.

3. Once the client closes the socket connection, compute-nodes knows directly that the whole file has been transmitted.

We observed that the 10 GB datasets took approximately 909 seconds. The transfer rate was 22 MB per second.

6.2 Discussion

The experiment showed that the dataset was deployed on the CedCom architecture successfully. The connection between CedCom architecture and the application was done successfully. We observed that the application client was able to establish connection successfully with the directory-node for requesting metadata. We also observed that the client was able to contact the compute-node to stores files in those nodes as data blocks. We found our architecture is able to prevent of data loss while the reception buffer is full.

It is worth noting that, the data are transferred as simple string and are not be encrypted. Additionally, unfortunately, we could not test the data migration which we will be doing in near future.

Another important conclusion we made through this experiment. Since our experiment was on a single-node cluster, the speed to data distribution was quite satisfactory. However, we do not confirm the same performance for a large cluster before testing the architecture.

7 Conclusion and Future Work

In this technical report, we implemented and presented the CedCom architecture. We provided the detail the existing technologies. We found COMA and HDFS are the potential technologies which can reused to develop our architecture. Essentially, the architecture has been developed by amalgamating COMA and Hadoop's architectural principle. The key objective of this architecture is is to leverage the power of COMA to improve the performance of Big Data or Blinked Data applications. Several components have been developed in this paper. We implemented components of Directory-Node and Compute-Node. Also, we implemented attraction memory using an efficient technique called n-way associative cache. This enables high-speed data access and increases the cache hit ratio significantly. Additionally, we implemented the data migrator that enables migrating data blocks automatically from the source to the target nodes in the clusters.

This architecture offers many functionalities. However, several works must be done to make it a complete product. Some notable works which we planned to carry out near future are as follows. First, we will conduct a rigorous test to evaluate all the functionalities of the architecture. Second, we will replace the set associative cache with the skewed associative cache which is a better approach. Third, we planned to develop a component that can migrate data in intelligent way such as by adapting the bandwidth. Fourth, we will complete the development of the protocols of the CedCom architecture.

References

[1] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990. [Available online[5]].

[2] Minwei Chen, Rafiqul Haque, and Mohand-Saïd-Hacid. CedTMart—A Triplestore for Storing and Querying Blinked Data. $\mathcal{CEDAR}$ Technical Report Number 7, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, July 2014. [Available online[6]].

[3] James R. Goodman. Using cache memory to reduce processor-memory traffic. In Harold W. Lawson Jr., Tilak Agerwala, Hans H. Heilborn, Hideo Aiso, Lars-Erik Thorelli, Jean-Loup Baer, and Mario Tokoro, editors, *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA'83)*, pages 124–131, Stockholm, Sweden, 1983. ACM. [Available online[7]].

[4] Erik Hagersten, Anders Landin, and Seif Haridi. DDM—a cache-only memory architecture. *Computer*, 25(9):44–54, 1992. [Available online[8]].

[5] Rafiqul Haque and Mohand-Saïd Hacid. Blinked Data: Concept, characteristics, and challenges. In *Proceedings of Service Congress 2014 (to appear)*, 2014.

[6] Henk L. Muller, Paul W. A. Stallard, and David H.D. Warren. The data diffusion machine with a scalable point-to-point network. Technical Report Number CTRS 93-17, University of Bristol, Department of Computer Science, Bristol, UK, October 1993. [Available online[9]].

[7] Henk L. Muller, Paul W. A. Stallard, and David H.D. Warren. The role of associative memory in VSM architectures: A price-performance comparison. Technical Report Number CTRS 95-009, University of Bristol, Department of Computer Science, Bristol, UK, October 1995. [Available online[10]].

[8] Henk L. Muller, Paul W. A. Stallard, and David H.D. Warren. Implementing the data diffusion machine using crossbar routers. In Kai Hwang, editor, *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 152–158, Washington, DC, USA, April 1996. IEEE Computer Society. [Available online[11]].

[9] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazìres, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, December 2010. [Available online[12]].

[10] Sanjay Raina. Virtual shared memory: A survey of techniques and systems. Technical Report Number CSTR-92-36, University of Bristol, Department of Computer Science, Bristol, UK, December 1992. [Available online[13]].

[5] http://www.cs.ucr.edu/~bhuyan/CS213/2004/LECTURE11a.pdf
[6] http://cedar.liris.cnrs.fr/documents/ctr7.pdf
[7] http://courses.cs.vt.edu/.../TransactionalMemory/Goodman-SnoopyProtocol.pdf
[8] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.6673
[9] http://www.cs.bris.ac.uk/Publications/Papers/1000019.pdf
[10] http://www.cs.bris.ac.uk/Publications/Papers/1000065.pdf
[11] http://dl.acm.org/citation.cfm?id=645606.660867
[12] http://web.stanford.edu/~ouster/cgi-bin/papers/ramcloud.pdf
[13] http://www.cs.bris.ac.uk/Publications/Papers/1000011.pdf

[11] Tanguy Raynaud. A cache only memory based architecture for big data applications. Master's thesis, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, July 2014.

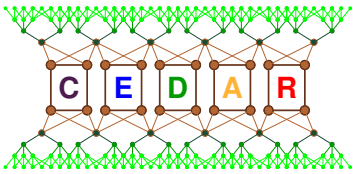
[12] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, Washington, DC (USA), 2010. IEEE Computer Society. [Available online[14]].

[13] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3rd edition, 2012.

[14] Paul C. Zikopoulos, Chris Eaton, Dirk deRoos, Thomas Deutsch, and George Lapis. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill, 2011. [Available online[15]].

[14] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.989

[15] http://public.dhe.ibm.com/common/ssi/ecm/en/iml14296usen/IML14296USEN.PDF



Technical Report Number 8

A Cache Only Memory Architecture for Big Data Applications

Tanguy Raynaud and Rafiqul Haque

July 2014