Native support for parallel and distributed computing by the network

Junior Dongo

UNIVERSITÉ PARIS-EST

École doctorale MSTIC

Native Support for Parallel and Distributed Computing by the Network

THÈSE

présentée et soutenue publiquement le 22 Mai 2020

pour l'obtention du

Doctorat de l'Université Paris Est

(mention informatique)

par

Junior Dongo

Composition du jury

Rapporteurs :	Pascal Lorenz	GRTC - Université de Haute-Alsace (Professeur)
	Martin Theobald	University of Luxembourg (Professeur)
Examinateurs :	Daniele Varacca	LACL - Université Paris Est (Professeur)
	Chantal Keller	LRI - Université Paris-Sud (Maîtresse de conférence)
Encadrant :	Charif Mahmoudi	Siemens Corporate Technology (Software Architect)
Directeur :	Fabrice Mourlin	LACL - Université Paris Est (Maître de conférence HDR)

Laboratoire d'Algorithmique, de Complexité et Logique — EA 4219

Mis en page avec la classe thesul.

Acknowledgments

First, I would like to thank my thesis advisor Fabrice Mourlin for agreeing to supervise this thesis. Thank you for your kindness, your availability, your advice, and your permanent encouragement. This thesis owes you a lot. Thank you for believing in me.

I also thank Charif Mahmoudi, who supervised me throughout this thesis and who shared his brilliant ideas. Thank you also for the various invitations for my stay in the United States (3 months at the National Institute of Science and Technology, and 6 months at Siemens Corporate Technology). I have learn a lot from you and thanks to you. Thank you for this confidence in me.

I would also like to thank Pascal Lorenz and Martin Theobald for agreeing to review this thesis.

Thanks to Daniele Varacca, Chantal Keller for agreeing to be part of my jury.

I extend my sincere thanks to the LRI verification team, in particular to Véronique Benzakem, Evelyne Contejean and Chantal Keller for their availability and their great help in my verification approach in Coq. This part of my thesis was possible thanks to you.

Many thanks to Kevin Mills and James Filliben from NIST for introducing me to the sensitivity analysis and for their help in the evaluation phase of my approach.

My thanks also go to my colleagues at LACL, in particular, Flore Tsila and Nicolas Herniou, thank you for always ensuring that I do not miss anything in the accomplishment of my work.

I thank my mother, my brothers, and sisters for their encouragement and their prayers.

Thanks to all those whose names I did not mention, who supported me by any means during these thesis years.

Finally, a big thanks to the one who on October 12, 2018, became my wife. Thank you for your support. Thank you also for this beautiful gift (Jayden Nathanael) that you gave me.

I dedicate this thesis
to the Lord for his support
and help till where I am now.

Abstract

In the Big data community, MapReduce has been considered as one of the main approaches to answer the permanent increasing computing resources demand imposed by the large data amount. Its importance can be explained by the evolution of the MapReduce paradigm which permits massively parallel and distributed computing over many nodes.

Information-Centric Networking (ICN) aims to be the next Internet architecture. It brings many features such as network scalability and in-network caching by moving the networking paradigm from the current host-centric to content-centric where the resources are important not their location. To benefit from the ICN property, Big Data architecture needs to be adapted to comply with this new Internet architecture. One dominant of these CCN architectures is Named Data Networking (NDN) which has been financed by the American National Science Foundation (NSF) in the scope of the project Future Internet Architecture (FIA).

We aim to define Big Data architecture operating on Named Data Networking (NDN) and capitalizing on its properties. First, we design a fully distributed, resilient, secure and adaptable distributed file system NDFS (NDN Distributed File System) which is the first layer (Data Layer) in the Big Data stack. To perform computation on the data replicated using a Distributed File System, a Big Data architecture must include a Compute Layer. Based on NDN, N-MapReduce is a new way to distribute data computation for processing large datasets of information. It has been designed to leverage the features of the data layer. Finally, through the use of formal verification, we validate our Big Data architecture.

Keywords: NDN, Big Data, DFS, Distributed Computing

Résumé

Dans la communauté du Big Data, le squelette MapReduce a été considéré comme l'une des principales approches permettant de répondre à une demande permanente et croissante des ressources informatiques imposées par des données massives. Son importance peut être expliquée par l'évolutivité du paradigme MapReduce qui permet une exécution massivement parallèle et distribué sur un grand nombre de noeuds de calcul.

Le réseau centré sur l'information (ICN) vise à être la prochaine architecture Internet. Il apporte de nombreuses fonctionnalités telles que la mise à l'échelle du réseau et l'utilisation de cache réseau en déplaçant le paradigme de communication actuel centré sur l'hôte, vers une approche de communication centrée sur la donnée, où les ressources sont importantes et non leur emplacement.

Pour tirer profit des propriétés des réseaux ICN dans le domaine Big Data, l'approche d'architecture Big Data doit être adaptée pour se conformer à cette nouvelle architecture réseau. L'une des dominantes de ces architectures ICN est Named Data Networking (NDN) qui a été financée par l'American National Science Foundation (NSF) dans le cadre du projet d'architecture de l'Internet du Future (FIA).

Notre objectif est de définir une architecture Big Data fonctionnant sur NDN et capitalisant sur ses propriétés. Premièrement, nous concevons un système de fichiers distribués NDFS (NDN Distributed File System) entièrement distribué, résilient, sécurisé et adaptable qui est la première couche (Data Layer) de la pile Big Data. Pour effectuer le calcul sur les données répliquées à l'aide d'un système de fichiers distribués, une architecture Big Data doit inclure une couche de calcul. Basé sur NDN, N-MapReduce est une nouvelle façon de distribuer le calcul de données pour traiter de grands volume de données. Il a été conçu pour tirer profit des fonctionnalités de la couche de données. Enfin, grâce à l'utilisation de la vérification formelle, nous vérifions un ensemble de propriétés temporelles sur notre architecture Big Data.

Mots-clés: NDN, Big Data, Calcul parallele, Calcul Distribué.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This thesis has for title "Native Support for Parallel and Distributed Computing by the Network" and deals with how future distributed systems should be designed and operate due to the changes introduced by the evolution of computer networks to Information-Centric Networking (ICN) protocols, and more specifically Big Data applications over Named Data Networking (NDN). In this introduction, we first present the context and the motivations for this subject, then we present the problem we address. Finally, we briefly present the specific contributions of this work and how this document is organized. Our main objective is to provide useful information that helps the reader to familiarize themselves with Named Data Networking, Big Data, and also how to perform formal verification of systems from these domains.

1.1 Context and thesis motivation

Introduced as a Future Internet Architecture, NDN (Named Data Networking) is deeply changing the way network communications are performed. Applications have to adapt to this new networking paradigm and these adoptions sometimes imply a complete change of how applications are built.

This is the case for IM (Instant Messaging) for which, a new implementation has been proposed with a serverless design approach, that enables IM clients to chat with each other without infrastructure support over NDN [1].

Video Streaming is a widely used application over Internet today. An implementation of NDNVideo was proposed for video streaming, it can provide reliable and rate-adaptive playback with no session negotiation necessary between parties using NDN [2].

Big Data is a hot topic for almost all domains. In fact, every domain is looking for a way to gain insight from data or simply find a way to manage huge volume of data. One of the most used architecture in Big Data is the Lambda architecture [3]. This architecture has three layers: the batch layer, the speed layer, and the service layer. Incoming data go through both the batch layer and the speed layer. The data in the batch layer are immutable, it is never updated and new data are added to the end of the file. Results that meet the business requirements are precomputed using distributed processing systems. The speed layer deals with real-time data flow processing.

In this layer, stream processing frameworks such as Spark Stream or Storm are used. In the service layer, the data from the batch views and those from the speed layer views are merged. The Lambda architecture is independent of the technologies to be implemented. Beyond the different architectures and technical solutions, to provide added value, we need to understand the data lifecycle management. This data management can be viewed in two layers: a Data layer used as a storage and a Compute layer used to perform computation on the stored data or data coming in real-time. Both of them work at the application level when considering TCP/IP communication. NDN enables in-network caching, meaning that the data are meaningful at the network layer. Data here represent information from files or applications, but also the script of any program and more generally any useful resource. This is a great advance in networking and it also offers great possibilities such as moving the data management from the application layer to the network layer. Managing data at the network level may have a significant impact on data processing and also data availability. Data locality becomes then an important property. This brings new challenges such as data freshness. Some data last longer than others, whereas freshness is a constant associated with the data.

The objective of this thesis is to propose a Big Data architecture based on NDN. This will respond to two subgoals: first, fill a gap concerning Big Data over NDN, and provide a continuity of services; being able to perform duties while moving from TCP/IP to NDN (adopting NDN as communication mechanism). We also capitalize and leverage NDN dissemination capabilities to improve how Big Data is performed today (for example, dealing with small files). We first propose a Distributed File System which we called NDFS (Named Data Networking Distributed File System) for the data layer and also a Computation Distribution mechanism based on the MapReduce paradigm called NMapReduce (Named Data Networking MapReduce).

1.2 Problem statement

In this doctoral thesis, we study the new content-centric architecture designed and proposed to be the basis for the future Internet architecture. Named Data Networking architecture addresses the data rather than the hosts of the network, implying that the routing mechanisms, applications used in IP networks are no longer adequate for this new architecture.

Thus, NDN can not be fully functional without a redesign of applications and can not be deployed to replace the current architecture of the Internet. In fact, the removal of IP addresses in the NDN approach and the adoption of a content-centric communication lead to data dissemination on the network instead of a point to point communication pattern. Existing applications can't function as they are, especially applications needing network communications. It is therefore essential to propose new application mechanisms that are adapted to this network architecture. This redesign will consist of changing the way applications are built, in such a way that the network communication matches the one provided by NDN.

We focus in this thesis on Big Data architecture, namely a Distributed File System and a Computation mechanism.

This architecture must also take advantage of new features provided by NDN architecture, such as content caching. In-network caching is a technique that helps to accelerate content distribution. It consists of storing part of data at the network level and then use it to satisfy future requests for the data. There exist cache replacement mechanisms that are used to manage the content of the cache when it gets full.

1.3 Contributions and thesis plan

In this PhD thesis, we study the Named Data Networking Architecture and we propose a new Big Data architecture based on this network architecture.

This is done by studying the problem considering different aspects. First, we start by defining a software architecture using formal specifications. We then prove properties on the specification to ensure invariants in the specified behaviors. Implementations for experimental validation of the expected results are performed and finally, we consider performing measurements for comparison purposes.

In this manuscript, we start by presenting in Chapter 2 the state of the art of Big Data, Distributed System, Distributed Computing, and introducing the Named Data Networking, its mechanism and methods of content distribution. Our Big Data architecture (NDFS and NMapReduce) and a formal specification of its components in Coq are presented in Chapter 3. In Chapter 4, we formally verify our architecture using formal methods. We prove properties about it using the UPPAAL model checker. Chapter 5 presents the implementations of our architecture and we evaluate the performance of our approach using simulation experiments on a prototype version to highlight the main factor impacting the system. In Chapter 6, we evaluate our solution using some experimentations and present the different results which are then compared with the Hadoop platform. We also consider three use cases where our approach is used in real-life situations.

Finally, we conclude this work in Chapter 7 where we summarize our contributions and present possible future research directions. We also list the various publications performed during this thesis period.

Chapter 2

State of the Art

Contents

In this chapter, we first present the Big Data domain and its challenges. Then, we present the two main components of Big Data architecture, which are a Distributed File System (DFS) (Section 2.2) and a Distributed Computing (2.3), and discuss the state of the art related to each research domains. Finally, we conclude with a description of the Named Data Networking paradigm (Section 2.4).

2.1 Big Data

In this section, we define Big Data as it is pertinent to our work. We highlight its characteristics, its main applications and the research challenges in this domain.

2.1.1 Big Data Definition

Several definitions for Big Data have been proposed over time as efforts have been made to understand this domain [4][5][6][7]. Those definitions have been based on different concepts related to big data such as the volume of the data, the engineering needed to process the data or the value of the data, etc. The most popular among them which tries to combine almost all the concepts is the one from Gartner [8]. According to this definition, Big Data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation. Called the "3Vs" definition, this definition characterizes big data according to three aspects namely the Volume, the Velocity and the Variety of the data. Big Data is a large relative concept that is evolving. Today's big data may not be tomorrow's big data. A big data challenge for an organization may not be a big data challenge for another organization.

From all of these, we provide our definition of big data. Big data for an organization is a collection of large volumes of data, coming from a variety of sources, fast and in complex formats that cannot be processed using traditional computing techniques at the given organization.

2.1.2 Big Data Characteristics

Big Data is generally characterized by the Volume, the Velocity, the Value, the Veracity and the Variety of the data. These are commonly referred to as the Big Data five Vs (Figure 2.1).

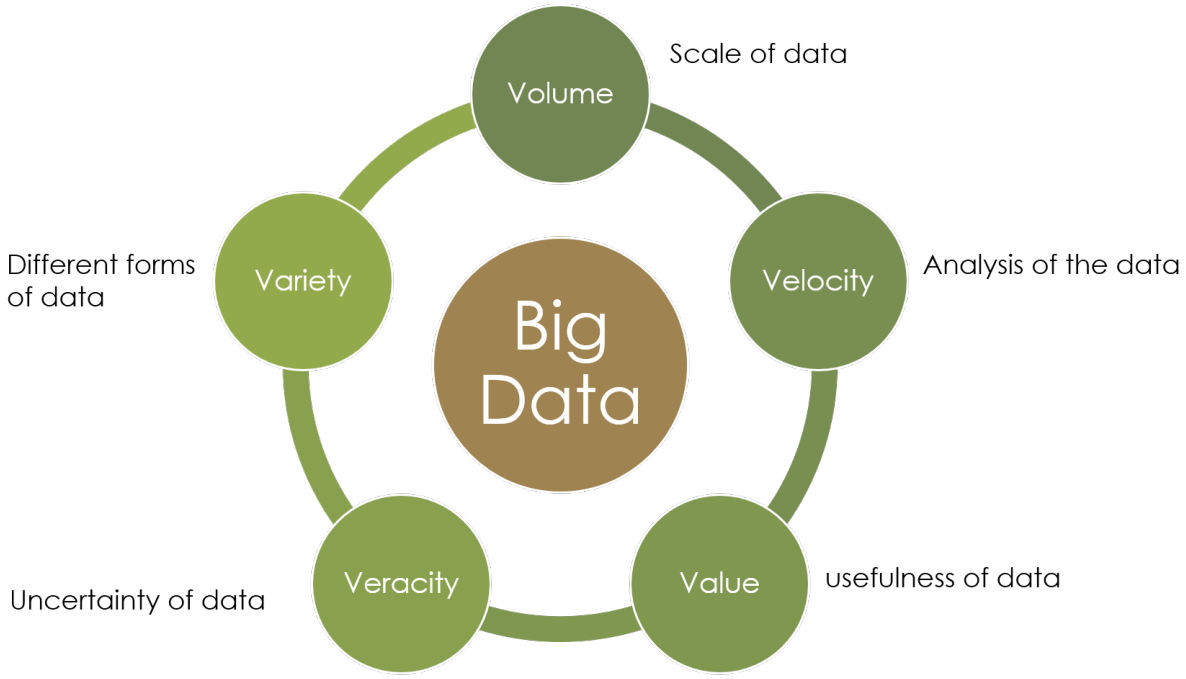

Figure 2.1: Big Data 5 Vs Characteristics

Volume

This aspect is related to the scale of the data. With current technologies, the hard disk size is growing faster than their read speed. The volume refers to the size of the data sets that need to be stored, read, analyzed and processed. Depending on an organization to another, the size of the data is frequently ranging from gigabytes to petabytes. Due to that size, the data require particular processing technologies than commonly used storage and processing capabilities. Namely, the data sets are too large to be processed using a simple computer. The 26 billion emails per day Yahoo has to deal with for its users is an example of a high-volume data set.

Velocity

This aspect is related to the analysis of the data. It refers to the speed and frequency at which the data to be processed is generated. According to this aspect, an analysis approach [9] (batch, real/near time, stream processing) will be chosen. The batch processing approach is no longer enough, and one might think of a computation where the data represent a potentially infinite stream of data processed as soon as elements appeared. Facebook status updates or Twitter messages are examples of data generated with high velocity.

Value

This aspect is related to the usefulness of the data. It refers to the worth of the data. When collecting data, unless the collected data are turned into value, it is useless. This value is very often the reason why the data are stored or not.

Veracity

This aspect is related to the uncertainty of the data. It refers to data reliability. Low veracity data, is data containing a high percentage of meaningless information also referred to as noise. High veracity data are data containing a high percentage of information that is trustworthy.

Variety

This aspect is related to the different forms of data. The data can be structured, unstructured or semi-structured.

Structured data generally refer to relational data. These data reside in a specified format. As an example, name, phone number, address, date amount, etc. These are the data contained in relational databases and spreadsheets.

Unstructured data refer to data that don't have a predefined data model. Data like PDF, Word, Text, audio, video, Media Logs, fall into this category.

Semi-structured data are data that have some organizational properties which ease their processing but do not reside in a relational database. Very often, some processing on the data can allow one to store them in a relational database. CSV data are an example of semi-structured data. NoSQL databases (often non-relational) are preferred for some data such as timestamped data.

2.1.3 Big Data Architecture

Many architectures have been considered for Big Data. The two dominant ones are Lambda Architecture and Kappa Architecture.

Lambda Architecture

Lambda architecture (Figure 2.2) has been proposed by Nathan Maz [3]. It has three layers: the batch layer, the speed layer, and the service layer. Incoming data go through both the batch layer and the speed layer. The data in the batch layer are immutable, they are never updated and new data are added to the end of the file. Results that meet the business requirements are precomputed using distributed processing systems. The speed layer deals with real-time data flow processing. In this layer, stream processing frameworks such as Spark Stream or Storm are used. In the service layer, the data from the batch views and those from the speed layer views are merged. These operations involve data locks at runtime and could involve latencies.

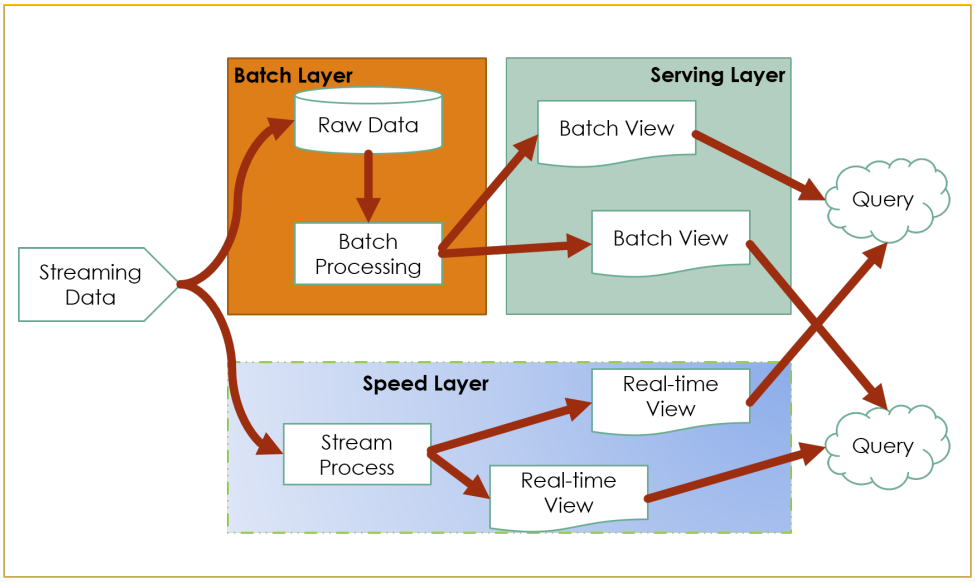

Figure 2.2: Lambda Architecture

Kappa Architecture

Kappa architecture (Figure 2.3) has been proposed by Jay Kreps [10]. In this architecture, everything is considered as a stream. It has two layers: a real-time layer and a serving layer. The real-time layer is used for data processing of the incoming data streamed through this layer. The serving layer receives the results from the real-time layer and is used for queries regarding these results. This architecture reduces the data locks, but the data storage is limited to the use of log files. In this context, a message broker like Apache Kafka can play the role of a distributed file system. Every queue has an address and is bound to a topic of exchange.

2.1.4 Technologies

Provided by different vendors such as Hortonworks, IBM, Amazon, Cloudera, Microsoft, etc., these technologies are important in providing infrastructures and means to manage and process huge volumes of structured, unstructured and semi-structured data. Big data technologies are divided into two categories: Operational Big Data and Analytical Big Data. Table 2.1 gives an overview of both classes.

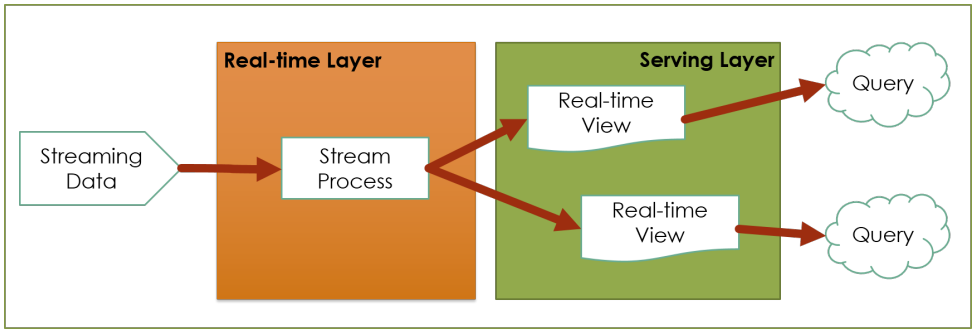

Figure 2.3: Kappa Architecture

Operational Big Data

Operational Big Data applications include MongoDB and Cassandra databases. The goal is to provide a faster and better way to deal with a large amount of data distributed over many sources. This is generally done using cloud computing architectures, allowing efficient and inexpensive massive computation. This makes operational big data workloads cheaper, faster to implement and much easier to manage. They also have capabilities for real-time analysis through the use of streaming frameworks such as Spark [11] or Flink [12].

Analytical Big Data

Analytical Big Data applications are systems that analyze data in a batch or interactive processing for retrospective and complex analysis. Analytical Big Data systems include Massively Parallel Processing database systems [13] and MapReduce (Section 2.3.3).

	Operational	**Analytical**
Latency	1 ms - 100 ms	1 min - 100 min
Concurrency	1000 - 100,000	1 - 10
Access Pattern	Writes and Reads	Reads
Queries	Selective	Unselective
Data Scope	Operational	Retrospective
End User	Customer	Data Scientist
Technology	NoSQL	MapReduce, MPP Database

Table 2.1: Operational Big Data Vs Analytical Big Data

2.1.5 Applications

Big Data is in use in every domain today. Most of the time, the use of Big Data is to enable processing for high volume or high-velocity data to increase productivity. In this section, we will give some application domain examples.

Banking

In this sector, Big Data is used for many purposes. It helps to improve employees' performance and management [14], improve cybersecurity and risk management, detect fraud and illegal activities [15][16]. It is also used for cost-saving and reduces the risk of failure by optimizing and

automating business processes [15]. In the stock market, it is used for short and medium-term predictive analytics [17].

Healthcare

The advent of many healthcare connected devices has increased the amount of data generated which organizations in this sector have to deal with. Big Data is used to predict the number of patients visiting at specific times to improved staffing [18]. It helps to improve medical screening, prognosis and diagnosis [19]. Big data applications are also used to predict locations where there is a chance of specific disease to spread [20].

E-commerce

In this domain, Big Data is used to increase the overall profitability [21], by helping to decrease product returns, offer customized promotions to customers, for example when a product is added to cart but was not bought by the customer, increasing the average revenue per customer. It is used to learn what customers really want and then deliver them accordingly [22].

Government

Governments need to deal with various complex issues on a daily basis at many levels. Big data has a very wide range of applications such as fraud detection, environmental protection, national security, scientific research, financial debts data management, taxes and tax categories management, cybersecurity, crime prediction and prevention, and financial market analysis [23].

Social Media

This sector can be seen as a cross-sector between all the others, as the data from this sector are also of benefit to the others. Marketing is one of the most impacted, where big data from social media can be used to better understand users, by analyzing their preferences, behaviors [24], in geo-marketing, challenge zones are recast based on visitors surfing in a commercial zone. Big data is also used to support healthcare in detecting disease spread [25].

2.1.6 Challenges

Big data came with many challenges and some of them are still prevalent. Those challenges are technical and non-technical. We will focus only on the technical ones in this part. The main challenges include storage, integration, processing, privacy, and security. In this section, we will discuss these challenges and the proposed solutions.

Storage

This is the most obvious challenge associated with big data. How to store the rapidly growing data in an efficient way that can ease the processing, while preserving the CAP theorem (Consistency, Availability, Partition Tolerance) introduced by Eric A. Brewer [26]. Traditional storage approaches consist of storing data in flat files as a record of various fields which are delimited by a space, comma, pipe, or any special character directly on the operating system's file system. Another approach is the use of traditional databases such as MySQL. These traditional storage approaches are difficult to apply in a big data context due to the increase in data generation. Advanced techniques such as cloud storage [27] (including Distributed File System) have been

proposed as a solution to deal with data in a Big Data context. The use of data replication along with data integrity preserved by data consistency in all the location, help users and applications to access the data consistently. New schema-free databases approaches such as Apache HBase [28], and Apache Cassandra [29] are emerging and becoming core technology for Big Data. They provide replication, consistency and support large scale data.

Integration

Big Data integration is an important phase in a big data project [30]. How to combine data coming from different sources, under different formats and being able to give a unified view of the data. ETL (Extract, Transform, and Load) [31] technologies such as Flume [32], are used to provide a way to move data from one or many sources and send them to another data environment in traditional data warehouses [33]. These technologies need to be improved to work within big data environments, due to the increasing variety and volume when dealing with big data [34]. To reduce the load time, new approaches have been considered [35]. ELT (Extract, Load, and Transform), opposed to traditional ETL approaches, is an approach for data integration process consisting of transferring raw data from the sources directly into the target and then perform the transformation there. Tools in this context, usually enable batch integration processes and real-time integration across several sources. Tools such as Talend [36], Sqoop [37] and Scribe [38] are the most used.

Processing

After the successful integration of the data and their storage, the next step is to get useful information from the data. How to efficiently process the generated large data sets. Data processing is a real challenge due to the volume of the data and their complexity (structured, unstructured, semi-structured, ...) [39]. Traditional approaches based on local processing lack the capacity to handle such data. Distributed computing such as MapReduce [40] is now widely used for processing large data sets. It consists of parallelizing the computation on a cluster of nodes. This aspect will be cover in detail in Section 2.3. Other applications such as Spark [11], Hive [41], Flink [12] are also used to support the data processing.

Security and privacy

Security and privacy are also big challenges to deal with in big data context [42]. The replication of the data and the use of distributed computation on many nodes create an environment difficult to secure but highly vulnerable to attack [43]. Data integrity and confidentiality implementation are also difficult and complex [43]. The value of a big data deployment can be interesting for attackers. Valuable information can be used, sold if an unauthorized user succeeds to gain access to a big data platform. Nowadays ransom demands related to information systems are increasing. Data might be lost in case of a successful ransomware attack, recovering from such an attack can be very challenging and costly. Traditional security tools are improved to cope with the scalable aspect of big data, with the ability to secure multiple types of data in different stages. Tools are for example encryption [44], Access Control [45], Intrusion Detection and Prevention [46], ... We also have Apache Ranger [47] which provides comprehensive security across the Apache Hadoop ecosystem and Apache Knox [48] which provides a secure way for interacting with Hadoop clusters.

2.2 Distributed File Systems

Storage is the first layer in a Big Data architecture (Figure 2.4). Many works [49] [50] have been conducted to be able to deal with the scalability of the data, and also the velocity at which data are being produced. The proposed approaches tend to improve how data are stored in such a way that ease big data processing. In this section, we define DFS, their characteristics and give examples of the most used ones.

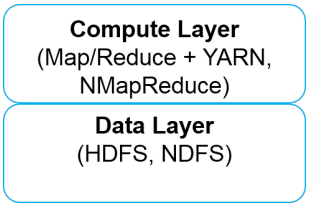

Figure 2.4: Big Data layers

2.2.1 DFS Definition

Before defining a DFS, it is good to know what a distributed system is. A distributed system is a collection of autonomous computing elements (hardware devices or software processes), collaborating and appears as a single coherent system (users or applications perceive a single system) [51]. From this definition, we define a distributed file system which is a type of distributed system [52]. A DFS is a file system residing on a collection of nodes but appears as a single integrated file system. It allows users to store and share data, but also working capability with the data as if the data were located on the user's computer.

2.2.2 DFS Characteristics

A distributed file system should have the following characteristics [52].

Transparency

A DFS has to provide different types of transparencies for the user such as access, location, relocation, migration, replication, concurrency failure. The DFS has to hide differences in data representation and the way data are accessed, the true location of the data. If the data are moved from a location to another one, that moving has to occur discretely. The fact that the data are replicated has to be hidden to the user, and also nodes failure and recovery.

Performance

The amount of time needed by a DFS to satisfy client requests such as data replication or data retrieval has to be good on average. This should also be the case when dealing with a large number of users. Also, the performance of a DFS should not depend on explicit file placement [53].

Scalability

A good Distributed File System should be able to support an increase in the number of nodes without causing any disruption of service or being noticed by the users. This characteristic

also includes the fact that the system should cope with a high service load and being able to accommodate the growth of the number of users or the resources. The scalability can be up and down.

High availability

A distributed file system should be resistant to failure. It should continue to operate in case of partial failures such as a storage device failure, a link failure, or even a node failure. Availability can be achieved through the use of file replication.

High reliability

A DFS should have a low probability of stored data loss. The System should provide a data recovery mechanism in the event of a loss. Having a good failover mechanism can help to achieve this goal.

Data integrity

A DFS should preserve the accuracy and consistency of the data when dealing with concurrent access requests from multiple users.

Security

The nature of a distributed system makes it much more prone to attack. Communication through the network can be intercepted. A DFS must provide data security to preserve data confidentiality and privacy. The resources must be protected from misuse and malicious use. Stored data protection mechanisms should be implemented.

2.2.3 HDFS

In this section, we present one of the most used DFS. This DFS is compared to our DFS in Chapter 6.2.

HDFS is a distributed, scalable, and portable file-system used in the Hadoop framework. It is used to store large files across multiple commodity hardware. It has a master/slave architecture (Figure 2.5). All servers are connected and the communication between them is made using TCP/IP based protocol. A master server, called NameNode is responsible for keeping and maintaining the file system namespace, storing file system metadata and coordinates the operations on the cluster such as finding where to store the replicated files and ensuring that the system performs correctly. The slave servers are called DataNodes. Their role is to store application data and provide access to the data.

HDFS supports data replication. In fact, when storing data on the DFS, the data are replicated on many DataNodes for reliability based on the replication factor, which is 3 by default. When there is a need for a client application to read or write data, the client sends a message to the NameNode which checks where the operation should be performed. The NameNode then sends the information about the location to the client which can directly read or write to the DataNodes specified by the NameNode.

With all the functionalities and the position of the NameNode within the cluster, it can be considered as a Single Point of Failure (SPOF) [54]. The loss of the NameNode would result in a loss of the entire cluster. To try to solve this issue, the Apache Software Foundation

has introduced an active/passive mechanism with the use of a second NameNode which is in a passive mode [55]. The active NameNode manages the clients' operations in the cluster, while the passive NameNode is in standby mode, acting as a slave, and having similar data as the active NameNode. It is used to provide failover in case of an issue with the active NameNode. Thus, when the active NameNode fails, the passive NameNode takes control and replaces the active node for the cluster to continue working. This approach brings some issues in terms of data consistency. In fact, there should always exist a perfect synchronization between the active and passive NameNodes to re-instantiate the cluster to the same state in case of a crash of the active NameNode [56]. Also, only one NameNode should be active at a time on the cluster. Having both NameNodes active will result in data corruption [57].

Furthermore, HDFS is inefficient in dealing with small data [58]. In fact, Big Data architectures may be required to handle a large collection of smaller datasets [59]. HDFS still faces challenges in this field of Big Data, for example, the lack of efficiency in the random reading of small files. The problem is that HDFS can't handle lots of files. Every file, directory, and block in HDFS is represented as an object in the NameNode's memory, each of which occupies 150 bytes. So 10 million files, each using a block, would use 3 Gigabytes of memory. Scaling up much beyond this size is a problem with current hardware.

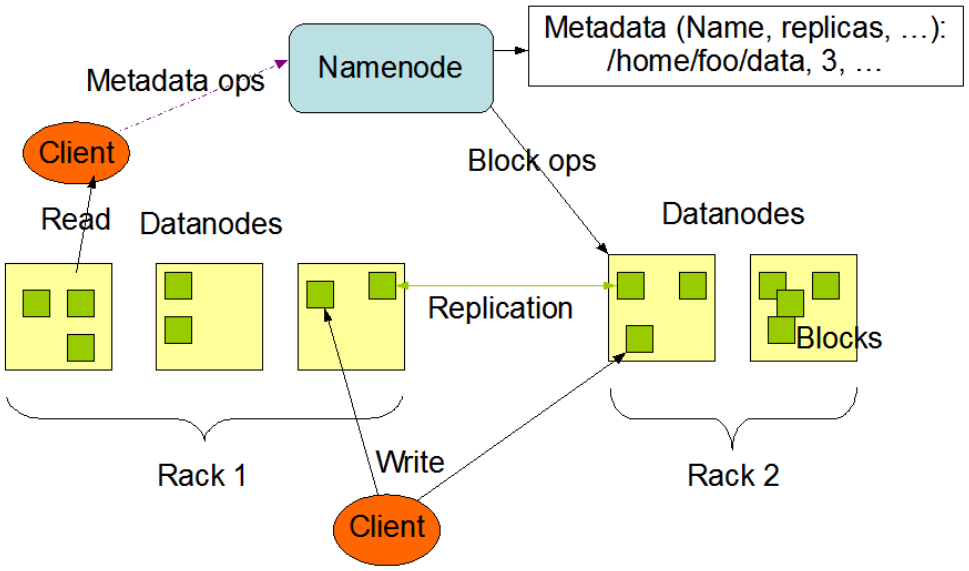

Figure 2.5: HDFS Architecture

2.3 Distributed Computing

After storing the data, one needs to gain insight from them. Distributed computing is the key to handle the deluge of data we have seen coming in recent years. It constitutes the second layer in a Big Data architecture (Figure 2.4). In this section, we define distributed computing, we give the main idea behind programming models and finally, we present MapReduce.

2.3.1 Distributed Computing Definition

Born in the late 1970s, distributed computing can be defined as an approach of using distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) to solve computational problems [60]. The problem is generally divided into many tasks, which are executed by one or more entities communicating with each other on a network [61]. This is done in such a way that each entity has only partial knowledge of the parameters involved in the problem that has to be solved. Data replication is also used to make sure that the failure of one node does not cause the failure of the entire computing. This approach is used when the working data set or problem to solve will take too long to execute on a single machine or fit on a single machine in memory.

2.3.2 Programming Models

A distributed application consists of a set of programs that are distributed over several nodes, which execute a script on data. In most cases, a parallel algorithm is obtained by splitting the data into pieces and defining a task for each part of the data. Each task is then assigned to a particular node. It is necessary to implement communications between the nodes executing the dependent tasks, except in the case where the processing of a task does not depend on the other tasks. Algorithms can, therefore, be broken down into calculation steps and communication steps. One of the main goals for distributing an algorithm is to reduce its execution time. During the execution, two requirements have to be constantly checked: minimize the communication and balance the workload between the available nodes.

The life-cycle of a distributed application can be described in 5 steps as follow:

- System initialization using user-supplied settings
- Data pre-processing
- Computation distribution on a maximum of available nodes
- Parallel processing at each node.
- Results collecting and post-processing for final result computation.

The master-slave paradigm is one of the simplest approaches for creating a distributed application. This consists of creating a "master" program that triggers other programs, called "slave", and waits for the end of their execution to retrieve their results. The process running the master program deals with the results one by one or aggregates them to start subroutines (slaves) or finalize the computation.

In the remainder of this section, we present some of the most popular and important programming models which can be used to design distributed computing applications.

Message passing

The concept of message is the main abstraction with this paradigm. In this model, several processes work on local data. Each process has its own variables and does not have direct access to the variables of other processes. To exchange data, processes explicitly encode the data in the form of a message and send them to each other. The content and structure of a message vary according to the model. Message Passing Interface (MPI) [62] is considered as the standard for parallel message exchange programming. Another example of this model is the OpenMP (Open Multi-Processing) [63].

Remote Procedure Call (RPC)

This paradigm extends the concept of procedure call at a multi-process level. It allows the execution of code in remote processes located on another entity on a shared network as if it were a local procedure call. This approach relies on a client/server architecture. A server component is hosted by a remote process that allows client processes to request the invocation of methods, and the server returns the result of the execution requested. Communications between server and clients are often handled by a middleware. An example is the Common Object Request Broker Architecture (CORBA) [64]. CORBA is a specification standardized by the Object Management Group (OMG). It provides automation of communication tasks, location, activation of objects, and the transmission of messages exchanged between systems irrespective of the platform and language used to develop. Another example is the ESB (Enterprise Service Bus) in SOA architecture [1].

Shared memory

The concept of shared memory is very often based on threads provided by operating systems. In this model, several tasks (threads) run in parallel, communicating with each other by reading and writing in physically or virtually shared memory. This programming model is the one implemented within Spark [11], where the context is distributed among the computing nodes.

2.3.3 MapReduce

The MapReduce programming model is derived from the map and reduce combiners of functional languages like Lisp [65]. In this type of languages, a map goes through a list of elements and independently applies an operation on each element. Reduce combines the elements of a list using a binary operator. To use the MapReduce model, the developer must define a Map function that processes input data and a Reduce function that processes the results of the Map function, to produce the final result. Figure 2.6 shows the data flow when applying the MapReduce model. First, the input data are divided into blocks and distributed over the compute nodes. Nodes are classified into two categories: those that perform the Map function (called Mapper) and those that perform the Reduce function (called Reducer). Mapper nodes apply the Map function to each data block. The result of this execution is a list of pairs that associate a key k with a value v (list (k, v)). These new data generated are called intermediate results. Each Reducer node retrieves all the values v that are associated with a key k and applies the Reduce function to all the values ((k, list (v)). Finally, the results computed by the Reducer nodes are assembled to give a final result.

The Open Source Hadoop project [55] provides an implementation of the MapReduce model. It defines two types of components: a jobtracker and several tasktrackers. They control the process of executing a MapReduce operation, which is called a job in Hadoop.

The jobtracker coordinates the execution of jobs across the cluster. It communicates with tasktrackers by assigning them execution tasks (Map or Reduce). It allows having a global vision on the progression or the state of the treatment distributed via an administration console. The jobtracker is a process that runs on the same node as the namenode. This is the case because the jobtracker should run on a master node and also because the namenode stores metadata about the data. So there is only one instance per cluster.

[1] https://www.opengroup.org/soa/source-book/soa/p1.htm

Tasktrackers perform the tasks (Map or Reduce) within a new JVM (Java Virtual Machine) that they instantiate for each task. A tasktracker is configured with a set of slots. A tasktracker can accept a task per slots such as a Map, a Reduce or a Shuffle. Thus, a crash of the virtual machine will not impact the tasktracker. In addition, tasktrackers periodically inform the jobtracker of the progress level of a task or errors so that it can reprogram and assign a new task. A tasktracker is a process that runs on the same node as a datanode. This is because the datanodes store the actual data, and the goal is to perform the computation very close to the data. So there are as many instances as there are datanodes. The communications between the different nodes (namenode / datanode, jobtracker / tasktracker) are carried out by RPC.

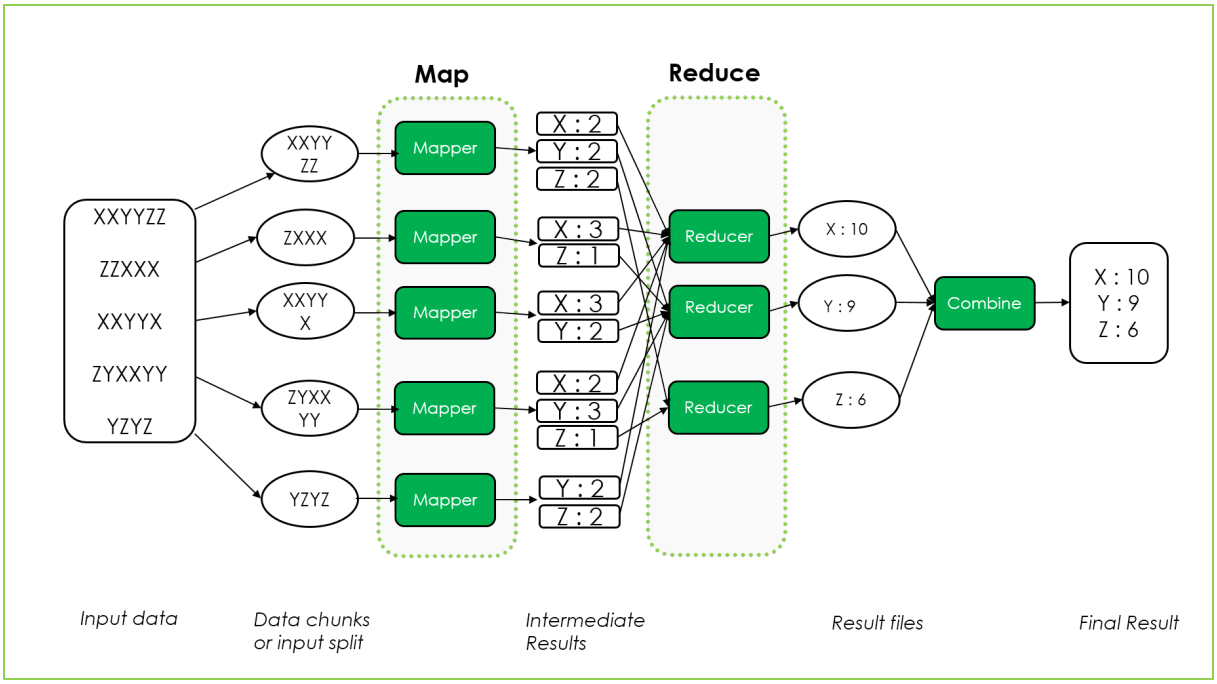

Figure 2.6: Word count using MapReduce model

2.4 Named Data Networking

Part of the Information-Centric Networking (ICN), the Named Data Networking architecture [66] is a fork of the CCN project originally proposed by Van Jacobson [67] in 2006. The CCN (Content-Centric Networking) project [68], initiated by the Palo Alto Research Center (PARC) has a proprietary architecture and has been acquired and managed now by CISCO. The NDN project is managed by UCLA with an open-source code. Many other architectures such as the Data-Oriented Network Architecture (DONA) [69], (Publish-Subscribe Internet Technologies) PURSUIT [70], and the Network of Information (NetInf) [71] have been proposed in the ICN context. In this section, we present NDN in more detail.

2.4.1 NDN Architecture

NDN architecture moves the current host-to-host communication model to a general-purpose network model supporting requests for named data. NDN has two types of applications: Producers

and Consumers. Producers advertise name prefixes for the content they can serve. Consumers send *Interest* packets to retrieve data by name. In this approach of network communication, we lose the notion of source and destination addresses.

Each NDN node has three main components (Figure 2.7):

- Forwarding Information Base (FIB): The FIB stores the different available routes to forward an *Interest* packet toward the requested content potential source (Producer). It is similar to IP routing tables with a difference in that it maintains data name prefixes instead of IP addresses. Instead of having a single best next-hop (next closest node a packet can go through) as in IP, FIB in NDN contains a ranked list of multiple interfaces and forwarding decisions for each *Interest* packet is based on a forwarding strategy module. The forwarding mechanism is discussed in more detail in Section 2.4.4.
- Pending Interest Table (PIT): The PIT allows a node to keep a list of *Interest* packets that have been forwarded and are waiting for a response. It also stores the interfaces on which they were received. One of the main objectives of the PIT is to avoid the forwarding of the same *Interest*. There is an entry for each pending content name so that if two *Interests* for the same content are received, it will be forwarded once. PIT entries are also used so that *Data* messages can use the reverse path of the *Interest* packets.
- Content Store (CS): the CS is a cache memory that allows the storage of *Data* packets received. It is in this table that all the answers that could potentially satisfy future *Interest* packets are stored. Each NDN node can answer directly an *Interest* request if the data are available in its Content Store. Its management can be done using different cache strategies. Many researches have been performed to manage the CS [72] [73] [74].

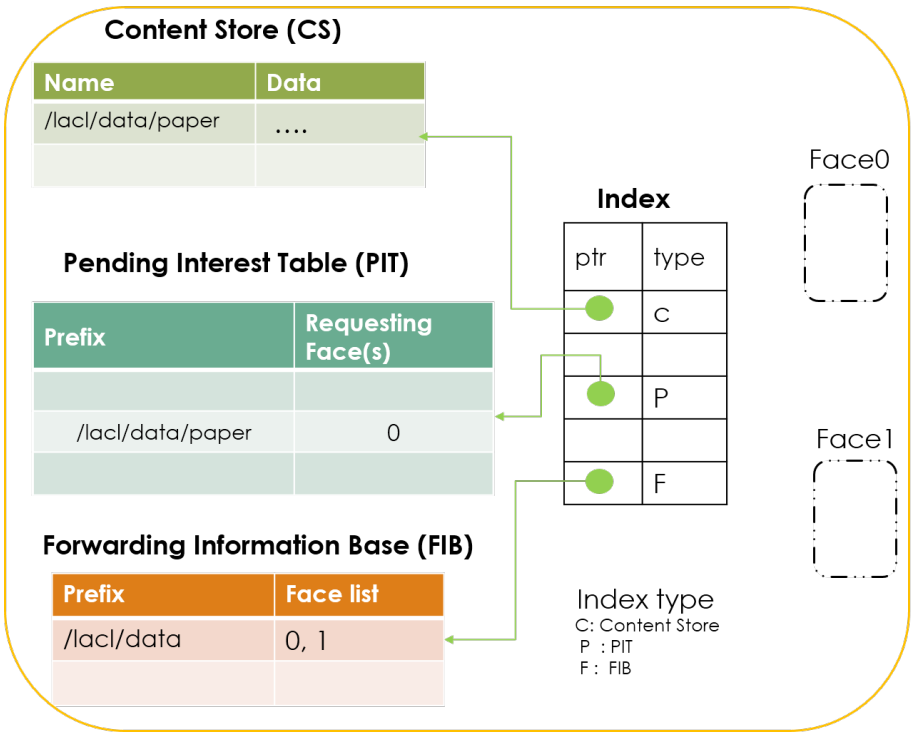

Figure 2.7: NDN node Structure

In the NDN architecture, for a user to get a content, he sends an *Interest* packet (Figure 2.8) on the network, using the node network interface (Face0 or Face1 on Figure 2.7) which will be answered with a *Data* packet (Figure 2.8) on the same interface. *Interest* packet contains the name of the data (representing the prefix of the data), as well as a nonce (randomly-generated 4-octet long byte-string) and several fields possible options such as the validity period of the request or the owner's public key who signed the message. *Data* packets are composed of the name of the data (the same prefix as the *Interest* packet), the content itself and the signature of the content. The network has an important task to resolve name queries in order to allow the routing of the data to the user. The NDN architecture enables caches, several copies of the same content may be present in different nodes of the network. Data can then be served from different nodes in the network.

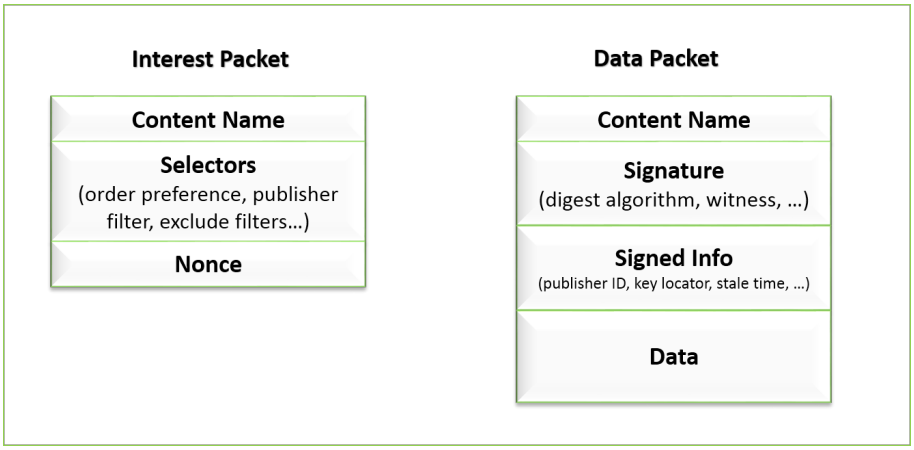

Figure 2.8: NDN Interest and Data packets

2.4.2 Naming

The basis of the exchanges in NDN is data. Each content is identified by a name or prefix which has a hierarchical structure like Unified Resource Identifiers (URI) [75]. This naming has the advantage of having a semantic meaning for users. For example, the prefix `/lacl/papers/2019/` stored in a forwarder may allow a node to properly forward all the *Interests* for the papers published the year 2019 by researchers from our laboratory. The hierarchical naming system offers the possibility for prefix aggregation. For example `/lacl/data/audio/call.mp3/part1/` and `/lacl/data/audio/call.mp3/part2/` identify two data that can be aggregated by the same prefix `/lacl/data/audio/call.mp3/`.

2.4.3 Security

NDN secures the content itself instead of the communication channel as in IP architecture. When created by an application, each piece of data is signed together with its name. With this approach, data integrity, pertinence and provenance can be verified by any node on the network using the data producer's public key, which allows users to trust data independently from where and how the data are obtained. This aspect is important because of the possibility NDN offers to retrieve already available data from cache with an intermediate node instead of forwarding the *Interest* to the producer. One question that can be asked is how to prevent access to certain data, as there is no information about the source and the destination in *Interest* or *Data* packet.

Content access control can be managed using encryption and the decryption keys distributed as encrypted data [76]. Thus only nodes allowed to access that data will have the decryption key and able to access the data.

2.4.4 Forwarding and Routing

The routing is performed by an application called NDN Forwarding Daemon (NFD). When a forwarder receives an *Interest* packet (Figure 2.9), it first checks its Content Store (CS) if the data are already available. If the data are found, they are sent back using the incoming interface of the *Interest* and the *Interest* is not forwarded because it is considered satisfied and dropped. Otherwise, the forwarder performs a PIT entries lookup. When an entry is found, that entry is interpreted and an action performed. If it's a duplicate *Interest*, it is dropped. If it's an *Interest* retransmitted by the consumer, it can be forwarded using a different outgoing interface. In the case of an *Interest* for the same data from another consumer, then the incoming interface of the *Interest* is added to the requesting list in the existing PIT entry, and the *Interest* is dropped. When the data are found, a copy is sent back using the *Interest* incoming interfaces. If the *Interest* name is not in the PIT, a new PIT entry for the incoming *Interest* is created by the forwarder, and it looks up for the *Interest* name in the FIB table. If no FIB entry matches the *Interest* name, the *Interest* is dropped and a NACK is sent on the incoming interface. Otherwise, the incoming *Interest* will be forwarded using the interface provided by the forwarding strategy module based on the active forwarding strategy. When a forwarder receives a Data packet, it performs a PIT entries lookup. If a matching entry is found, the forwarder sends the Data packet to all the interfaces from which the *Interest* was received and clears the entry from the PIT. The Data packets take the same paths of *Interests* in the reverse direction. If no match is found, the Data packet is discarded. Each *Interest* also has an associated lifetime set by the consumer; a PIT entry is removed if the *Interest* is not satisfied before its lifetime expires. According to its caching policy, a forwarder stores the received Data packet in its CS. Many routing protocols such as Hyperbolic routing [77] and the NDN Link State Routing (NLSR) [78] have been designed to help the forwarder.

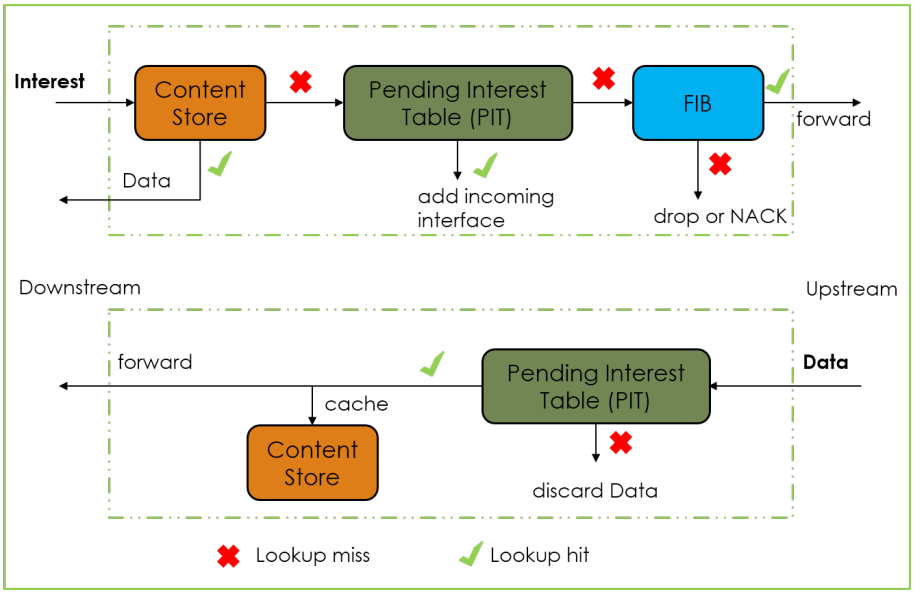

Figure 2.9: Forwarding

2.4.5 Caching

NDN enables automatic in-network caching. Nodes in the network can keep a copy of the content that has been forwarded through them. The contents are disseminated in the network and are closer to the users. Thus, the *Interest* will potentially encounter a closer node holding the content and not be propagated to the original producer, which will reduce the load on this producer and limit the impact of a search in the network. Delays in accessing content are reduced for the users and their quality of experience is improved.

2.4.6 NDN Testbed

An open platform of shared resources has been created for research purposes. The NDN testbed allows researchers to design, test and deploy new NDN-based solutions at large scale. This testbed includes software routers and has several participating institutions, application host nodes, and other devices. The Algorithmic, Complexity and Logic Laboratory (LACL) is part of the testbed (Figure 2.10). Participating in this testbed is a great opportunity for us to test applications on a real scale.

2.4.7 Big Data on NDN

Adopting NDN as the Future Internet architecture [68] [79], implies a rethought of applications in almost all domains [80] [81] [2]. Big Data topic is also concerned by this move. In [82], a big data platform running on NDN has been proposed by porting Hadoop on NDN. NDN is data-centric and Hadoop is designed for address-based IP architecture. Therefore, Hadoop native code must be modified to run on NDN. While it has been possible to run Hadoop on NDN, the single point of failure issue of Hadoop still holds.

In [83], a Named Data Storage System (NDSS) has been proposed for NDN architecture. NDSS adopts named data to describe network packets and local storage. Data blocks are stored on the local machines in the file system, and metadata of blocks are managed by a central node. There is no replication of the data and NDSS implies a redesign of NDN architecture. Finally, NDSS nodes can't be used for another purpose. This work shows the advantages of the use of NDN in a distributed context. Based on that, our approach is to avoid the use of a central component in the DFS architecture.

The project NDNFS aims to provide a file system over an NDN network and support efficient data access [84] by local and remote applications. This result is essential for many data usages; but in the case of big data applications, rules such as data availability through the use of replication and failover modes are added.

2.4.8 Simulation on NDN

Simulation has been for a long time a good way to run network experiments. In fact, they allow the execution of multiple scenarios for testing and validation purposes quickly, without the need to buy costly network equipment. ndnSIM [85] has been developed based on ns-3 [86] to provide a simulation environment for NDN experiments. This simulator meets the latest advances in NDN development.

In military communications [87], L. Zhang et al. studied the Army's Warfighter Information Network-Tactical (WIN-T) and the Navy's Automated Digital Networking System (ADNS) through emulations. They showed that NDN provides better efficiency in terms of average delivery delay, maximum delivery delay and bandwidth consumption than TCP-Based FTP.

Other network simulators have since been adapted to be able to run NDN experiments with them. This is to benefit from specificities proposed by those simulators. This is the case for example for the OPNET simulator [88], wherein [89], Hafnaoui TALEB et al. used it to evaluate their NDN proposed multi-layer architecture for energy-aware Wireless Sensor Networks integration on the cloud computing.

2.5 Summary

In this chapter, we have presented the state of the art related to the context of the research in which this thesis is situated. Big data is in use in every domain. A lot of challenges related to big data still exist today. For example, the SPOF in distributed file systems, also transparency in distributed computing needs to be improved. NDN enables in-network caching which leads to improving data transfer across the network and improves computation in big data. NDN appears to be a good candidate to improve big data storage and processing. In the next chapter, we propose a big data architecture based on NDN with a fully distributed approach to maintain the replication coherence without any need for a central component.

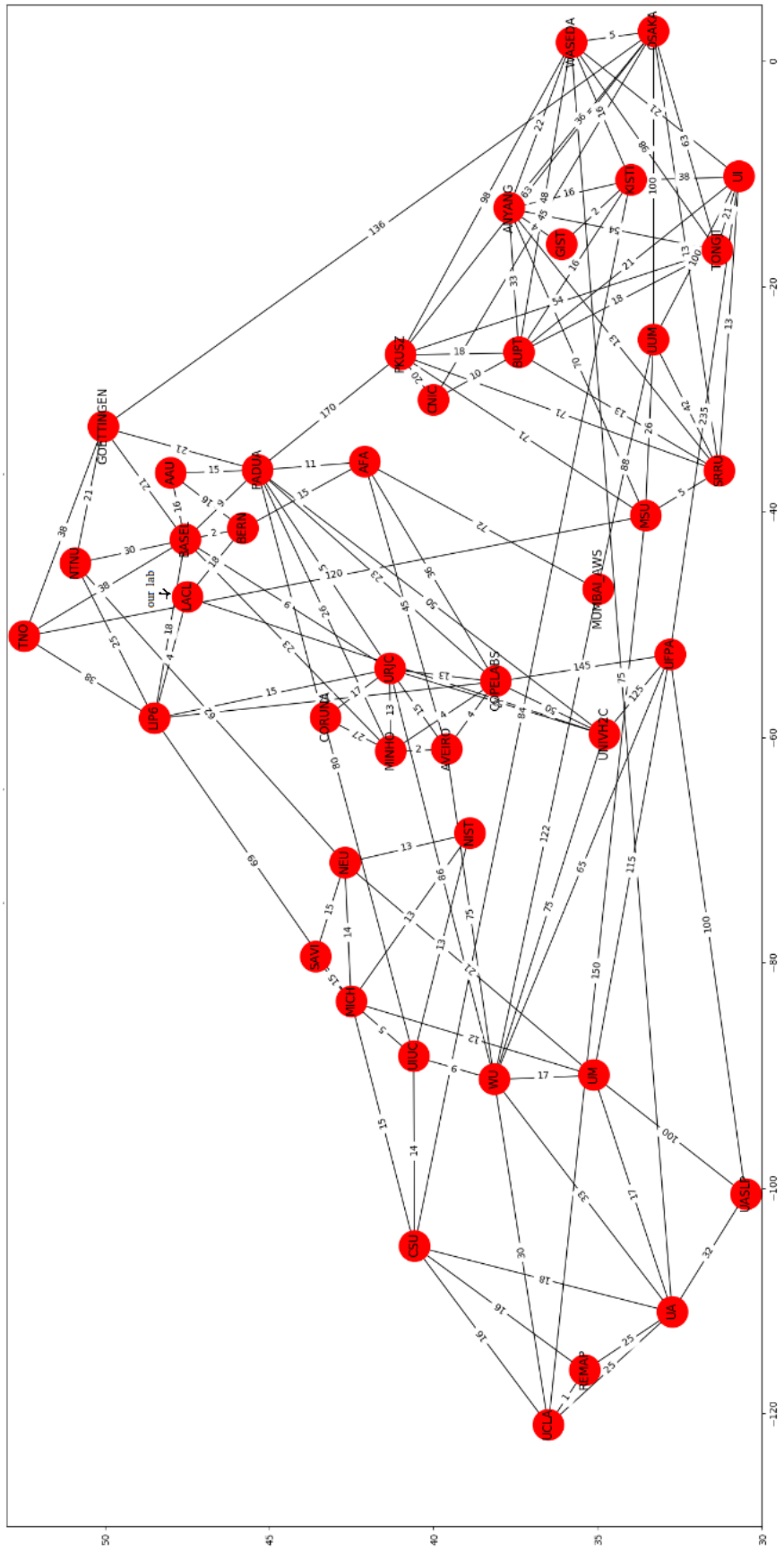

Figure 2.10: NDN Testbed (43 nodes, 121 links with NLSR costs)

Chapter 3

Architecture and Specification

Contents

In this chapter, we aim to define an architecture for Big Data application processing based on NDN [79]. Based on best practices and guidelines on big data applications development using the Hadoop framework [49], two layers are considered: a data layer where data are split into segments and a compute layer where the application reads the data and computes a new view. First, the data are imported from the local file system into a distributed file system. Next, computation over the stored data is launched over the network. When the computation is done, the results are saved over the NDN nodes and finally exported into the local file system if needed.

We define the architecture of these layers (Section 3.1 and 3.2), define formal languages (Section 3.3) and formally specify each component using Coq proof assistant (Section 3.4 and 3.5). We also formally verify each component by proving properties about them (Section 3.6).

3.1 Named Data Networking Distributed File System

In this section, we present NDFS (Named Data Networking Distributed File System), a fully distributed, resilient, and scalable DFS (Distributed File System). Our solution is not based on TCP/IP Internet architecture as we aim to deploy it on NDN. Our DFS protocol uses only NDN architecture defined messages (*Interest* and *Data*) (Section 2.4) to manage the communications between the nodes. We first present the different components of the architecture, the main operating principle, and the failover mechanism. We then through an example explain and illustrate the different prefixes of the *Interest* messages used by the protocol.

3.1.1 Architecture Overview

NDFS is a DFS based on NDN. It consists in a cluster of a set of nodes called Storage (Figure 3.1) whose function is to replicate data, manage the file system and ensure that the system performs in good conditions. They perform the functions of both a namenode and a datanode in HDFS [49]. A ReplicationClient node is a node running an application which is used to load data into the DFS (Distributed File System).

NDFS has a client/server architecture between the ReplicationClient and the Storages, and a peer to peer architecture between the Storage nodes. All the Storage nodes play the same role. This approach avoids the use of a central component for the DFS management, avoiding a Single Point of Failure issue (SPOF).

3.1.2 Data replication

The working context considered is where the data are replicated for better availability. The data block size and replication factor are configurable in big data context.

An application can only specify the number of replicas of a file (replication factor). This number should be chosen by the user when importing the file into the DFS unlike Hadoop where this feature is fixed by the big data platform administrator for all the files.

The placement of replicas is crucial for reliability and performance. The purpose of a node replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The caching availability of NDN (Section 2.4.5) gives the possibility to not bind the replica distribution to the node location as it's the case with HDFS where two replicas should not be distributed on the same rack [55]. The current implementation for the replica placement policy consists of a node set which advertises their storage capability. These nodes called Storage Nodes are the ones responsible for replicating the data and are those holding copies of the

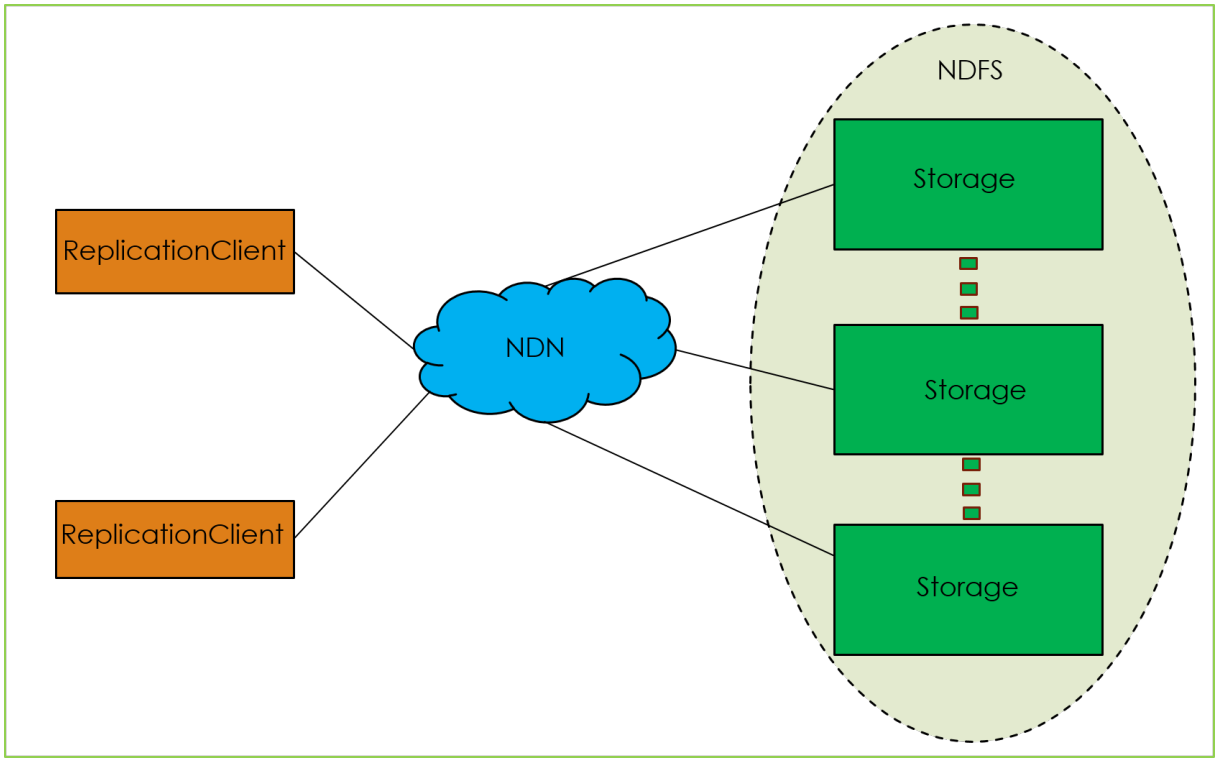

Figure 3.1: NDFS Architecture

data. The ReplicationClient advertises the name of the data it wants to replicate and sends an *Interest* to store this data on the DFS with a replication factor. Because of their capability, the Storage nodes will maintain full or partial copies of the data and advertise the data name over the network. By the end of this dissemination, the data will be replicated as much as requested by the ReplicationClient, which receives an *Interest* about the end of this deployment which in turn allows for the start of a first computation with the data.

3.1.3 Failure, Heartbeats and after failure replication

By maintaining data on many nodes, a potential failure on the nodes may occur. Therefore, the system needs to protect itself from data loss by using a failover strategy. A heartbeat algorithm already exists in frameworks such as ZooKeeper [90]. In a cluster architecture, its role is essential because it guarantees an error recovery of nodes. Thus, when the nodes host data, they advertise a recovery name which allows one to check whether a given node is alive. In case of a lack of answer, another node is looked for a new storage capability with an *Interest* for a new replication demand. Of course, such a test of liveness should be done before sending any storage *Interest*. This heartbeat mechanism is performed in a circular way with all the nodes participating in the replication. For example, in Figure 3.2, StorageA checks StorageB which checks StorageC, and finally, StorageC checks StorageA.

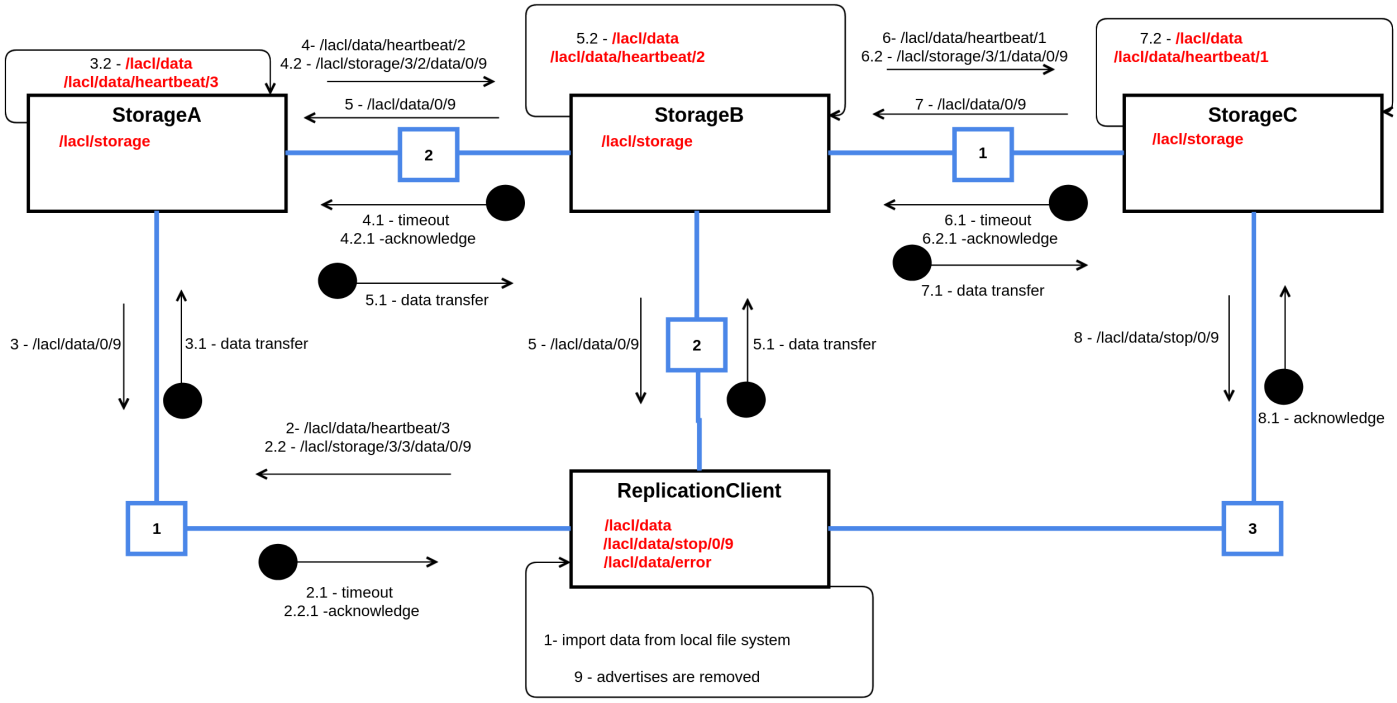

Figure 3.2: Data replication in NDFS

3.1.4 Protocol

For the design of our system, we chose to express a first view of our protocol with the UML collaboration diagram notation. The collaboration diagrams are used when we want to understand the message flow and the structural organization. Here, a message flow represents the sequence of control from one node to another and structural organization means the visual organization of the elements in a NDN network. It allows us to show explicitly the requests and the advertised names. The Figure 3.2 displays a simple topology with four nodes: ReplicationClient pilots the loading and the replication of the data on the NDN network. The links between the nodes play the role of the network configuration. The square over a line displays the weight of this link in a best route strategy[91]. The message routing strategy follows the minimum cost path. When an *Interest* is transmitted, then the request is not broadcasted on the network but a path is selected depending on the lowest cost. When multiple choices occur then the rule of longest prefix match is applied as used in NDN protocol.

An arrow pointing from the starting node to the destination node shows the interaction between these nodes. The number represents the order/sequence of this requests. Therefore, two arrows with the same number are sent in the same broadcast exchange. The resulting data is displayed as another arrow in the opposite direction with a bullet on the source side. Its label starts with a hierarchical number which shows the relationship between the request and the response.

The initial advertised names are displayed in the box of each node. After the importation of the data from its local file system, the ReplicationClient node sends an *Interest* over the

network with the name **/lacl/data/heartbeat/3**. This *Interest* allows the ReplicationClient node to check whether a node has already advertised this name. It would mean that the third replication of the data is already done. After a timeout event (arrow labeled 2.1), it knows such node does not exist. As a second consequence, the ReplicationClient node emits the following *Interest* **/lacl/storage/3/3/lacl/data/0/9** where:

- **/lacl** is the domain name,
- **/storage** is the precise demand. It expresses an *Interest* of storage capability for loading the data,
- **/3/3** is the replication factor, followed by the rank in the replication process. This rank is essential for the heartbeat strategy.
- **/lacl/data/0/9** is the name of the data followed by the index of the first segment and the index of the last segment.

The configuration of our example has enough nodes for three replications per data with a weight over the edges. In case of a lack of Storage node, the ReplicationClient node will receive an *Interest* with the name **/lacl/data/error/0/9**.

At the beginning of the scenario, three nodes have an advertised name called **/lacl/storage**. With the computation of the lowest cost path StorageA receives the *Interest* (arrow labeled 2.2) and then sends an acknowledgement (arrow labeled 2.2.1) to the ReplicationClient node. Therefore, the responding node called StorageA on Figure 3.2, sends an *Interest* about the data to be saved (**/lacl/data/0**) for the first segment and (/lacl/data/9) for the last segment. On Figure 3.2, the arrow labelled 3 gathers all the segment requests from the StorageA node. Only the ReplicationClient node has advertised the right name, which means it's the only node able to satisfy this *Interest* at this time. It returns the data and a first replication is done. Next the StorageA node advertises a new data name about the received data (arrow labeled 3.2) and another name for the failover algorithm (**/lacl/data/heartbeat/3**). This name is used to check that the node is always alive. If it fails, then a Storage node will send a storage *Interest* to find another node with a storage capability able to handle the data managed by the failed node.

In parallel, the StorageA node sends an *Interest* about the existence of a second replication of the data with the *Interest* **/lacl/data/heartbeat/2**. When it receives a timeout (arrow labelled 4.1), it creates another storage *Interest* by decrementing the replication factor which has the value 2 (arrow labeled 4.2). As previously, one node with the right capability answers (arrow labeled 4.2.1). The message process follows the same pattern as before and the data is replicated for the second time on the StorageB node (arrow labeled 5.1). New advertised names are registered. One for the new data access (**/lacl/data**), another name for the heartbeat algorithm (**/lacl/data/heartbeat/2**). Because the replication factor is greater than 0, the StorageB node sends an *Interest* to check the existence of the first replication (arrow labeled 6). After a timeout, a last replication request is sent by StorageB over the network (arrow labeled 6.2). When the StorageC node receives the data (arrow labeled 7.1), it advertises also new names for the data access and the heartbeat control and decrements the replication factor. Its value becomes 0 and this stops the replication process. StorageC node sends an *Interest* to end the process (arrow labeled 8) with the name **/lacl/data/stop/0/9**. Only the ReplicationClient node can filter this request. It then removes all the advertised names and ends its activity.

The heartbeat monitoring continues and every 10 seconds (this interval has been chosen based on NDN *Interest* default timeout), each Storage node sends an *Interest* to check a replication

data existence. As an example, the StorageA node sends the *Interest* **/lacl/data/heartbeat/2** (arrow labeled 4) to control that the StorageB node is always alive. If it receives a timeout, then it will trigger a new request with the same *Interest* as it did during the first deployment: **/lacl/storage/3/2/lacl/data/0/9**.

The heartbeat does not apply to ReplicationClient as it represents a file loading task initiated by the administrator of the system. If ReplicationClient fails, the administrator of the system must manage the fail over. The Storage nodes will keep trying to retrieve the missing segments. One solution to fix such issue is to make the segments available by running again the ReplicationClient in recovery mode. It will advertise the segments and wait for the stop command without sending any replication request. Algorithm 1 presents a pseudo code of the protocol.

Algorithm 1 Replicas distribution (Storage node)

```
Replication request Interest received
if node capacity is not OK then
    Forward the replication request Interest
    EXIT
end if
Send acknowledgment
Send Interest to retrieve Data packets
Advertise data name
Advertise heartbeat name
if replication is needed then
    Send next replication request
else
    Send stop Interest to end the replication process
end if
if errors then
    Send error Interest
end if
```

3.2 Computation distribution (NMapReduce)

In this section, we present the computation distribution NMapReduce. Our solution is based on NDN architecture and aims to provide a way for parallel and distributed computation based on the MapReduce [40] approach, through the use of NDN architecture *Interest* and *Data* messages (Section 2.4). We first present the different components of the architecture, the main operating principle and the underlying protocol.

3.2.1 Architecture Overview

Based on NDN, It consists in a cluster of nodes called Compute (Figure 3.3) whose function is to orchestrate data computation distribution, perform the computation in a parallel and distributed way for data replicated on a NDFS. Computation requests are sent to the cluster using a ComputationClient. The NMapReduce has a client/server architecture between the ComputationClient application and the Compute nodes, and a peer to peer architecture between the Compute nodes. Moreover, it is good to mention that a node can be both a Storage and Compute at the same time.

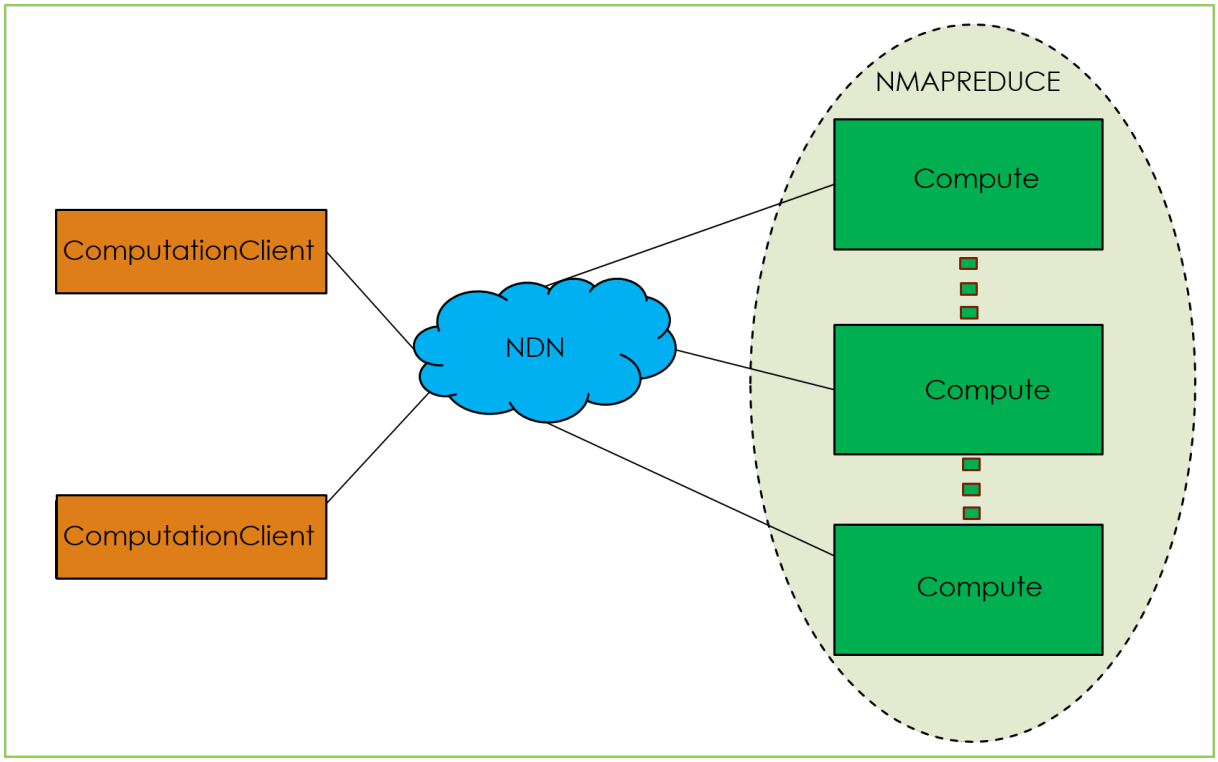

Figure 3.3: NMap Reduce Architecture

3.2.2 Principle

The computation distribution aspect has been introduced to take advantage of the replicated data. The idea is to distribute the computation and also avoid executing the same computation twice. Similar approach is used in the Hadoop framework where computations are performed on stored data using programming skeleton such as MapReduce [40]. The current implementation for the computation distribution policy consists of a set of nodes which advertise their computation capability. These nodes called Compute nodes are the ones responsible for performing the computation on the data. When a ComputationClient node sends an *Interest* for data computation, it is received by a Compute node which will perform a computation and using the same approach as in the replication, will find another node to compute another part if needed. Ultimately, the end result is consolidated by a node and sent back to the ComputationClient.

3.2.3 Protocol

The computation distribution is an incremental process. Similar to the traditional MapReduce, it is performed in two phases: a map phase and a reduce phase. In our approach, the script of the program to be computed is considered to be a data available on the cluster over the network. For example, such script can be available through the name **/domain/code/name**. Every node able to perform computation advertises a special name: **/domain/compute**.

Let's consider data available under the name **/lacl/data**. This data has 9 segments, labeled from 0 to 9. To perform computation on this data, replica of the data are available over the network on the Storage nodes.

The compute command used by the ComputationClient to request computation on the data is of the form **/domain/compute/domain/data/start/end/domain/code** where :

- **/domain/data** is the name of the data
- **/domain/compute** is a keyword to qualify the command as a compute command
- **/domain/code/name** is the script to apply on the data
- **/start** is the first segment
- **/end** is the last segment

For the map phase, when a compute command is emitted on the network, it is forwarded to a compute node based on the network configuration using the lowest path cost. This node is then responsible for performing map computation for the first segment (start) of the data. The node increases the start index, if the new start is not equal to the end index, a compute request is sent with this new value to find a node for the map computation of the next segment. An *Interest* will be sent to retrieve the script for the computation if it is not already present locally. The same apply to the segment that the node is expected to compute. This process is repeated until the last segment is processed.

By default, we have one node in charge of the reduce phase. The first node to perform the map is the one responsible for this phase. It advertises a special name **/domain/code/domain/data/heartbeat/reduce**, with:

- **/domain/code** is the name of the script to apply on the data;
- **/domain/data** is the name of the data
- **/heartbeat/reduce** is a keyword to qualify the heartbeat for the reduce.

This name is used to inform the other nodes about this task.

When a node completes a map, it advertises a name to inform that it holds a result for the map on a given segment. This name has the following structure: **/domain/code/domain/data/map/segment**, with :

- **/domain/code** is the name of the code to apply on the data;
- **/domain/data** is the name of the data
- **/map** is a keyword to qualify the command as a map result
- **/segment** is the identifier of the segment on which the map has been performed

During the map phase, we provide a heartbeat *Interest* to check if the map is still working and to find another node if not. This *Interest* can be used to have information about a job status. This *Interest* is of the form **/domain/code/domain/data/heartbeat/map/segment**. It also sends an *Interest* of the form **/domain/code/domain/data/heartbeat/reduce/map/segment** to the reducer to inform him about the availability of the result. These names mean:

- **/domain/code** is the name of the script to apply on the data;

- **/domain/data** is the name of the data
- **/heartbeat/map/segment** is a keyword to qualify the command as a map heartbeat check
- **/heartbeat/reduce/map/segment** is a keyword to qualify the command as a command to notify a reducer about a map availability.

The reduce phase starts in parallel with the maps. When receiving an *Interest* about the completion of a map by a node, the reducer sends a result retrieval using the name advertised by that node.

Algorithm 2 presents a pseudo code of the protocol.

Algorithm 2 Computation distribution (Compute node)

```
Computation request Interest received
Send acknowledgment
if first segment then
    Advertise reduce name
    Advertise name for the result
end if
if computation is needed then
    Send next computation request
end if
if node has not data then
    Send Interest to retrieve data packet
end if
if node has not code then
    Send Interest to retrieve code
end if
Advertise map name
Advertise heartbeat names
Perform the computation
Send Interest to reducer
```

Different commands are used to communicate with the system and to send orders to perform different actions. As an example from Figure 3.2, to initiate a replication request, the replicationClient had to issue the following command:
/lacl/storage/3/3/lacl/data/0/9.
The command needs to have a specific format. Semantically speaking, this command can be considered as a formal language [92] where semantic actions are computed.

3.3 Formal Language

In this section, we recall what a formal language is, and its different components. From these definitions, and regarding the semantic of the replication and computation commands, we identify two languages: the replication language and the computation distribution language (Section 3.5).

3.3.1 Symbol

A symbol is an abstract thing that represents something else [93]. In formal language theory, the most commonly used symbols are letters from alphabets, special characters and digits.

3.3.2 Alphabet

An alphabet is considered to be a finite set of symbols [92]. An alphabet is very often named. The most frequently used name for an alphabet is Σ. For example:

- an alphabet of two symbols, 0 and 1 is denoted $\mathcal{A} = \{0, 1\}$
- $\mathcal{Z} = \{x, y, z\}$ is an alphabet of three symbols, x, y and z.

3.3.3 Word or String

A word, also denoted string is defined as a finite sequence of symbols from an alphabet [92]. As an example, 0011 and 11111 are words from the previous alphabet $\mathcal{A}$. An example of strings from the alphabet $\mathcal{Z}$ are zzzxxyyy and xy. A particular string is the null which is a string with no symbol and is noted ϵ.

3.3.4 Formal language definition

A formal language $\mathcal{L}$ consists of a set of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules. There exist different types of formal languages [94]. The set of all possible strings over some alphabet Σ is defined Σ^*. The notation for a language defined by a grammar G is $\mathcal{L}$(G). The grammar G recognizes a certain set of strings, which all belongs to the language.

3.3.5 Grammar

A language is characterized by a grammar. It's a way to define which words belong to the language and those which do not.

3.3.6 Context Free Grammar (CFG)

Definition 3.1 (Context Free Grammar)**.** *A CFG is a set of recursive rules used to generate patterns of strings [95].*

It is a 4-tuple (V, N, P, S) where:

- *V is a set of terminal symbols; the characters of the alphabet that are present in the words generated by the grammar.*
- *N is a set of nonterminal symbols, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.*
- *P is a set of productions (each production has the form $V \rightarrow (V \cup N^*)$), which are rules for rewriting nonterminal symbols (on the left side of the production) in a word with other nonterminal or terminal symbols (on the right side of the production).*
- *S is a start symbol, which is a special nonterminal symbol that appears in the initial string generated by the grammar.*

The generation of a string of terminal symbols from a CFG, first, starts with a string consisting of the start symbol; then apply one of the productions with the start symbol on the left hand side, rewriting the start symbol with the right hand side of the production. Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols. A language generated by a context-free grammar is called a Context Free Language.

3.4 Coq Proof Assistant

In this section, we provide a short introduction to the Coq Proof Assistant, in which the replication and computation language parsers are specified, implemented and formally verified. This presentation is designed to be a quick introduction to understand the notation used in this chapter. For a complete introduction to Coq and more technical aspects, the reader can refer to the following materials [96][97][98][99].

3.4.1 Presentation

Coq (meaning Calculus Of Constructions) is a proof assistant developed in 1984 by Thierry Coquand and Gérard Huet. It has been since then improved and maintained by a INRIA team in France and also by its large users community. It allows the user to write formal specifications, functional programs and do proofs. It also offers through an extraction mechanism the possibility to produce ML (such as Ocaml) programs. Coq is based on a typed lambda-calculus with dependent types and inductive types. In fact, programs and proofs are formalized in the same language, the Calculus of Inductive constructions. The user can interact with Coq proof assistant using a basic shell command (coqtop), a graphical interface (coqide) or using emacs editor with access to the same functionality available in the graphical interface.

Coq has been used in many projects for different purposes. For example, Coq has been used to certify theorem provers [100] [101], as a framework for formalizing programming environments [102], to develop certified program such as the optimizing C compiler CompCert [103], and in mathematics for theorem proving [104] [105].

Many other proof assistants exist, such as Isabelle [106], HOL4 [107] and PVS [108]. They are all based on higher-Order Logic. Our choice for Coq is mainly based on the extraction feature it offers which enables a verified application, and also its large community of users. Coq is the proof assistant used in our laboratory [109] [110].

3.4.2 Coq programming language

The programming language of Coq is called Gallina [111]. It allows the development of mathematical theories built from axioms, hypotheses, parameters, lemmas, theorems and definitions of constants, functions, predicates and sets. It enables one to prove specifications of programs. The language of commands for Gallina is called the Vernacular (Figure 3.4). A sentence of the vernacular language begins with a capital letter and ends with a dot.

The language generated by this grammar permitted us to write the function in Figure 3.5 and the ASTs in Figures 3.8 and 3.9.

```
sentence            ::= declaration
                      | definition
                      | inductive
                      | fixpoint
                      | statement [proof]

declaration         ::= declaration_keyword assums .

declaration_keyword ::= Axiom | Conjecture
                      | Parameter | Parameters
                      | Variable | Variables
                      | Hypothesis | Hypotheses

assums              ::= ident ... ident : term
                      | binder ... binder

definition          ::= Definition ident_with_params := term .
                      | Let ident_with_params := term .

inductive           ::= Inductive ind_body with ... with ind_body .
                      | CoInductive ind_body with ... with ind_body .

ind_body            ::= ident [binderlet ... binderlet] : term :=
                          [[|] ident_with_params | ... | ident_with_params]

fixpoint            ::= Fixpoint fix_body with ... with fix_body .
                      | CoFixpoint cofix_body with ... with cofix_body .

statement           ::= statement_keyword ident [binderlet ... binderlet] : term .

statement_keyword   ::= Theorem | Lemma | Definition

proof               ::= Proof . ... Qed .
                      | Proof . ... Defined .
                      | Proof . ... Admitted .
```

Figure 3.4: Vernacular's partial syntax of sentences

Types

Every expression in Coq has a type. There exist many predefined types in Coq that can be used when programming. The use of these predefined types very often requires loading specific packages. For example, to use boolean type and functions on them, one needs to load the Bool package using the syntax: $RequireImportBool$. To use Natural numbers, $RequireImportArith$. Integers are enabled using $RequireImportZArith$. Coq also provides a powerful mechanism for defining new data types. This is done using the Inductive command. We used this mechanism when defining the inputs and outputs in Figures 3.8 and 3.9.

To verify if an expression is well-formed in Coq, one can use the $Check$ command. This command also gives the type of the expression in case it is well-formed.

Functions

The definition of a function is performed by the use of the $Definition$ command. The arguments and the return type of this function are explicitly declared. Coq can perform type inference, figuring out these types when they are not given explicitly. But it is a good practice to define

them, as they improve the readability of the code.

Coq offers the possibility to define intermediate results which are forgotten after returning the main result. This can be done using the syntax *let x := ...in...* . After having defined a function, it is possible to check that it works on some examples using the *Compute* command and also the *Eval* command.

Figure 3.5 gives an example of a function which takes a number, decreases its value and checks whether the result is 0 or not. This function is used in the replication process to check if another replication request is needed.

```
1 Definition replication_needed (n:nat) : bool :=
2   match  decrease_replication_factor n with
3   | 0 => false
4   | _ => true
5   end.
```

Figure 3.5: Function example

Proofs

The ability to write proofs of properties is an important aspect of the language. Coq provides a set of commands to deal with the proof development, and also uses specialized commands called tactics which allow the user to deal with logical reasoning. When writing a proof, there is a list of goals to prove at each stage. Initially, the list consists only in the theorem itself. After having applied some tactics, sub-goals are generated and included in the list of goals. A proof starts with the command *Proof* and is closed with a closing *Qed* (Quod Erat Demonstrandum).

3.5 Replication and computation language parser

In order to use a formal language, one needs to provide a parser which recognizes expressions of that language. In this section, we explain the role of a parser, and then present our approach for building the parsers for the replication and computation languages after having defining the grammar related to these languages.

3.5.1 Parser definition

A parser is a component of an interpreter or a compiler used to determine if an input data is part of a language defined by a grammar [112]. The parsing process is done at three stages:

- Lexical Analysis: A lexical analyzer is used to produce tokens from the stream of input string characters, which are then broken into small pieces to form meaningful expressions.
- Syntactic Analysis: Checks whether the tokens generated from the lexical analysis form a lexeme.
- Semantic Parsing: The final parsing stage in which the meaning and implications of the validated expression are determined and necessary actions are taken.

Two parsing approaches exist, which are:

- Top-Down Parsing [113]: this approach involves searching a parse tree to find the left most derivations of an input stream by using a top-down expansion. LL (Left-to-right, Leftmost derivation) parsers and recursive-descent parsers are some examples based on this technique.

- Bottom-Up Parsing [114]: in this technique, the input is rewritten back to the start symbol. This type of parsing is also known as shift-reduce parsing. LR (Left-to-right, Rightmost derivation) parser are built using this approach.

3.5.2 Approach

The idea is to build a verified parser (LR) which will then be integrated to the different components of our architecture (ReplicationClient, Storage, ComputationClient Compute).

For this purpose, we use Menhir [115] to generate the parser from the grammar and the Abstract Syntax Tree. We validate the generated parser using the approach from Jacques-Henri Jourdan, François Pottier and Xavier Leroy [116], consisting in a posteriori validation of an LR(1) automaton produced by a parser generator.

Menhir is a parser generator. It is able to generate a parser (LR) whose correctness can be formally verified using the Coq proof assistant [96]. This feature is used to construct the parser of the CompCert verified compiler [117].

Figure 3.6 gives an overview of our generation process. The first step is the specification of the grammar of our language. We use the grammar specification language provided by Menhir. Then, we define an AST (Abstract Syntax Tree) using Coq language. We generate the parser in Coq using Menhir and its coq backend and using the Coq proof assistant, we verify our parser by writing theorems and proving them. Finally, we extract an Ocaml version of the verified parser to be integrated into our components at the development phase.

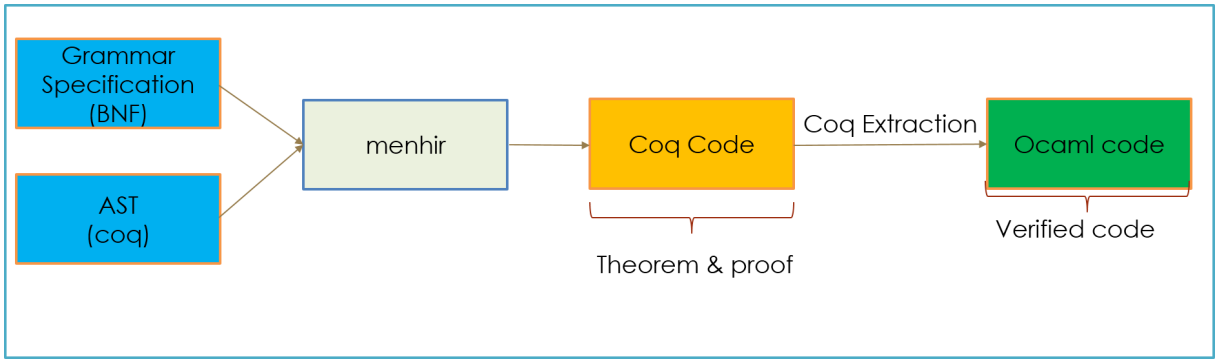

Figure 3.6: Parser generation process overview

3.5.3 Coq Specification

For our Coq specification, we use an abstract representation of a node as presented in Figure 3.7 A node is represented as a component which takes an input, parses this input, and based on the semantic actions of the language, produces an output. Semantic actions are associated to the parser. These are actions that are performed when a command is accepted. The model consists of specifications of the input, the parser, lemmas, theorems with their proofs.

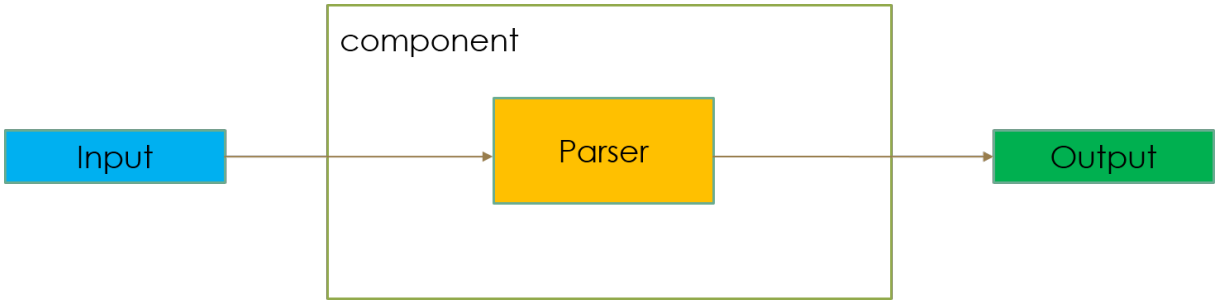

Figure 3.7: Coq model of a component

In the Figure 3.7, an example of input can be **/lacl/storage/2/2/lacl/data/0/9**, the output will be two *Interests* (**/lacl/storage/2/1/lacl/data/0/9** and **/lacl/data/0/9**), the first for the next replication and the second to retrieve the data to replicate.

Grammar specification

First we specify the replication language, and then the one for the computation distribution.

The replication language can be formally defined using the CFG :

- V_r: {rootDomain, replicationParameter, dataName, domain, data, replicationFactor, replicationIndex, firstSegment, lastSegment}
- Σ_r : { ['A-Z' 'a-z' 0-9]* , SLASH, STORAGE, EOL}
- S_r : main $\rightarrow$ SLASH rootDomain SLASH STORAGE replicationParameter dataName EOL

The computation language CFG can be defined as follow:

- V_c: {rootDomain, dataName, codeName, domain, data, firstSegment, lastSegment}
- Σ_c : { ['A-Z' 'a-z' 0-9]* , SLASH, COMPUTE, EOF}
- S_c : main $\rightarrow$ SLASH rootDomain SLASH COMPUTE dataName codeName EOL

Figures 3.16 and 3.17 present the whole specifications which are used as input for Menhir as described in Figure 3.6

Abstract Syntax Tree (AST)

The abstract syntax tree is a light version of the parse tree which only contains information related to analyzing the input, and skipping extra syntactic information that are used while parsing [114]. It is created as the final result of the syntax analysis phase. For Menhir to be able to generate the parser, one needs to provide the data structure of the AST that is expected. This AST data structure is constructed using elements from the grammar specification. It is used by Menhir to carry the results of the syntax analysis phase and to provide a way for further processing the obtained results. The AST data structure of the replication language (Figure 3.8) is obtained by defining 3 Inductive types. The first is the AST inductive type, which is

obtained by using three parameters, a string, a replicationParameter and a dataName. The replicationParameter is the second inductive type which is built using 2 nat parameters. Finally, the inductive type dataName is built using 2 string and 2 natural (nat) parameters. A similar approach is used to build the computation language AST data structure (Figure 3.9). ASTs are built using semantic actions associated with the grammar in Figure 3.16 and 3.17.

```
1  Require Import String.
2
3  Inductive replicationParameter : Type :=
4  |Parameters : nat ->  nat -> replicationParameter.
5
6  Inductive dataName : Type :=
7  |Name : string -> string -> nat -> nat -> dataName.
8
9  Inductive ast : Type :=
10 | Main : string -> replicationParameter -> dataName -> ast.
```

Figure 3.8: Replication Language AST data structure

```
1  Require Import String.
2
3  Inductive dataName : Type :=
4  |Name : string -> string -> nat -> nat -> dataName.
5
6  Inductive codeName : Type :=
7  |Name : string -> string -> codeName.
8
9  Inductive ast : Type :=
10 | Main : string -> dataName -> codeName -> ast.
```

Figure 3.9: Computation Language AST data structure

Extracting Programs from Coq

Another interesting feature of Coq is its extraction feature. Extraction transforms a Coq program into OCaml code, so that it can be used as a component of a larger OCaml program. We have used this feature to extract an OCaml version of our parsers (Figure 3.10) which will be integrated during the implementation phase. First we define the working directory. Next we import the necessary modules and the Coq version of the Parser, specify the extraction target language. Finally, we define the location where the extracted code should be saved.

Input and output

In order to have a complete representation of our model in Coq, we need to specify the input and output. They are specified using Coq types. They are defined as message and data which represent *Interests* and *Data* (Section 2.4). A message in the replication context can be of the type Heartbeat, Replication or Stop (Figure 3.11). When dealing with the computation language, a message can be a Computation, a Heartbeat, or a Code (Figure 3.12). These definitions are used in the next section when proving properties about the components. The whole specification of the parsers and details about them can be found in Appendix A .

```
1  Add LoadPath "/home/lacl/Documents/coq/ndfs".
2  Require Import Parser.
3  Require Import String.
4
5  From Coq Require Import extraction.ExtrOcamlString.
6  Require Import ExtrOcamlBasic.
7
8  Extraction Language OCaml.
9
10 Extraction "/home/lacl/Documents/coq/ndfs/parser.ml" main.
```

Figure 3.10: Parser extraction commands

```
1  Inductive message : Type :=
2  |Heartbeat
3  |Replication
4  |Stop
5  .
6
7  Inductive data : Type :=
8  |Data
9  .
```

Figure 3.11: Input and output specification for replication language

```
1  Inductive message : Type :=
2  |Heartbeat
3  |Computation
4  |Code
5  .
6
7  Inductive data : Type :=
8  |Data
9  .
```

Figure 3.12: Input and output specification for computation language

3.6 Theorems and proofs

In this section, we give all the theorems and properties related to our parsers and provide their proofs in Coq.

3.6.1 Correctness

The correctness property states that if a command (replication or computation) is accepted by a parser, then the word is valid (with respect to the grammar) and the semantic value that is constructed by that parser is valid as well. The Coq proof (Figure 3.13) of this property is provided during the parser generation by mehnir.

```
Theorem g_expr_correct iterator buffer:
  match g_expr iterator buffer with
  | Parser.Inter.Parsed_pr sem buffer_new =>
    exists word,
      buffer = Parser.Inter.app_str word buffer_new /\
      inhabited (Gram.parse_tree (NT g_expr'nt) word sem)
  | _ => True
  end.

Proof. apply Parser.parse_correct. Qed.
```

Figure 3.13: Parser correctness proof

3.6.2 Completeness

The completeness property states that if a command (replication or computation) is valid (with respect to the grammar), then it is accepted by the parser. The Coq proof (Figure 3.14) of this property is provided during the parser generation by mehnir.

```
Theorem g_expr_complete (iterator:nat) word buffer_end (output:       (string)):
  forall tree:Gram.parse_tree (NT g_expr'nt) word output,
  match g_expr iterator (Parser.Inter.app_str word buffer_end) with
  | Parser.Inter.Fail_pr => False
  | Parser.Inter.Parsed_pr output_res buffer_end_res =>
    output_res = output /\ buffer_end_res = buffer_end  /\
    le (Gram.pt_size tree) iterator
  | Parser.Inter.Timeout_pr => lt iterator (Gram.pt_size tree)
  end.
Proof. apply Parser.parse_complete with (init:=Aut.Init'0); exact complete. Qed.
```

Figure 3.14: Parser completeness proof

3.6.3 Consistency

An expected behavior of our parser is the sending of a NDN message when a command is accepted. This leads to formulate the following theorem:

If a word is accepted, then a message is sent and a data request is sent. We express the theorem in Coq and also provide the proof of this property (Figure 3.15). To this extend, we load the module Psatz [118] which gives access to several tactics for solving arithmetic goals.

3.7 Summary

In this chapter, we have defined the architecture of our Distributed File System and our approach to computation distribution based on NDN. We have presented the main operating principles of our protocols, their different initiation and transmission phases, and their algorithms. Our approach doesn't use a central component for the system management, but this task is distributed among the different nodes. We use *Interest* and *Data* messages for which we design a specific

```
Require Import Psatz.

Theorem ast_implies_message word buffer_end (output:ast)
        (tree:Gram.parse_tree (NT main'nt) word output):
  match main (Gram.pt_size tree) (Parser.Inter.app_str word buffer_end) with
  | Parser.Inter.Parsed_pr sem buffer_new =>
    (send_message sem = Replication \/ send_message sem = Stop)  /\ (send_data_request sem = Data)
  | _ => False
  end.
Proof.
  generalize (main_complete (pt_size tree) word buffer_end output tree).
  destruct (main (pt_size tree) (Parser.Inter.app_str word buffer_end)) as [ | |sem buffer_new]; auto.
  - lia.
  - intros _. split.
    + apply message_equal.
    + case sem; auto.
Qed.
```

Figure 3.15: Parser consistency proof

naming prefixes to allow nodes to communicate. Considering this specific naming as a formal language, we formally gave a grammar associated to the language, built a parser which accepts expressions from the language, and using Coq, we were able to prove that each node has a correct behavior. This formal verification is the first step in our approach for the architecture formal verification. In the next chapter, we will formally verify the whole architecture from the communication between components point of view.

```
(* expression example:  /lacl/storage/3/2/lacl/data/0/9 *)
%{
Add LoadPath "/home/lacl/Documents/coq/ndfs".
Require Import ast.
Require Import String.
%}
%token <nat> INT
%token <string> STRING
%token SLASH
%token EOL
%token STORAGE
%type<string> rootDomain
%type<replicationParameter> replicationParameter
%type<dataName> dataName
%type<string> domain data
%type<nat>    replicationFactor replicationIndex firstSegement lastSegment

%start <ast> main
%%
main:
 | SLASH r = rootDomain SLASH STORAGE rp =
   replicationParameter dn = dataName EOL
     { Main r rp dn };

rootDomain:
| s = STRING
    { s };

replicationParameter:
| SLASH factor = replicationFactor SLASH index = replicationIndex
    { Parameters factor index };

replicationFactor:
| r1 = INT
    { r1 };

replicationIndex:
| r2 = INT
    { r2 };

dataName:
| SLASH dom = domain SLASH dat = data SLASH seg1 = firstSegement SLASH seg2 =
    lastSegment
    { Name dom dat seg1 seg2 };

domain:
| d = STRING
    { d };

data:
| da = STRING
    { da };

firstSegement:
| seg1 = INT
    { seg1 };

lastSegment:
| seg2 = INT
    { seg2 };
```


Figure 3.16: Replication language grammar specification

```
(* expression example:  /lacl/compute/lacl/data/1/2/lacl/code
SLASH STRING SLASH COMPUTE SLASH STRING SLASH STRING SLASH INT SLASH INT SLASH
    STRING SLASH STRING EOL *)
%{
Add LoadPath "/home/lacl/Documents/coq_learning/nmap".
Require Import ast.
Require Import String.
%}
%token <nat>  INT
%token <string> STRING
%token SLASH
%token EOL
%token COMPUTE

%type<string>  rootDomain
%type<dataName> dataName
%type<codeName> codeName
%type<string> domain data
%type<nat>   firstSegement lastSegment

%start <ast> main
%%
main:
 | SLASH r = rootDomain SLASH COMPUTE dn = dataName
   cn = codeName EOL
     { Main r dn cn };

rootDomain:
| s = STRING
    { s };

dataName:
| SLASH dom = domain SLASH dat = data SLASH seg1 = firstSegement SLASH seg2 =
    lastSegment
    { Name dom dat seg1 seg2 };

codeName:
| SLASH dom = domain SLASH cod = code
    { Name dom cod };

domain:
| d = STRING
    { d };

data:
| da = STRING
    { da };

firstSegement:
| seg1 = INT
    { seg1 };

lastSegment:
| seg2 = INT
    { seg2 };
```

Figure 3.17: Computation language grammar specification

Chapter 4

Model Checking for System Verification

Contents

Software verification remains an essential element in the software development cycle. Unfortunately, many projects still don't include this phase in their early development. In fact, a verification phase can help to detect bugs early in the project. This is important as the cost to fix a bug is lower if the bug is detected early, but higher if detected at a later phase of the software development.

One approach used for system verification is formal methods. Formal method can be defined as the use of mathematical modeling approach for the specification, development and verification of a system. Our approach combines two verification techniques. In Chapter 3, we formally verified the components (Storage and Compute) taken individually using Coq.

In this chapter, we aim to verify properties related to the whole system (communication between components). Our system is obtained by components communicating (Figures 4.5 and 4.10) in order to perform the required actions. Real-time systems are systems including a timing aspect. The verification of these systems should include temporal properties. This notion is missing in Coq and it is not convenient for these types of verification. To this extend, we use

model checking techniques and the TCTL* (Timed Computational Tree Logic) temporal logic to express system properties that need to be verified, and verify them using UPPAAL model checking tool.

4.1 Real Time System Verification

System verification is the use of techniques to establish that a system under development complies to some properties. Many approaches can be considered when one needs to verify a system. Peer review is an approach consisting in the static inspection of the source code of the system by a team of software engineers that were not involved in the development process. This approach presents some limitations such as its handcrafted nature, and also the fact that it can not be performed at a early stage of the process. Another approach consists instead of analyzing source code without execution as in peer review, to run the developed application for testing purpose. While this approach has good importance, it also, can not be applied at early stage. Moreover, it only allows the detection of errors, but not their absence.

Model checking techniques is another approach which can be applied at a very early stage. The system is modeled as a set of automata. The properties to be verified are then expressed using a logic, basically a temporal logic.

4.1.1 Automaton System Specification

Model checking techniques [119] are generally performed by the use of finite state systems. Good candidates for applying these techniques are systems that can be easily represented as an automaton. This is the case for our systems (Named Distributed File System (NDFS) and NMapReduce) specified in Chapter 3.

Definition 4.1 (Automaton)**.** *An automaton $\mathcal{A}$ can be defined as a 5-tuple $<Q, E, T, q_0, \ell >$, where :*

- *Q is a finite set of states;*
- *E is the finite set of transition labels;*
- *q_0 is the initial state of the automaton ($q_0 \in Q$);*
- *$T \subseteq Q \times E \times Q$ is the set of transitions;*
- *ℓ is the application that binds any state of Q with the finite set of elementary properties verified in this state.*

As in programming languages, when modeling real systems, the model needs sometimes to keep record of some events such as tracking the number of times a transition has been fired, the number of errors, or the value of specific properties related to the system under modeling. To this extend, one can include variables which can be read, written, and subject to common arithmetic operations. These variables are part of the state of the system.

A current trend in software development is the use of modular programming. This approach consists in dividing a system into separate sub-systems (modules). It helps to circumvent the system complexity and group in the same unit of programming code similar functions that can be reused to compose different systems. The modeling of such system has to follow the same concept, which is an automaton modeling of each sub-system composing a global system and

then synchronize them to obtain the global system model. The synchronization can be performed using different approaches, but the obtained result (called synchronous product) may lead to a state explosion, which in this case makes the global system modeling difficult.

A state of the system is defined by the locations of all automata, and the values of the variables. For our system modeling, we will consider the particularity offered by the message synchronization approach, which is the use of a message sending/receiving mechanism for communication between all the automata composing the global system.

4.1.2 Time in Automaton

Real application processes are subject to different phases in their execution. The process can for example be in a waiting state (for a resource or a lost message to be resent). All these elements can influence the run time, which aspect has to be taken into account for the modeling. Two approaches exist for the time modeling: discrete time and continuous time. Some works [120] [121] consider a discrete time based modeling approach while we will consider a continuous time for this thesis.

Timed Transition System are used for expressing the timed model semantics. In this representation, the time domain ($\mathbb{T}$) is the natural number domain. The formal definition of a timed transition system is the following:

Definition 4.2 (Timed Transition System)**.** *A timed transition system $\mathcal{TTS}$ can be defined as a tuple (S, $\rightarrow$,Act,s_0) where :*

- *S is a finite set of control states;*
- *$\rightarrow \subseteq S \times (\mathbb{T} \cup Act) \times S$ is a transition relation;*
- *Act is the set of actions;*
- *s_0 is the initial state of the automaton ($s_0 \in Q$);*

This timed transition system allows two types of transition:

- those which correspond to instant actions and are represented with $\xrightarrow{\alpha}$, where $\alpha \in Act$
- transition $\xrightarrow{\nu}$, with $\nu \in \mathbb{T}$, expressing an amount of time flow which satisfies the following conditions:
  - zero delay: a $\xrightarrow{0}$ b if and only if b = a;
  - associativity: if a $\xrightarrow{\nu}$ b and b $\xrightarrow{\nu'}$ c, then a $\xrightarrow{\nu+\nu'}$ c;
  - temporal determinism: if a $\xrightarrow{\nu}$ b and a $\xrightarrow{\nu}$ c, then b = c
  - continuity: a $\xrightarrow{\nu}$ b, then for all ν' and ν" $\in \mathbb{T}$ such that $\nu = \nu' + \nu"$, there exists c such that a $\xrightarrow{\nu'}$ c $\xrightarrow{\nu''}$ b.

A timed transition system can have a finite or infinite sequence of transitions of S. An execution can be represented as follow:
$\sigma = s_0 \xrightarrow{\nu_0} s_0, \xrightarrow{\alpha_0} s_1 \xrightarrow{\nu_1} s_1, \xrightarrow{\alpha 1} \ldots\ s_n \xrightarrow{\nu_n} s_n, \ldots$

The works of Alur and Dill [122] [123] have extended normal automaton (definition 4.2) with a set of real-valued variables modeling clocks. The behavior of an automaton is restricted by the

use of constraints on the clock variables. This new type of automaton is called timed automaton and is formally defined as follow:

Definition 4.3 (Timed Automata)**.** *A timed automaton $\mathcal{TA}$ can be defined as a tuple (S,T,A, C,I,s_0) where :*

- *S is a finite set of location or control states;*
- *A is the set of actions;*
- *C is a finite set of clocks;*
- *$T \subseteq S \times \varphi(C) \times A \times 2^C \times S$ is a finite set of transitions ($\varphi(C)$ set of all clock constraints);*
- *$I : S \to \varphi(C)$ for each location, corresponding invariants;*
- *s_0 is the initial state of the automaton ($s_0 \in S$).*

A clock constraint is a conjunctive formula of atomic constraints of a form $\mathtt{x} \sim \mathtt{y}$, with $\mathtt{x} \in$ C, y a natural number and $\sim \in \{=, <, \leq, >, \geq\}$. It can be used as guard for a transition, telling at which moment the transition can be fired, or used as a location invariant constraining the amount of time that may be spent in that location.

To express the fact that a timed automaton can evolve from a location s_0 to a location s_1 when a clock constraint ϕ holds, and b an action associated to the transition, the representation $s_0 \xrightarrow{\phi,b,\varphi(C)} s_1$ is used. When firing the transition, all the clocks(φ(C)) are reset to zero and b is performed.

4.1.3 Temporal Logic

Temporal logic (first introduced by Arthur Prior [124] in 1960) is a way for reasoning with time-related propositions, using temporal quantifiers. It has then been extended for concurrent systems global properties specification by Amir Pnueli [125].

Computation Tree Logic

Computational Tree Logic(CTL) has been first proposed by Edmund M. Clarke and E. Allen Emerson [126] in 1981. It is used to express properties on the states of labeled transition systems.

Definition 4.4 (CTL Syntax)**.** *CTL formulas are generated by the following grammar:*

$$\phi, \varphi ::= p \mid \neg\, \phi \mid \phi \wedge \varphi \mid \phi \vee \varphi \mid \phi \Rightarrow \varphi \mid EX\phi \mid EF\phi \mid EG\phi \mid E\phi U\varphi \mid AX\phi \mid AF\phi \mid AG\phi \mid A\phi U\varphi$$

where:

- *p ranges over a set of atomic formulas;*
- *E means "for some path";*
- *X means next;*
- *F means sometime;*

- *A means "for all paths";*
- *G means always;*
- *U until;*

The following expressions are the basic CTL operators which are:

- EF ϕ: it is possible to get to a state where ϕ is true;
- AX ϕ: ϕ is Always true in the next state;
- EX ϕ: ϕ is Eventually true in the next state;
- AG ϕ : ϕ is Always true Globally;
- EG ϕ: there exists a path where ϕ is true Globally;
- AF ϕ: ϕ is Always true sometime (in the Future);
- AU ϕ: ϕ is Always true Until an event is true;
- EU ϕ: there exists a path where ϕ is true Until an event is true;

Timed Computation Tree Logic

With CTL temporal logic, it is not possible to reason about the duration related to events. Timed Computation Tree Logic (TCTL) [127] has been introduced as an extension of CTL for the specification of timed properties related to real-time systems.

Definition 4.5 (TCTL Syntax)**.** *TCTL formulas are generated as follow:*

$$\phi, \varphi ::= a \mid c \mid \neg\, \phi \mid \phi \wedge \varphi \mid E(\phi\ U^K\ \varphi) \mid A(\phi\ U^K\ \varphi)$$

where:

- *a is an atomic action;*
- *c is a clock constraint;*
- *E means "for some path";*
- *A means "for all paths";*
- *K is an interval whose bounds are natural numbers;*

TCTL allows the expression of the following type of properties:

- Reachability properties: A specific condition holds in some state of the system. They are expressed in the form : EF ϕ "Exists eventually ϕ" meaning there is an execution path in which ϕ eventually holds.
- Safety properties: A specific condition holds in all the states of an execution path. They are expressed using one of the two possible forms. Eϕ "Exists globally ϕ" meaning there is an execution path in which ϕ holds for all the states of the path, *or* Aϕ "Always globally ϕ" for each execution path ϕ holds for all the states of the path.

- Liveness properties: A specific condition is guaranteed to hold eventually. Two possible expressions: Aϕ "Always eventually ϕ"; For each execution path ϕ holds for at least one state of the path, *or* ϕ - -> φ "φ always leads to ϕ"; Any path that "starts" with a state in which φ holds, reaches later a state in which ϕ holds.
- Deadlock properties: A deadlock is a state system where it is impossible to move to a next state by any mean.

TCTL* is the temporal logic which is used by UPPAAL model checker.

4.2 UPPAAL model-checking tool

Jointly developed by the Swedish University Uppsala and the Danish Aalborg University, UPPAAL is a tool box used for real-time systems modeling, validation and verification. The modeling and validation are performed using graphical simulations via a Graphical User Interface(GUI) which runs on the user work stations. The verification is done using automatic model-checking by a model-checker engine which is by default executed on the same computer as the GUI, but can also run on a more powerful server.

UPPAAL uses a network of timed automata with properties expressed using temporal logics. During the modeling phase, model validation is performed to detect errors in the model.

The first release [128] of the tool was in 1995. Since then, it has a growing community, and many research activities such as [129] [130] [131] [132], and a lot of works for example [133] [134] [135] [136] are related to the tool. Also, UPPAAL Graphical User Interface is very user friendly. Due to all theses factors, UPPAAL is preferred instead of other model checking tools such as TAPAAL[137], Kronos [138], HYTECH [139]. It's the tool used in our research group and has been used for system properties verification in many thesis [140] [141] [142].

4.2.1 UPPAAL Automaton Formal Representation

UPPAAL is mainly based on timed automata. We give a formal definition of the timed automaton that it uses.

Definition 4.6 (Timed Automaton in UPPAAL)**.** *A Timed Automaton $\mathcal{T}$ can be defined as a 8-tuple <Q, V, C, F, Assign, Inv, K_L, q_0 >, where :*

- *Q is a finite set of locations;*
- *V is a set of Variables;*
- *C is a set of Clocks ($C \cap V = \emptyset$);*
- *$F \subseteq L \times \mathcal{G}(C, V) \times Sync \times Act \times L$ is the finite set of transitions where $\mathcal{G}(C, V)$ is a set of guard constraints, Sync is a finite set of actions (internal or synchronization), Act is a set of clocks setting or resetting;*
- *$Assign \subseteq Act$ is a set of assignment which set variables with initial value;*
- *$L \rightarrow Inv(C, V)$ is the application which for each location associates an invariant.*
- *$K_L : L \rightarrow \{n, u, c\}$ assigns a type (normal, urgent, committed) to each location.*

- q_0 *is a location of type initial* ($q_0 \in Q$);

A transition between two locations l_1 to l_2 is defined by the tuple $<l_1$, Select, Guard, *action*, A*ssign*, $l_2>$ where :

- Select: set of local variable related to the transition,
- Guard: set of guards,
- *action*: synchronization action,
- A*ssign* finite set of variable assignment or function call.

In UPPAAL, a system is modeled as a Network of Timed Automaton. From the previous definition, we give a formal definition of a Network of Timed Automaton in UPPAAL.

Definition 4.7 (Network of Timed Automaton in UPPAAL)**.** *A Network of Timed Automaton* $\mathcal{N}$ *can be defined as a 7-tuple* $< \overrightarrow{\mathcal{T}}$ *,* V_q*,* C_g*,* Ch*,* K_{Ch}*,* $Assign_g$*,* $\overrightarrow{q_0} >$*, where :*

- $\overrightarrow{\mathcal{T}} = (\mathcal{T}_1, \ldots, \mathcal{T}_n)$ *is a vector containing n timed automata* $\mathcal{T}_i = <Q_i, V_i, C_i, F_i, Assign_i, Inv_i, {K_i}^L, {q_i}^0 >$
- V_q *is a set of all the shared global variables by all the automata* $\mathcal{T}_i$ *(i ranges from 1 to n);*
- C_g *is a set of clocks shared by all the automata* $\mathcal{T}_i$ *(*$C_g \cap V_g = \emptyset$*);*
- Ch *is a set of channels used by the timed automata* $\mathcal{T}_i$ *to communicate (*$C_g \cap Ch = \emptyset$ *and* $V_g \cap Ch = \emptyset$*)*
- K_{Ch}*:* $Ch \rightarrow \{n, u\}$ *assigns a type (normal or urgent) to each channel.*
- $Assign_g$ *is a set of assignments which sets initial value for global variables;*
- $\overrightarrow{q_0} = ({q^0}_n, ..., {q_n}^0)$ *is a vector of the initial locations;*

4.2.2 Modeling and Validation with UPPAAL

An UPPAAL model is built as a set of concurrent processes. Each process is graphically designed as a timed-automaton. During the modeling phase, different steps have to be performed:

- **Global declarations**: the user has the possibility to define types, constants, variables and functions by placing them in the global section labeled Declarations. These elements will be accessible by all the timed automata in the system. An example of global declaration can be found in Figure 4.1. We define a constant *FACTOR* with value 100 of type integer, two boolean arrays *replica* and *waiting*, with 10 and 5 elements respectively, an integer variable *count* with the range [0, 10] initialized to 6, a clock *time*, a synchronization channel *shut_down*, a new type definition whose variables range is [1, 100]. Finally, we define a function which returns the sum of two integers.

```
1  const int FACTOR = 100; // constant definition
2
3  bool replica[10], waiting[5];  // definition of two boolean arrays
4
5  int[0,10] count=6;  // an integer variable definition
6
7  clock timer;  // a clock definition
8
9  chan shut_down; // a synchronization channel definition.
10
11 typedef int[1,100] id_client; //a new type definition
12
13  /* declaration of a function which returns the sum of two integers */
14 int add(int a, int b)
15 {
16      return a + b;
17 }
```

Figure 4.1: Global scope declaration in UPPAAL

- **Models definition**: A timed-automaton is used to define a model also called template. It is represented as a graph which has locations as nodes and transitions as oriented edges between locations. The user has the possibility to define types, constants, variables and functions related to the model, by placing them in the local section labeled Declarations (Figure 4.4).

  Edges can be annotated with guards, updates, synchronizations and selections.

  - *A guard* is an expression which uses the model elements such as variables or clocks in order to define when the transition is enabled.
  - *An update* is an expression whose evaluation changes the state of the system. This evaluation is performed as soon as the corresponding edge is fired.
  - *The synchronization* is the basic mechanism used to coordinate the action of two or more processes, by allowing them to take a transition at the same time. This is done by the use of a channel. For example, declaring a channel c, then one process will have an edge annotated with $c!$ and the other process another edge annotated with $c?$. There are three different kinds of synchronizations: Regular channel, Urgent channel and Broadcast channel.

  Locations can be of type Initial, Urgent, Committed or Normal and can have an optional name and invariants.

  - *Initial location*: identified with a double circle, it's the location from which the process starts.
  - *Urgent location*: identified with a "U", location in which time is not allowed to pass.
  - *Committed location*: identified with a "C", time is not allowed to pass when a process is in a location of this type and no transition other than those leaving a committed location can be enabled.
  - *Location name*: identifier used to refer to the location during model checking and when writing documentation for the model.

– *Invariant*: conditions related to variables and clocks that must be fulfilled while the automaton is in that location.

Figure 4.2 gives an example of a model using UPPAAL graphical user interface.

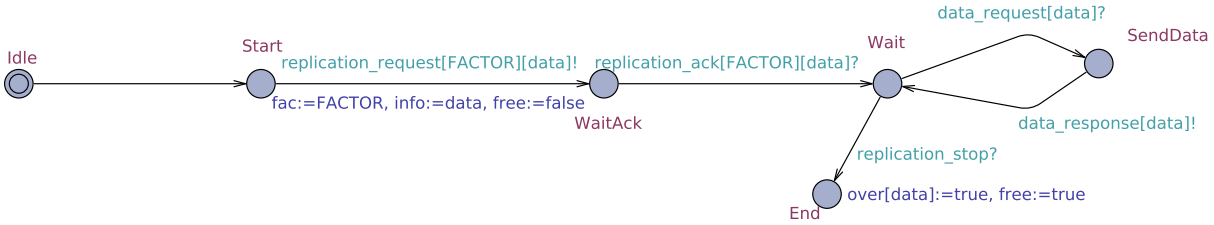

Figure 4.2: UPPAAL timed automaton model GUI representation

- **System declarations**: This step consists in the instantiation of one or more concurrent processes which compose the system, each process having been previously modeled as a template (timed-automaton). As in the Global declaration, channels, variables and functions can be defined in the system declaration with a global scope. They are used when giving arguments to the formal parameters of templates. Having been declared after the templates, they are not directly accessible by any of them. An example of a system declaration containing two concurrent processes is presented in Figure 4.4. One process defined by the model ReplicationClient presented in Figure 4.2 and the second modeled by the timed automaton User defined in Figure 4.3

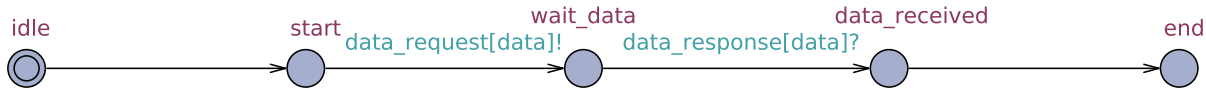

Figure 4.3: UPPAAL timed automaton User model

UPPAAL model validation is performed through the use of a simulator. The simulator permits the examination of the possible dynamic executions of a system after the modeling, providing an inexpensive way of fault detection prior to verification by the model-checker. The simulator is also used to display executions generated by the verification engine.

4.2.3 Verification using UPPAAL

After using the simulator to ensure that our model behaves as the system we wanted to model, the next step is to check that the model verifies our properties. The first step is to decide the properties to verify, formalize them and then translate them into UPPAAL query language which is a subset of TCTL (Definition 4.5). This language gives way to the expression of very simple properties directly. This design approach has been chosen by the UPPAAL designers instead of allowing complex queries, with the objective of improving the efficiency of the tool. One can still verify complex properties by checking many different simple queries.

UPPAAL allows for the verification of Reachability properties, Safety properties, Liveness properties and Deadlock properties. Considering φ and ψ some atomic propositions, a property that can be verified by UPPAAL is of the form:

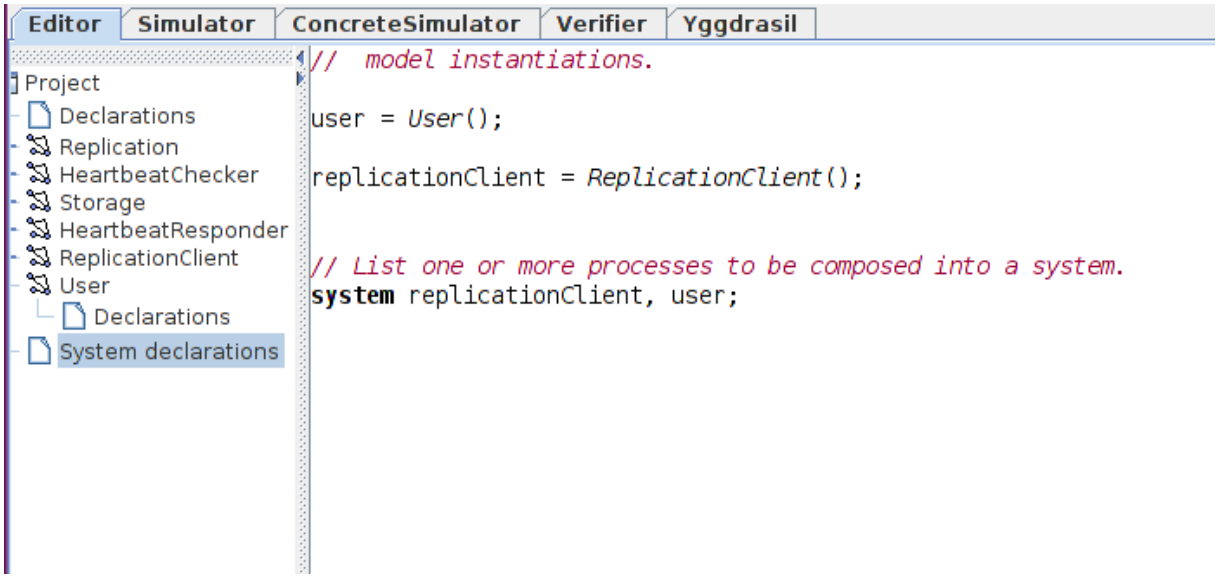

Figure 4.4: System declaration UPPAAL

- E <> φ (Exists eventually φ): there is an execution path in which φ eventually (in some state of the path) holds;
- E[] φ (Exists globally φ): there is an execution path in which φ holds for all the states of the path;
- A[] φ (Always globally φ): for each (all) execution path, φ holds for all the states of the path;
- A<> φ (Always eventually φ): for each (all) execution path, φ holds for at least one state of the path;
- ψ -> φ (ψ always leads to φ) : any path, that "starts" with a state in which ψ holds, reaches later a state in which φ holds.

An atomic proposition is one or a combination of the form:

- $\mathcal{T}.\ell$, which states that the timed automaton $\mathcal{T}$ is in the state ℓ (if $\ell \in Q$), or references the value of the local variable ℓ of the timed automaton $\mathcal{T}$ (if $\ell \in V$).
- variable or clocks values compared with a variable, clock or an integer. The comparison operators used are : $=$, $<$, $>$, $\leq$ et $\geq$.

Finally, deadlock-freeness is checked using the property A[] not deadlock.
In Section 4.4 we use this language to express properties related to our system.

4.3 System Modeling

In this section, we design each component that has been specified and verified using Coq (Section 3.5.3) as a timed automaton (model). We then use the automata to compose the whole system. The next step will be the verification of properties related to the whole system.

To reduce space state explosion, we used some tricks during the modeling. We used wherever applicable committed locations which reduce significantly the state space. The number of variables plays an important role for the verification and simulation time. For each variable, we provide a precise range for the integers. We also reset the shared variables when they are no longer needed.

4.3.1 NDFS

The NDFS is composed by a set of Storage Nodes (Section 3.1). It is the first layer of our Big data framework. In this section, we describe the timed automata templates used to define the Distributed File System (DFS). The global system is composed by four automata describing a Storage node (Figure 4.5). Two other automata modeled the ReplicationClient application to request data replication on the DFS, and also a user application to retrieve, or delete data.

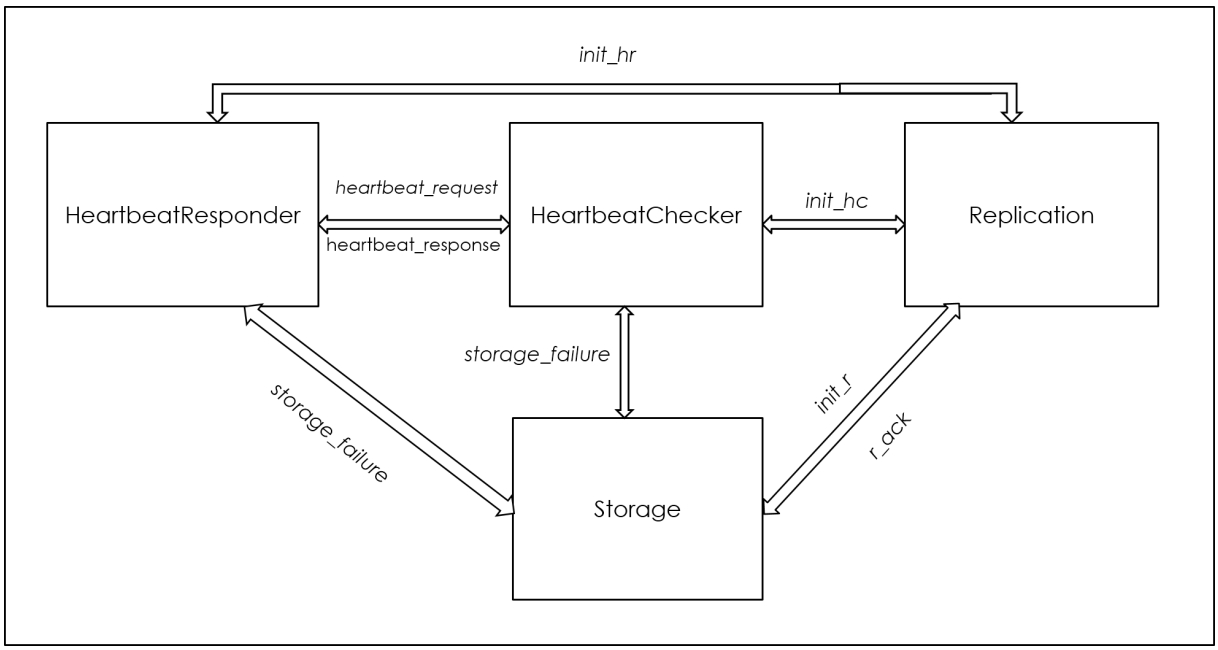

Figure 4.5: Storage node automata network

Storage Model

This model (Figure 4.6) represents a global view of the storage node. It has seven states:

- Idle: initial state, or inactivity state;
- Start: state in which the storage is able to process requests. It can be requested from a ReplicationClient (for replication) or the user (for data retrieval).
- Ready: this state is reached after a synchronization on the channel $replication\_request[inIndex][inData]?$ with the ReplicationClient (Figure 4.2). This state corresponds to a state in which the storage has received a replication request from the ReplicationClient. All the necessary parameters are passed through the channel, with *inIndex* for the replication index and *inData* used to identify the data to replicate.

- Replicating: this state is reached after the storage initiates a Replication model (Figure 4.7) to handle the request. The initiation is done through the use of a channel $init\_r[id]$! (id is the storage identifier). This state indicates that the node is processing the request.
- ReplicationComplete: this state is reached after the synchronization on the channel $r\_ack[id]$? (id is the storage identifier) by the Replication model, which uses that channel to notify the completion of the replication process. The automaton can evolve to the Start state waiting for another replication request, or request from a User (Figure 4.3). The node is able to respond to data request.
- SendData: this state is reached after a synchronization on $data\_request[in]$?, which corresponds to the reception of an *Interest* for a data (identified by the incoming identifier in) replicated by the node. This request can be initiated by a User, or another Storage node through it's Replication template when replicating a data. The data are then sent to the requester using a synchronization on the channel $data\_response[outDataChan]$!.
- Failure: this state is used to simulate in a non deterministic way the failure of a node.

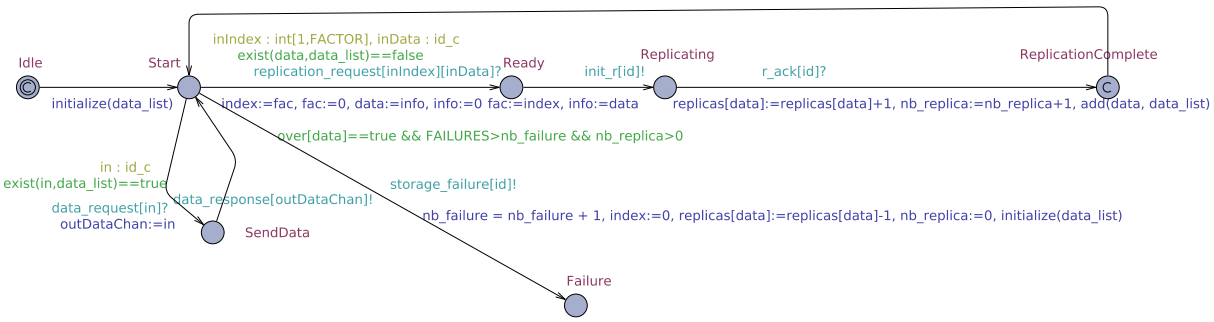

Figure 4.6: Storage model

Replication Model

This model (Figure 4.7) represents the process in charge of performing the replication for a storage node. It has fourteen states. This large number of states can be explained by the fact that UPPAAL doesn't allow the use of multiple synchronization on a same channel. To be able to achieve that, we use a sequence of committed states. Then they represent an atomic action. This allows to execute all of them without time delay, when the first is enabled.

- Idle: initial state;
- Ready: this state is reached after the initialization by the Storage process using $init\_r[id]$? channel (Figure 4.6).
- Ack: this state is reached after the Replication automaton has responded with its ability to deal with the replication. This acknowledgement is sent directly to the ReplicationClient (Figure 4.2) using
  $replication\_ack$[FACTOR][$data$]! (with $FACTOR$ representing the replication factor, and $data$, identifies the data to replicate).

- Waiting: this state is reached after sending a request for the data to replicate;
- Replicating: when the data are received, the replication is performed;
- CompleteInitI: this state is reached when there is a need to send a replication request to another node and after the initialization of the heartbeat process (Figures 4.8 and 4.9) in the states InitResponderI and InitCheckerI (Figure 4.7);
- CheckNextReplica: in this state, an heartbeat *Interest* is sent to check if the replica related to the next replication index doesn't already exist.
- Replication: this state is reached when there is a need to send a replication request to another storage;
- CompleteF: this state is reached after the initialization of the heartbeat process in the states InitResponderF and InitCheckerF (Figure 4.7) and when there is no need to send a replication request to another node. From this state, a message is sent to the ReplicationClient using the channel *replication_stop*! to complete the replication process.

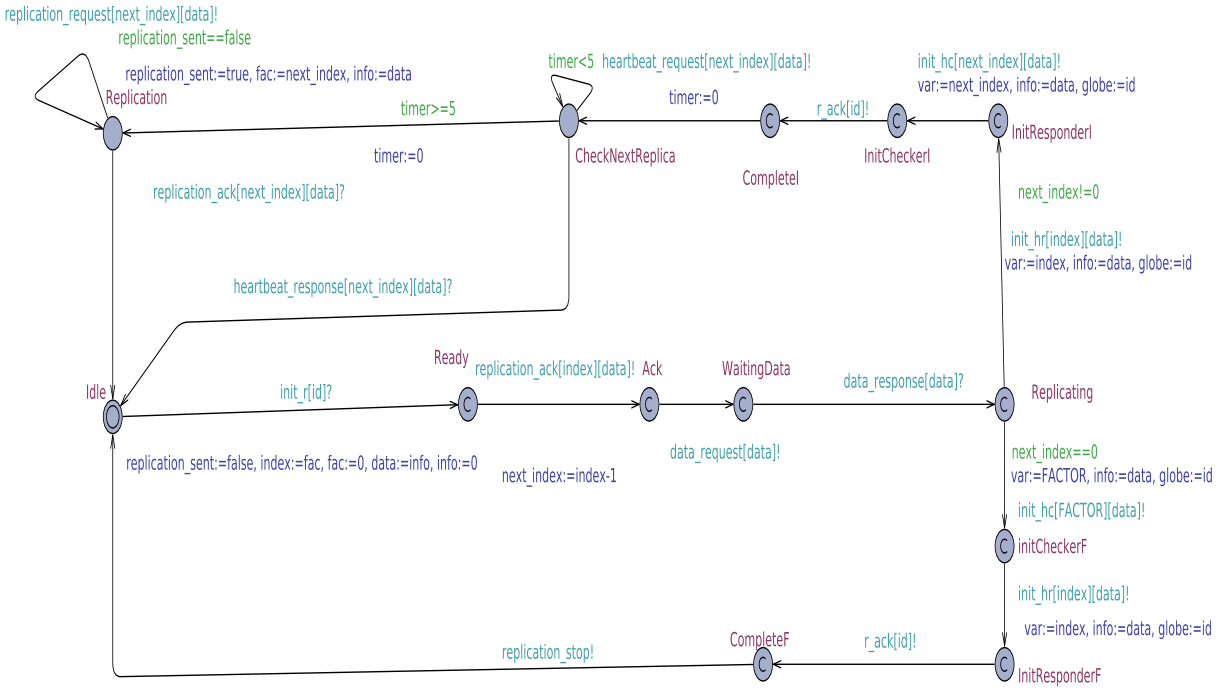

Figure 4.7: Replication model

HeartbeatChecker Model

This model (Figure 4.8) represents a sub-process in the heartbeat process which consists in checking the availability of a replica. This is done by the node which sent the replication request for that replica.

This model has six states:

- Idle: initial state;
- Ready: state from which the heartbeat *Interests* are sent;
- Replication: state from which a replication *Interest* is sent in replacement of a node which failed;
- CheckingNode: state in which a replica has been checked and response is waited;
- Detected: when a timeout for a heartbeat *Interest* is detected;
- Failure: This state is used to simulate the failure of the node. When a Storage automaton goes into end, all processes related to it (HeartbeatChecker and HeartbeatResponder) do the same.

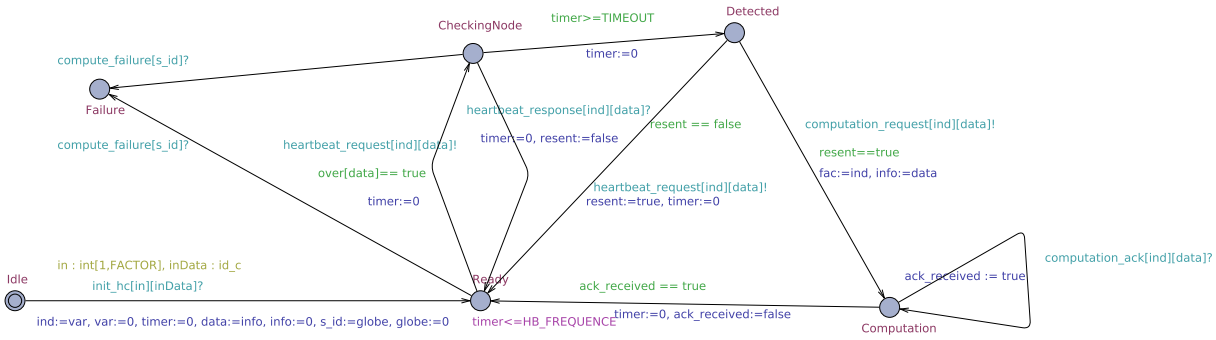

Figure 4.8: HeartbeatChecker model

HeartbeatResponder Model

This model (Figure 4.9) represents the second sub-process in the heartbeat process which consists in responding about the availability of a replica when checked by a HeartbeatChecker.

This model has three states:

- Idle is the initial state
- Ready: this state is reached after initialization using the synchronization channel called $init\_hr[index][id\_data]$? (with *index* for the replication index and id_data used to identify the data to replicate) by a Replication automaton (Figure 4.7). During this initialization, all the information about the replica to check are received. In the Ready state, when the automaton receives an heartbeat *Interest* from the HeartbeatChecker via the synchronization channel $heartbeat\_request[in][data]$? (with *in* for the replication index and *data*, the data identifier), it sends back a confirmation message using the channel $heartbeat\_response$!.
- Failure: this state represents the end of the process. It is reached when the Storage having initialized the process fails.

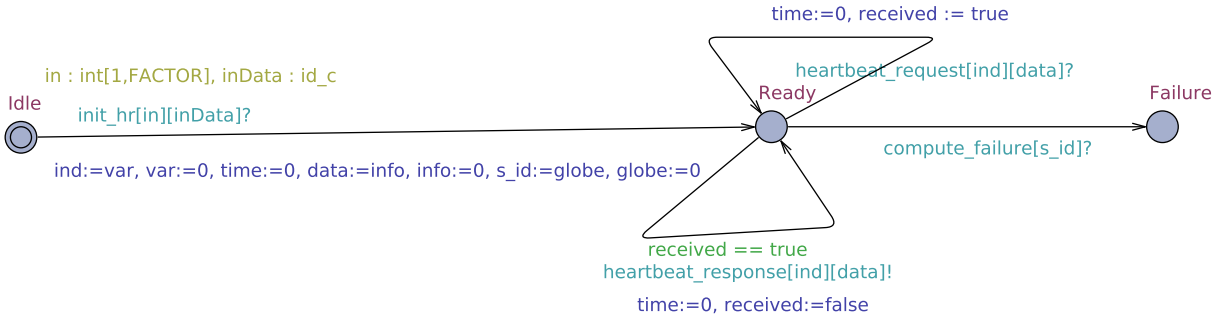

Figure 4.9: HeartbeatResponder model

ReplicationClient Model

This model represents the ReplicationClient, which is used to send a replication request to replicate data on the DFS. The automaton has six states.

- Idle: initial state;
- Start: state from which the automaton is ready to send replication *Interest*. This is done by synchronizing with a Storage automaton using *replication_request*[FACTOR][*data*]! (Figure 4.6) ;
- WaitAck: after sending a replication request, the automaton goes into this state and waits for the acknowledgement that its request has been received by the Storage automaton. This is done by a Replication automaton through a synchronization on channel *replication_ack*[FACTOR][*data*]?, the automaton then changes to the Wait state (Figure 4.2);
- Wait: state in which the ReplicationClient waits for an *Interest* request for the data to be replicated, in this case it moves to the SendData state or receives an *Interest* about the completion of the replication process, then it evolves to the End state.
- SendData: in this state, the ReplicationClient automaton sends a data message to the node requesting the data. This is done using the synchronization channel *data_response*[*data*]!.
- End: This state is reached at the end of the replication process after receiving a confirmation from a Replication automaton using the synchronization channel *replication_stop*?.

User Model

The User model (Figure 4.3) is used to represent data retrieval from the system. It has five states:

- Idle: initial state;
- Start: state from which the automaton is ready to send *Interest* to retrieve data. This is done by synchronizing with a Storage replicating the data using *data_request*[*data*]! (Figure 4.3);

- WaitData: after sending data request, the automaton goes into this state and waits for the data. The response will come by a Storage automaton through a synchronization on channel *data_response*[*data*]?,then this automaton evolves to state DataReceived;
- DataReceived: state meaning that the User has received a copy of the data.
- End: This state is reached at the end of the data request process.

4.3.2 NMapReduce

The NMapReduce is composed by a set of Compute Nodes (Section 3.2). It is the second layer of our Big data architecture. In this section, we describe the timed automata templates used to define the NMapReduce. The global system is composed of four automata describing a Compute node (Figure 4.10). Two other automata are used to modeled the ComputationClient application used to request data computation on the system, and also a code application which contains the script of the program to use. The HeartbeatChecher (Figure 4.8) and the HeartbeatResponder (Figure 4.9) are shared components with the DFS.

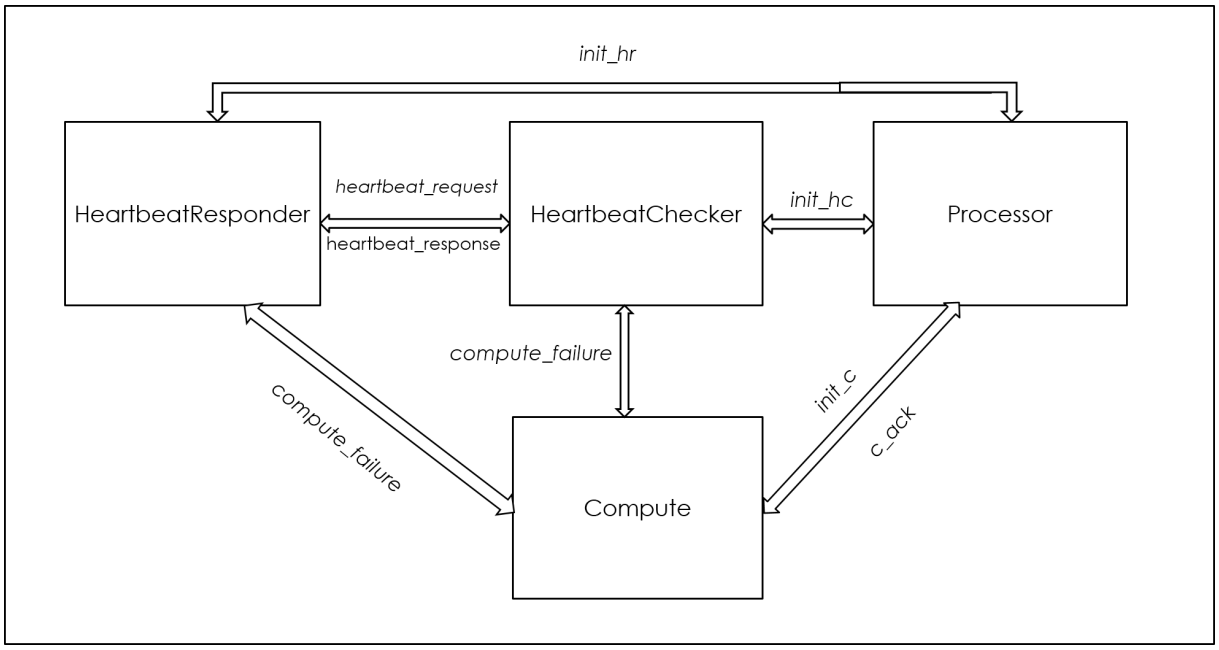

Figure 4.10: Compute node automata network

Compute Model

This model represents a global view of the compute node. It has seven states:

- Idle: initial state, or inactivity state;
- Start: state in which the compute node is able to process requests. It can be requested from a ComputationClient (for computation) or the other compute nodes (for map result retrieval).

- Ready: this state is reached after a synchronization on the channel called *computation_request*[*inIndex*][*inData*]? (with *inIndex* for the computation index and *inData* used to identify the data) with the ComputationClient. This state corresponds to a state in which the compute node has received a computation request from the ComputationClient.
- Computing: this state is reached after the compute node initiates a Processor model to handle the request. The initiation is done through the use of a channel *init_r*[*id*]!. This state indicates that the node is processing the request.
- MapComplete: this state is reached after the synchronization on the channel *r_ack*[*id*]? by the Processor model, which uses that channel to notify the completion of the map process. The system can move to the Start state waiting for another computation request, or request for the map it has just completed.
- SendMap: this state is reached after a synchronization on *data_request*[*in*]?, which corresponds to the reception of an *Interest* for a map computed by the node. The data are sent to the requestor using a synchronization on the channel *map_response*[*outDataChan*]!.
- Failure: this state is used to simulate in a non deterministic way the failure of a node.

Figure 4.11 shows the model from UPPAAL.

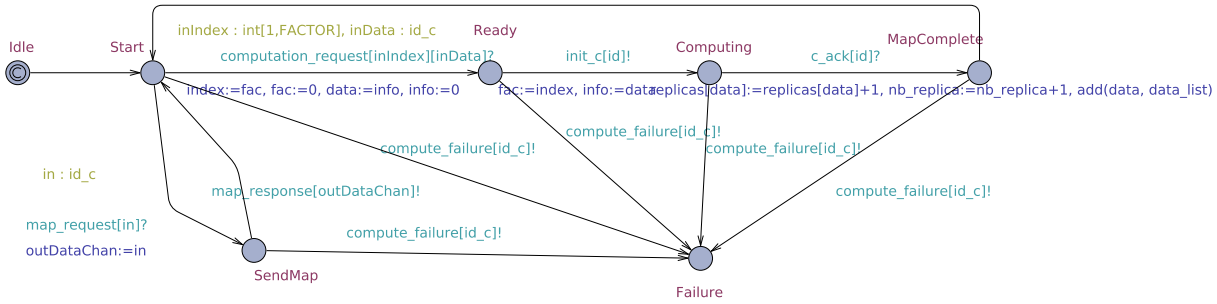

Figure 4.11: Compute model

Processor Model

This model represents the process in charge of computing the map and the reduce for a storage node. It has fourteen states.

- Idle: initial state;
- Ready: this state is reached after the initialization by the Compute process using *init_r*[*id*]? channel.
- Ack: this state is reached after the Processor has answered with its ability to deal with the computation. This acknowledgement is sent directly to the ComputationClient using *computation_ack*[*index*][*data*]!.
- Waiting: this state is reached after sending a request for the script that has to be applied on the data to the code node (Figure 4.14);

- ComputingMap: when the script is received, the map computation on the requested segment is performed in this state;
- CompleteInitI: this state is reached after the initialization of the heartbeat process in the states InitResponderI and InitCheckerI (Figure 4.12) and when there is a need to send a compute request to another node;
- CheckNext: in this state, an heartbeat *Interest* is sent to check whether the next index in the computation process has already been processed or not.
- SendComputation: this state is reached when there is a need to send a computation request to another compute node;
- ComputeReduce: this state is reached after the initialization of the heartbeat process in the states InitResponderF and InitCheckerF (Figure 4.12) and when there is no need to send a compute request to another node. In this state the reduce action will be performed. After computation, the result is sent to the ComputationClient using the channel *computation_result*!
- MapRequest: from this state a node sends an *Interest* to retrieve all the intermediate map computation results in order to perform the reduce.

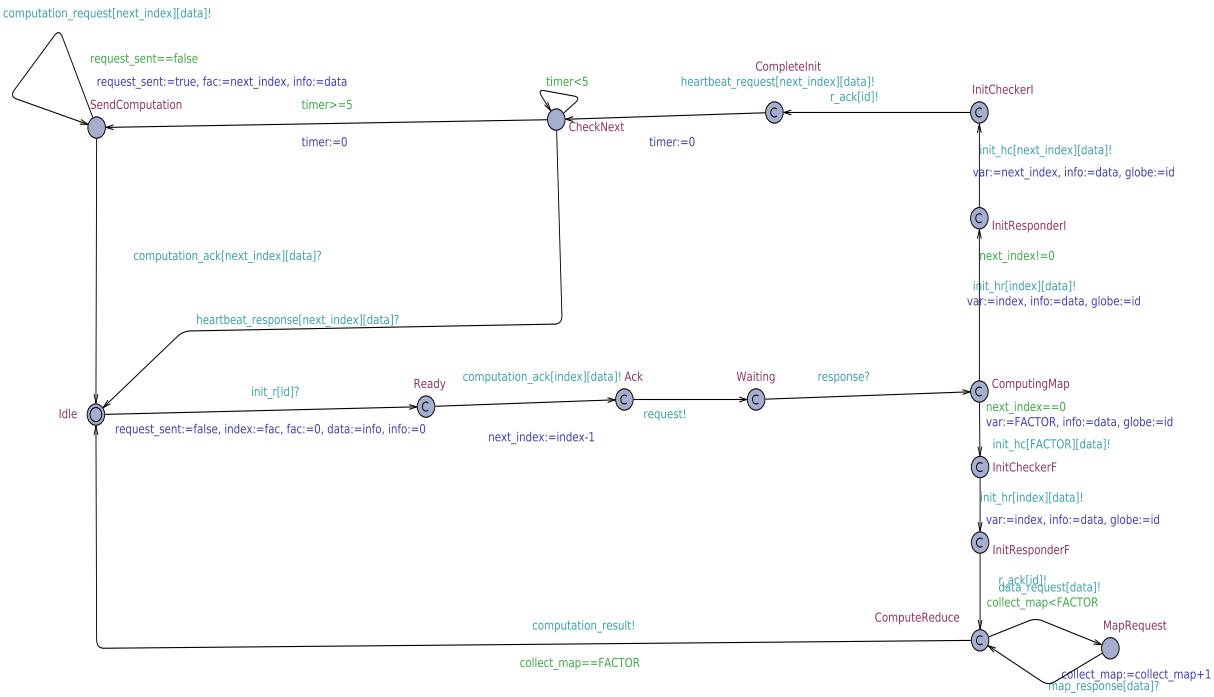

Figure 4.12: Processor model

ComputationClient Model

This model represents the ComputationClient, which is used to send a computation request on data. The automaton has five states.

- Idle: initial state;
- Start: state from which the node is ready to send computation *Interest*. This is done by sending the request through $computation\_request[index][data]!$;
- WaitAck: this state is reached after sending a compute request. The client is waiting for the acknowledgement that its request has been received.
- Wait: this state is reached after the reception of the acknowledgment on channel $computation\_ack[index][data]$. The ComputationClient waits then for the result of its data computation request (Figure 4.13).
- End: this state is reached at the end of the computation process after receiving a data corresponding to the result of the computation on the synchronization channel $computation\_result?$.

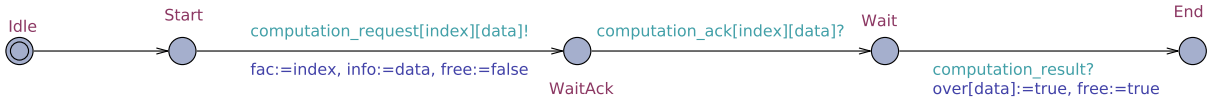

Figure 4.13: ComputationClient model

Code Model

The Code model (4.14) is used to represent a node holding the script for the computation. Has described (Section 3.2), the script is considered as a data on the cluster. It has three states:

- Idle: initial state;
- Start: state from which the node is ready to respond to *Interest* for script retrieval. This is done by sending a message on the channel $code\_request?$;
- Sending: this state is reached after receiving an *Interest* for the script. The script is then sent using $code\_response!$ to the requesting node.

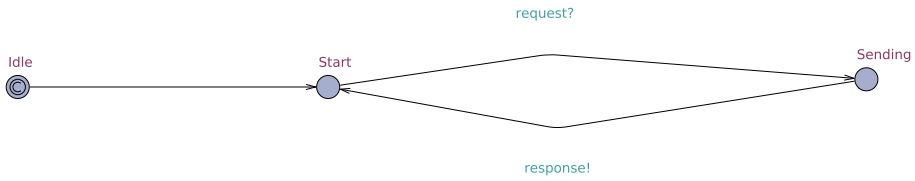

Figure 4.14: Code model

4.4 System Verification

In this section, we are interested in the verification of system properties that our system must satisfy. Those properties are proved by applying model checking to our previous models. Our architecture is defined in two layers as presented in Section 4.3.1 and Section 4.3.2, with each layer

having its own properties. The properties are related to the replication and the computation distribution. We have divided the properties into 3 categories, respectively communication, completeness and recovery after failure. These are essential properties that a distributed system should meet [51].

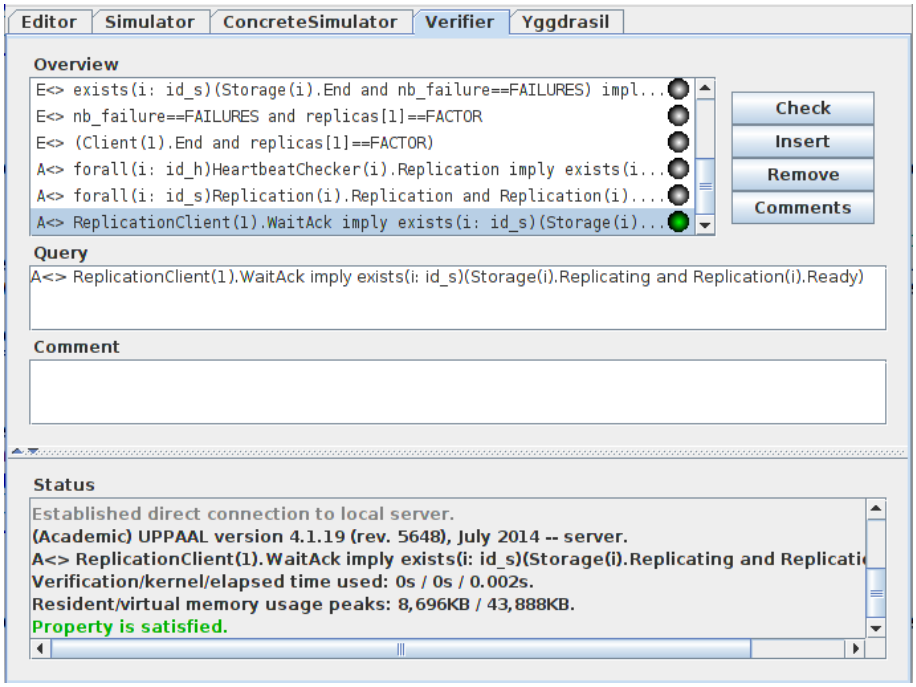

Figure 4.15: Property P1 UPPAAL verification

4.4.1 Communication properties

We have identified properties related to the communication between the ReplicationClient and Storage (Section 4.3.1) and between the Compute and ComputationClient (Section 4.3.2). First, we present those related to the NDFS and then those related the NMapReduce.

A replication message sent by a node is eventually received at some point in time by a Storage node. This property is verified using three formulas, as we have three possible cases. The replication message can be sent by a ReplicationClient, a Storage automaton or a HeartbeatChecker.

P1: `A<> ReplicationClient(1).WaitAck imply exists(i:`id_s`)(Storage(i).replicating and Replication(i).ready)`

This property covers the case where the replication message is sent by a ReplicationClient node. It expresses the fact that when a ReplicationClient reaches the state WaitAck, a replication message has been sent, a Storage reaches the state Replicating and the replication process is Ready, meaning that the replication message has been received by this Storage. The property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.15. The green bullet in the overview indicates that the system satisfies the specified property. The lower part has logged the communication with the model-checking engine and explicitly mentions the property satisfaction.

P2: `A<> forall(i:`id_s`)Replication(i).Replication and Replication(i).`$replication\_sent == true$` imply exists(i:`id_s`)(Storage(i).Replicating and Replication(i).Ready)`

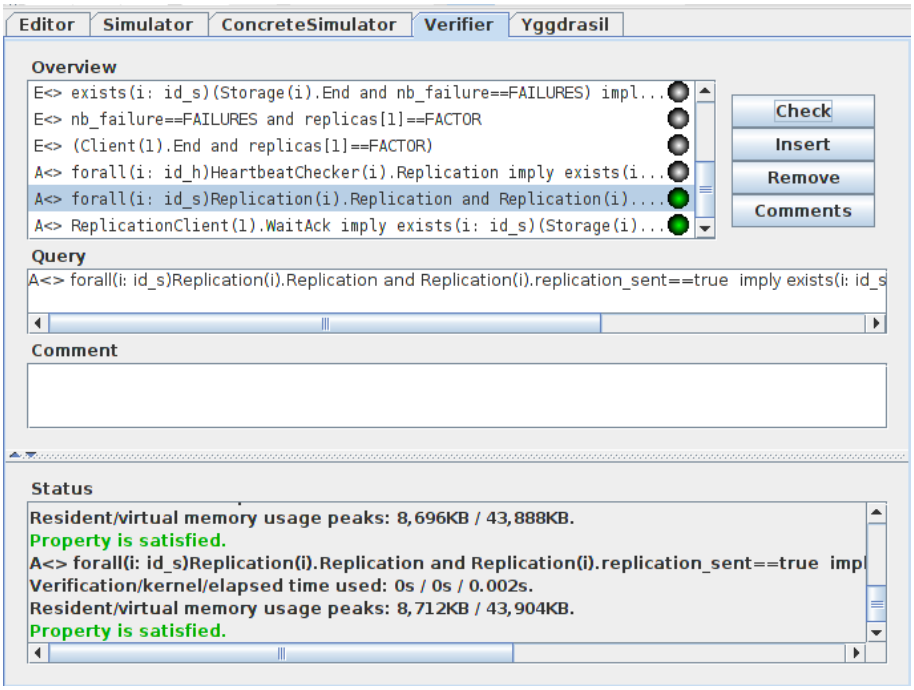

Figure 4.16: Property P2 UPPAAL verification

This property concerns the case where the replication message is sent by a Storage node during the replication process. It expresses the fact that when a Replication automaton reaches the state Replication with the variable *replication_sent* is true, a replication *Interest* has been sent by this node. A Storage then reaches the state Replicating and the replication process is Ready, meaning that the replication message has been received by this Storage. The property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.16. The green bullet in the overview indicates that the system satisfies the specified property. The lower part has logged the communication with the model-checking engine and explicitly mentions the property satisfaction.

P3: `A<> forall(i:`*id_h*`)HeartbeatChecker(i).Replication imply exists(i:`*id_s*`)(Storage(i).Replicating and Replication(i).Ready)`

This property is for the case where the replication message is sent by a Storage node during the Heartbeat process. It expresses the fact that when a HeartbeatChecker reaches the state Replication, a replication *Interest* has been sent by this node. A Storage then reaches the state Replicating and the Replication process is Ready, meaning that the replication message has been received by this Storage. This property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.17. The green bullet in the overview indicates that the system satisfies the specified property. The lower part has logged the communication with the model-checking engine which explicitly mentions the property satisfaction.

Another aspect of this property is the one related to the computation. A computation message sent by a node is eventually received at some point in time by a Compute node. This property is verified using three formulas, as we have three possible cases: the computation message can be sent by a ComputationClient, a Compute automaton or a HeartbeatChecker.

P4: `A<> ComputationClient(1).WaitAck imply exists(i:`*id_s*`)(Compute(i).Computing`

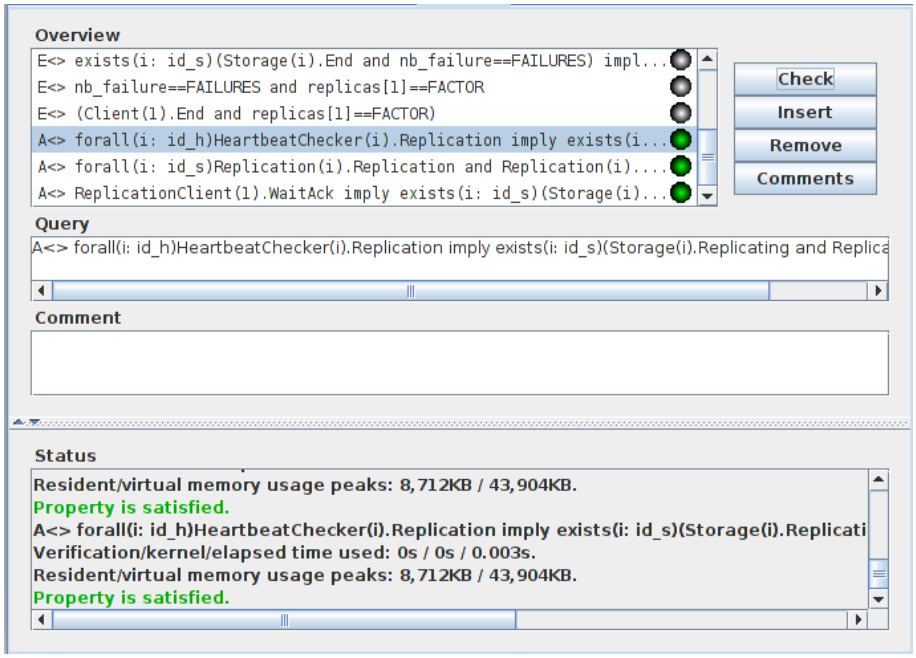

Figure 4.17: Property P3 UPPAAL verification

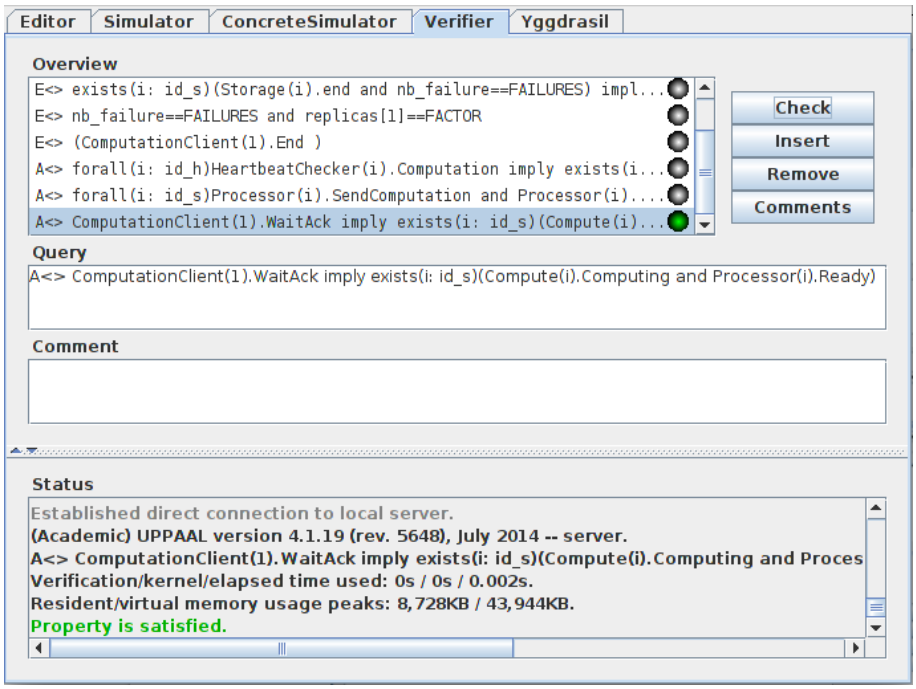

Figure 4.18: Property P4 UPPAAL verification

`and Processor(i).Ready)`

This property covers the case where the computation message is sent by a computation client node. It expresses the fact that when a ComputationClient reaches the state WaitAck, a computation message has been sent and a Compute node reaches the state Computing and the Processor process is Ready, meaning that the computation message has been received. This

property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.18. The green bullet in the overview indicates that the system satisfies the specified property.

P5: `A<> forall(i:`*id_s*`)Processor(i).SendComputation and Processor(i).`*request_sent* == *true* `imply exists(i:`*id_s*`)(Compute(i).Computing and Processor(i).Ready)`

This property concerns the case where the computation message is sent by a Compute node during the computation process. It expresses the fact that when a Processor reaches the state SendComputation with the variable *request_sent* is true, a computation *Interest* has been sent by this node. A Compute automaton then reaches the state Computing and the Processor process is Ready, meaning that the computation message has been received. This property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.19. The green bullet in the overview indicates that the system satisfies the specified property.

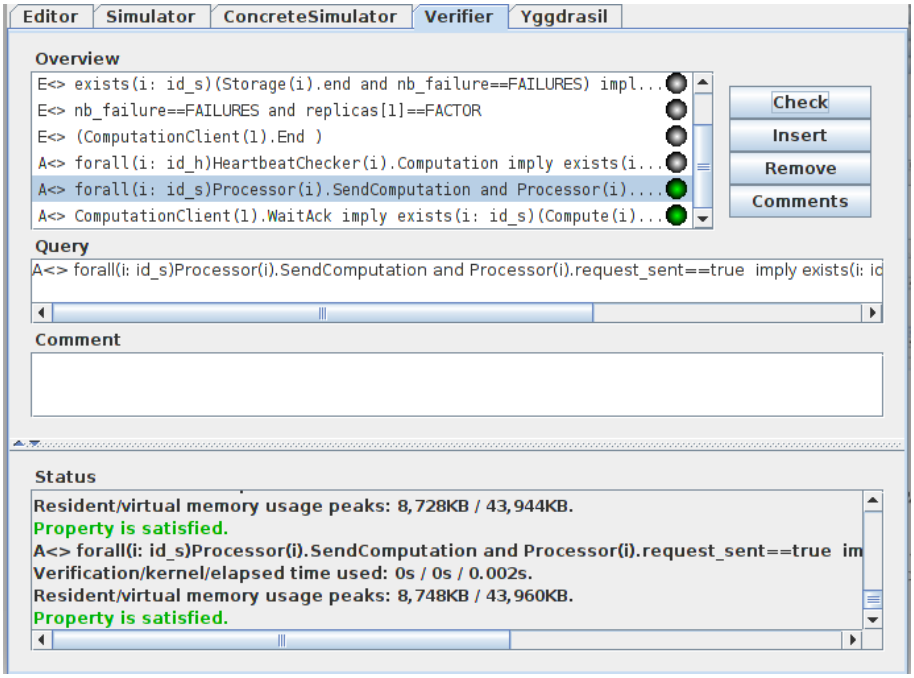

Figure 4.19: Property P5 UPPAAL verification

P6: `A<> forall(i:`*id_h*`)HeartbeatChecker(i).Computation imply exists(i:`*id_s*`)(Compute(i).Computing and Processor(i).Ready)`

This property describes the case where the computation message is sent by the Heartbeat process when a node fails. It expresses the fact that when a HeartbeatChecker reaches the state Computation, a computation *Interest* has been sent. A Compute then reaches the state Computing and the Processor process is Ready, meaning that the computation message has been received. This property has been specified as a query to the system in the upper section of the verifier in UPPAAL GUI as displayed in Figure 4.20. The green bullet in the overview indicates that the system satisfies the specified property.

The communication properties from P1 to P6 related to our two layers are very important. They prove with certainty that the protocols correctly transfer messages from ReplicationClient to Storage, from ComputationClient to Compute, between Storages and between Computes.

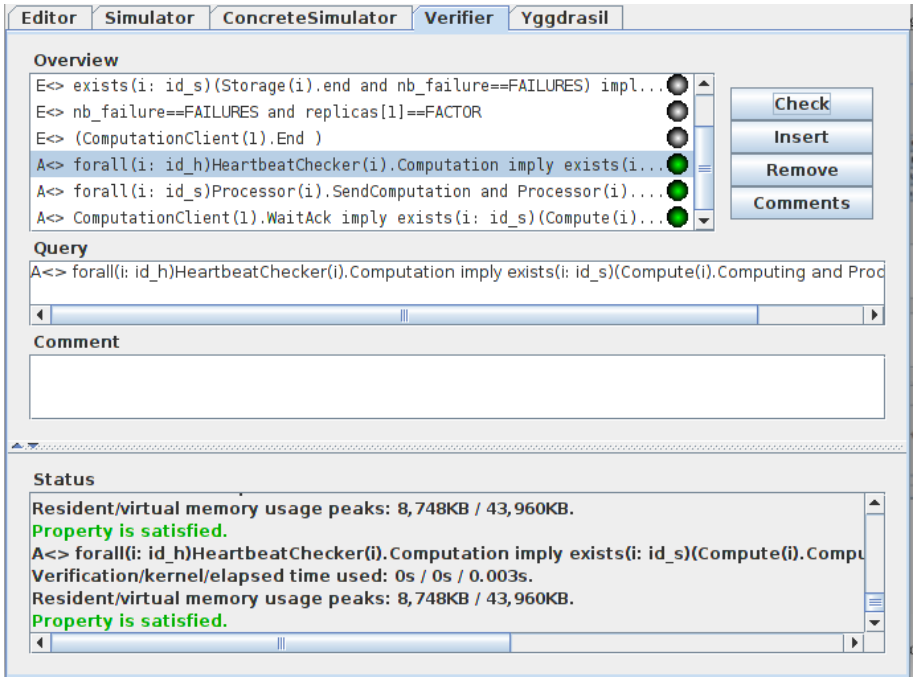

Figure 4.20: Property P6 UPPAAL verification

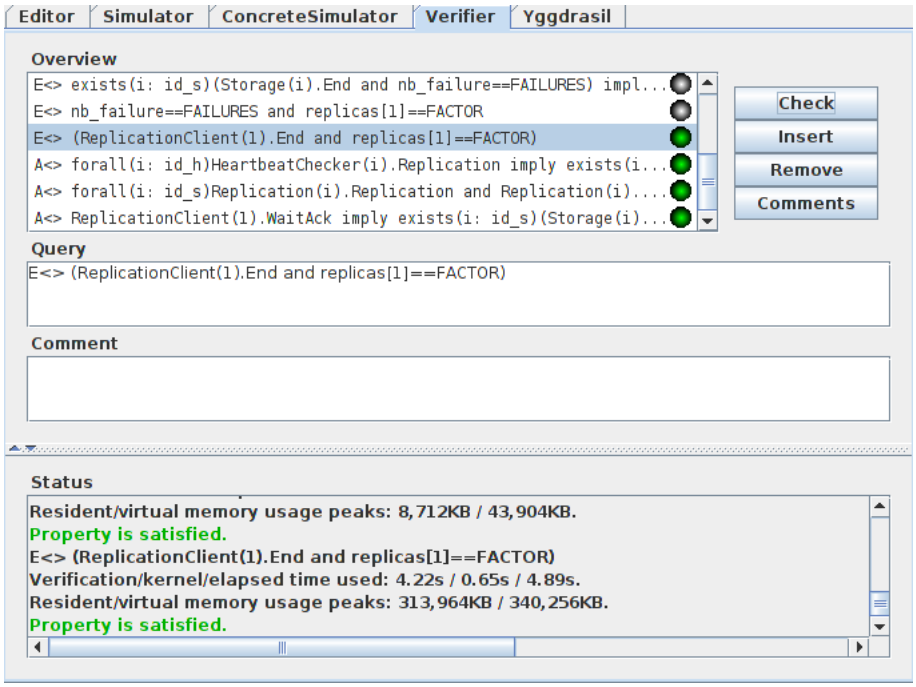

Figure 4.21: Property P7 UPPAAL verification

4.4.2 Completeness properties

These properties are related to the execution of a request. The first one is about the completeness of the replication process: the replication process completes.

P7: `E<> (ReplicationClient(1).End and replicas[1]==FACTOR)`

This property means that after a replication *Interest* has been sent by a ReplicationClient, it reaches the End state at some point in time and the number of replica requested is equal to the number of replica available on the cluster. This property has been verified using the UPPAAL verification system as shown in Figure 4.21. This property has been specified as a query to the system in the upper section of the verifier. The green bullet in the overview indicates that the system satisfies the specified property.

The second one is about the completeness of the computation process: the computation process completes.

P8: `E<> (ComputationClient(1).End)`

This property means that after a computation *Interest* has been sent by a ComputationClient, the ComputationClient at some point reaches the state End, meaning that it has received a message from a Compute node. This property has been verified using the UPPAAL verification system as shown in Figure 4.22. The green bullet in the overview indicates that the system satisfies the specified property.

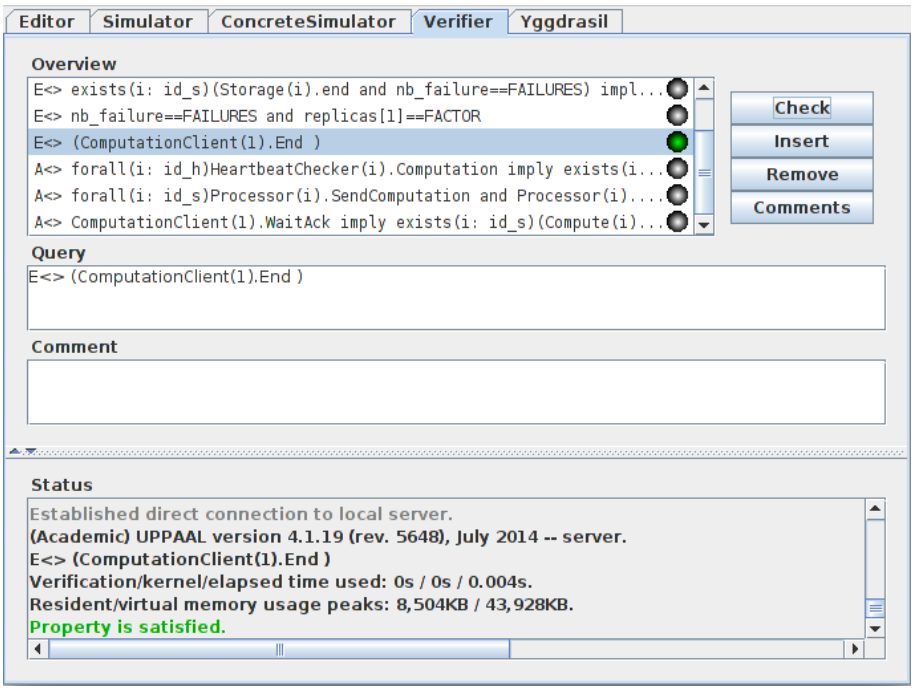

Figure 4.22: Property P8 UPPAAL verification

P9: `A[] not deadlock`

We also express a property regarding deadlock freeness. Deadlock is a very common problem in distributed systems. It is a state of the system in which each component is forever waiting for another component to take an action which will never occur. This property has been verified using the UPPAAL verification system. We have faced some issues trying to run this property using UPPAAL user interface. We used the command line version of the verifier to specify the property as a query to the system. The output (Figure 4.23) shows the property, the result and some metrics. The message "Formula is satisfied" indicates the property satisfaction.

The completeness properties from P7 to P9 related to our two layers guarantee that the different components work correctly to perform the required actions without any deadlock.

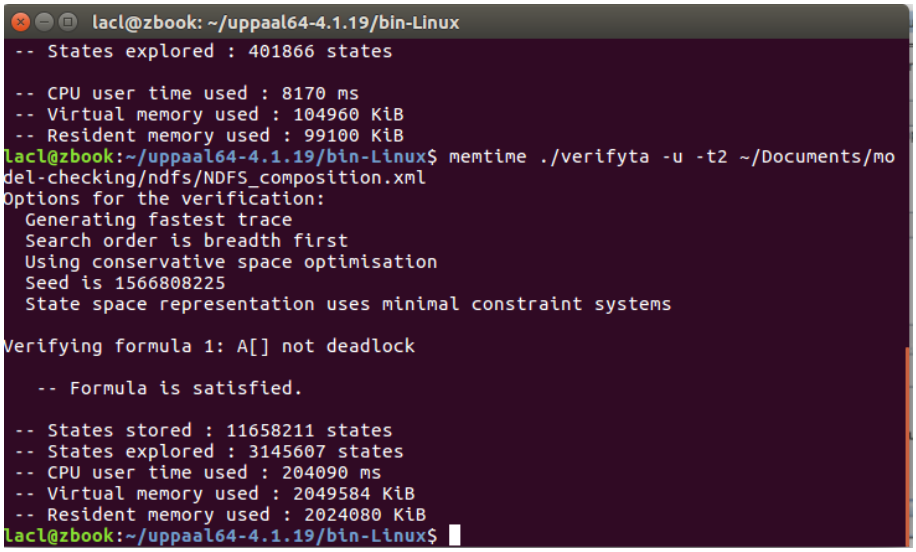

Figure 4.23: Property P9 UPPAAL verification

4.4.3 Recovery properties

These properties highlight our heartbeat process and our failover approach.

P10: `E<> Storage(1).Failure imply exists(i:`*id_h*`)(HeartbeatChecker(i).timer > HeartbeatChecker(i).TIMEOUT) and HeartbeatChecker(i).Detected`

This property means that when a Storage node fails, it is detected by the heartbeat. This property has been verified using the UPPAAL verification system as shown in Figure 4.24. The green bullet in the overview indicates that the system satisfies the specified property.

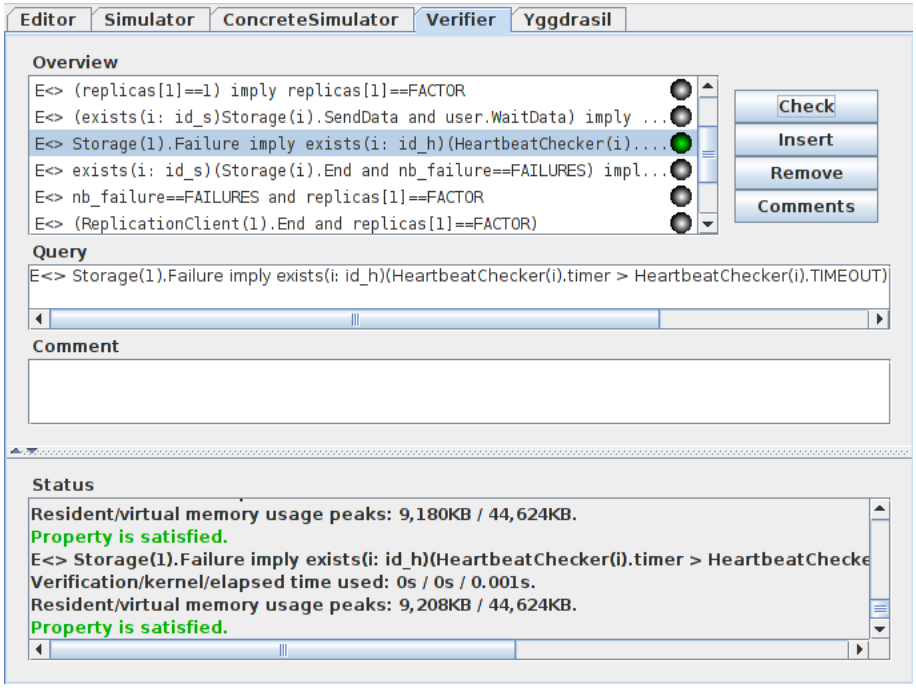

Figure 4.24: Property P10 UPPAAL verification

P11: `E<>Storage(1).Failure imply exists (i:`*id_s*`)(Storage(i).index==Storage(1).index)`

This property verifies that when a data node fails, there is always a component which will take over the data managed by this node. Figure 4.25 gives the proof from UPPAAL verification system. The green bullet in the overview indicates that the system satisfies the specified property.

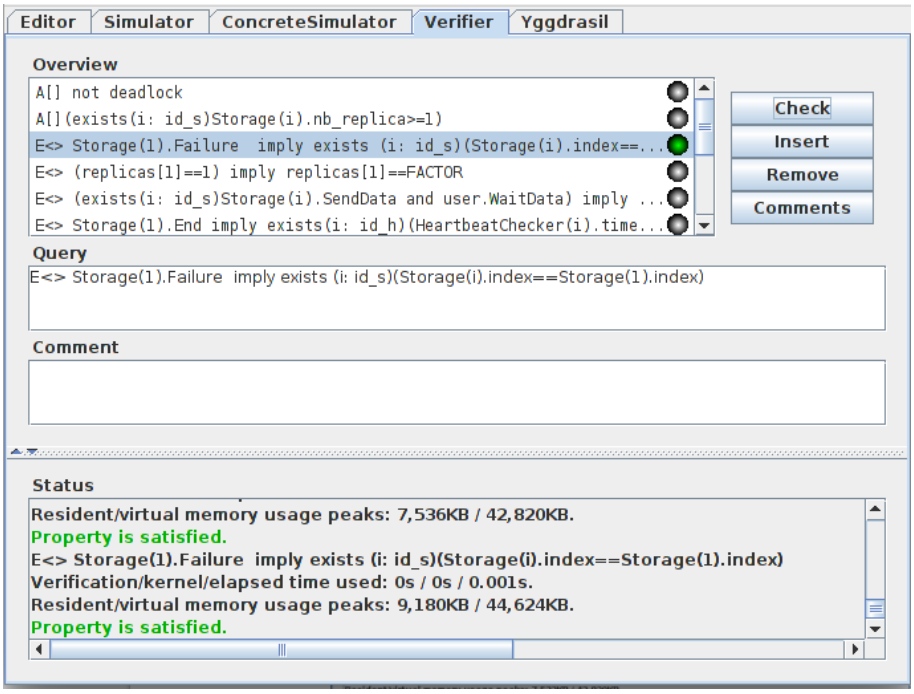

Figure 4.25: Property P11 UPPAAL verification

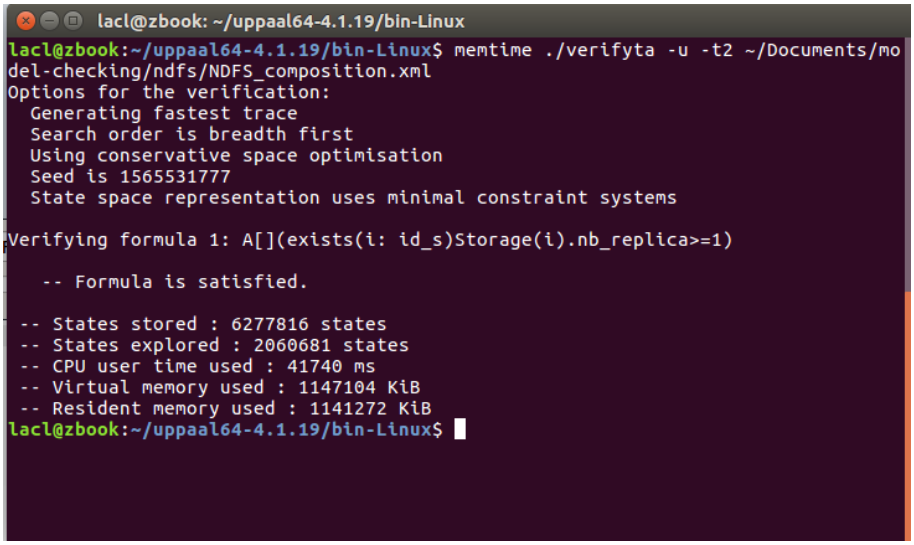

Figure 4.26: Property P12 UPPAAL verification

P12: `A[](exists(i:`*id_s*`)Storage(i).`*nb_replica*`>=1)`

There is always at least one replica on the cluster. After the replication is completed, the number of replica related to a data is at least one. Figure 4.26 gives the proof from UPPAAL

verification system. The output of the command line version of the verifier displays the message "Formula is satisfied", which indicates the property satisfaction.

P13: `E<> (replicas[1]==1) imply replicas[1]==FACTOR`

The whole replication can be rebuilt from one replica. If the number of replica related to a data is one at some point in time, the number of replication will be the same as the number initially requested by the client after some time. Figure 4.27 gives the proof from UPPAAL. The green bullet in the overview indicates that the system satisfies the specified property. verification system.

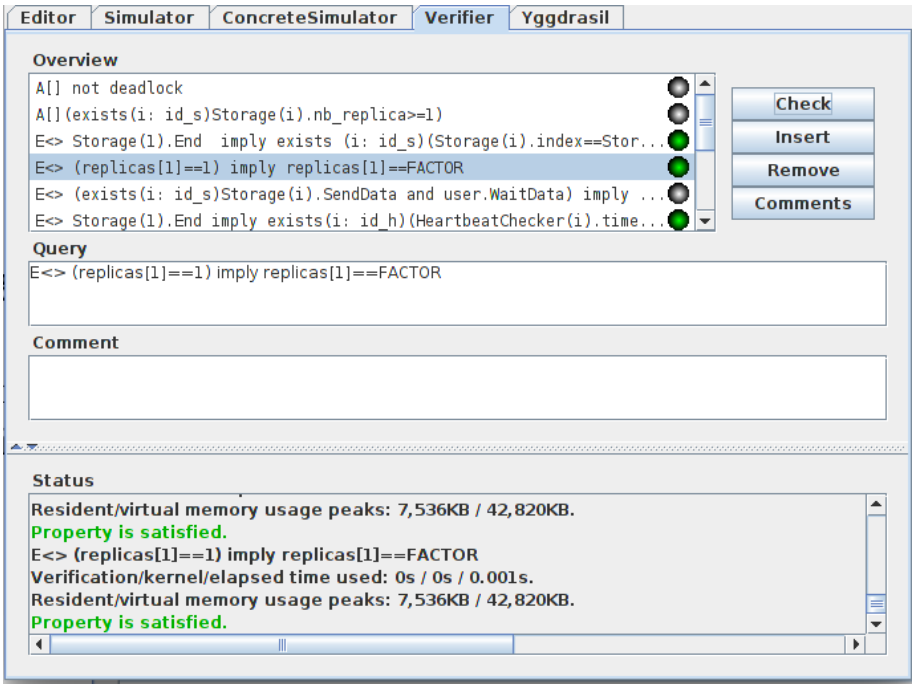

Figure 4.27: Property P13 UPPAAL verification

The recovery properties from P10 to P13 prove that our system is fault tolerant. In fact, there is a continuity of services despite a node failure.

4.5 Summary

In this chapter, we have explored potential formal verification possibilities for our system.

Traditional approaches to system verification very often consider only the system in a global view when proving properties about it. Properties related to the internal behavior of the different components composing the system should also be proved. Combining model checking with Coq proof assistant help us to achieve this goal. First in Chapter 3, we have proved properties related to each components internal functionality and in this chapter, we considered properties for the interaction between these components. Our system is modeled into a network of timed automata, which allowed us to perform temporal system properties verification using model checking techniques.

The properties to prove have been chosen regarding essential properties Big Data architectures should satisfy, mainly recovery properties related to fail-over. Having nodes automatically taking over when a node fails is important for DFS and computation distribution. Through these

properties, we have shown the fault tolerant aspect of our system. In fact, despite the inevitable interruptions caused by problems with equipment, normal functions can be maintained. We have also shown through communication properties verification that the different components of our system communicate correctly to perform the desired functionalities which are then proved using completeness properties. We were also able to prove that the system is free of communication issues such as deadlock. All these proofs give us more confidence in the system to implement. These properties should be preserved during the implementation phase. In the next chapter (Chapter 5), we propose a prototype and then an implementation of the system based on the specification, which satisfies the properties.

Chapter 5

Prototyping, Implementation and Simulation

Contents

To validate the different approaches introduced in Chapters 3 and 4, early in our work, we propose a prototype version of our solution.

Our prototype is built with the following goal: evaluate and validate the network communication aspects implied by our approach using the network simulator ndnSIM. It helps us to save time, but also to consider the validation of the scalability aspect of our approach that would be difficult to perform using a concrete implementation due to resource limitations related to our project. The implementation language is C++.

We then perform a concrete implementation of the solution. Our implementation is performed to confirm the network communication results obtained through simulation, but also to have a version that can run on physical devices for being able to compare our approach to existing solutions. This version of our architecture is developed in JavaScript and can run on any platform supported by the CCNx library.

In Section 5.1, we present the software architecture, Section 5.2 deals with our implementation approach. Section 5.3 describes our simulation experiments.

5.1 Software Architecture

In this section, we present our system requirements, the components involved in the construction of our system, but also the relation between these components and the interfaces they expose. Those components are coming from our formal specifications in Chapters 3 and 4.

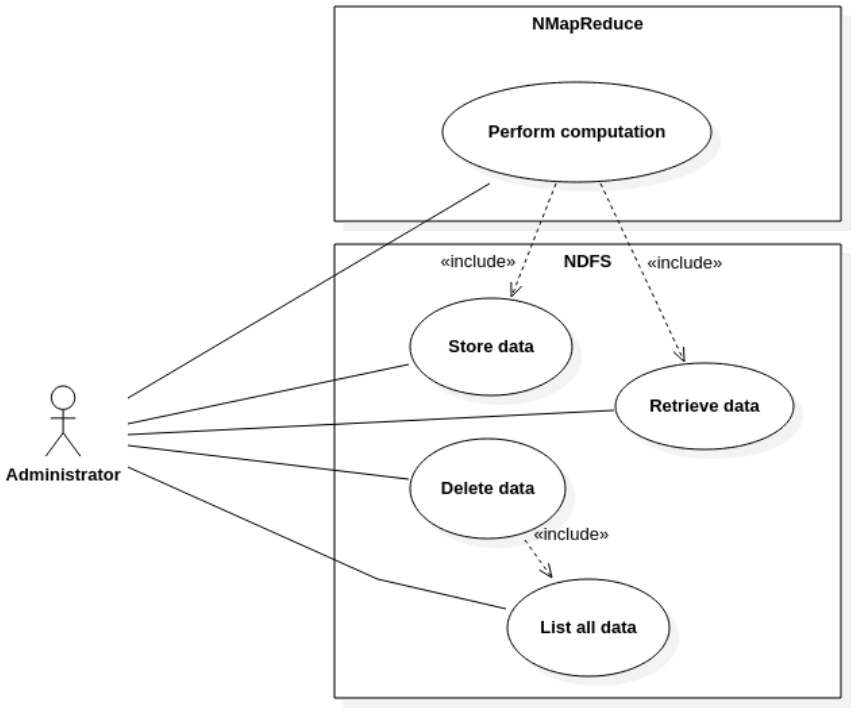

Figure 5.1: Use Case diagram

5.1.1 Requirements

This section presents the requirements for our system. These requirements are grouped into three categories: functional requirements, non-functional requirements, and constraints.

Functional requirements

The functional requirements are related to the functions the system has to accomplish. One way to represent them is through the use of a Use Case Model (Figure 5.1). Our system is packaged into subsystems: NDFS and NMapReduce. We describe the most important use cases for our software architecture. The use cases that we have retained are the following:

- **Store data**: The user wants to save a file on the DFS. He sends a replication request and defines the replication factor (number of replicas he wants to be available on the network), he then receives a confirmation about his request.
- **Retrieve data**: The user or an application wants to get the content of data stored on the DFS. It sends a retrieval request specifying the name of the file. A copy of the file is then returned to the requester. It is good to mention that, as the DFS is based on NDN, any

application able to send an NDN *Interest* using the name of the file can be used to get the content of the data.

- **Delete data**: The user wants to remove a file previously stored on the DFS. He sends a delete request specifying the name of the file to delete, and gets an acknowledgement about it.

- **List all data**: The user needs to have an overview of the files stored on the DFS. He sends a display request and gets a list of the different files stored on the DFS.

- **Perform a computation**: The user wants to perform a computation on data. He sends a computation request, specifying the name of the file, the script to be applied to the data. He receives back the name which has to be used for an *Interest* to get the result of the computation.

Non-functional requirements

Non-functional requirements are properties that are not directly used by the user to interact with the system but are important for its good performing. In our case, we have defined the following non-functional requirements:

- **Usability**: The usage of the system should be intuitive to the end-users.

- **Performance**: Big data architects are looking for solutions which help them to quickly get results from their data. Therefore, the system should operate in such a way that the time needed to store a file or perform a computation is minimal.

- **Reliability**: The system should be fault-tolerant, and able to recover from a disaster. In the case of replication, the system should be able to maintain the exact number of replica requested by a user for every data replicated on the DFS.

- **Portability**: The system should be developed in such a way that it is platform independent, and being able to operate on low constrained devices.

- **Scalability**: The system should support an increase in the number of users and also in the number of physical devices.

Constraints

A constraint is a restriction on a part of the system or on the whole system. For our architecture, we have the following constraints:

- the system should integrate the replication and computation parsers extracted in Ocaml from the Coq specification (Section 3.5);

- the system should provide a Command Line Interface (CLI) for the end-user to interact with the system.

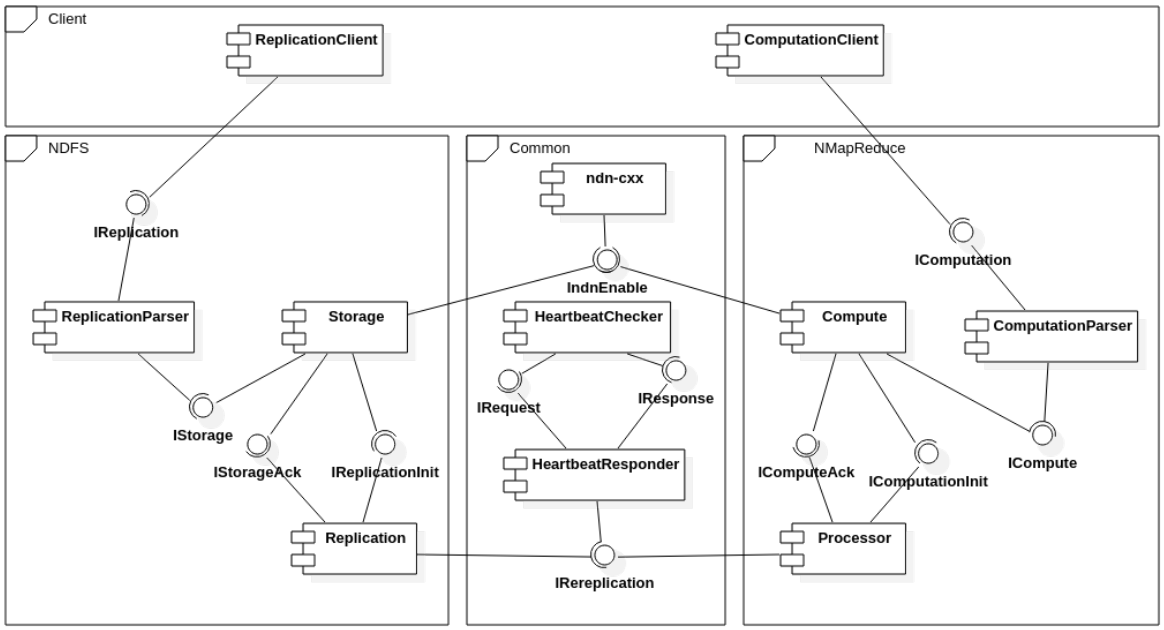

Figure 5.2: Component diagram

5.1.2 Component diagram

The component diagram describes the system as reusable components and highlights their dependency relationships. The architecture that we propose is composed of two layers, the first being the DFS (Section 3.1) and the second being the computation distribution (Section 3.2). The different components for our system have been obtained from our specification in Chapter 3 and our model in Chapter 4. Figure 5.2 shows the component diagram, which can be organized into 4 groups:

- NDFS: This package contains the different components related to the DFS. It is composed with:
  - ReplicationParser: This component is an implementation of the parser for the replication language. This component is extracted in Ocaml from the Coq specification (Section 3.5.3).
  - Storage: This component implements the management of a Storage node, activates the replication mode, responds to replication requests and also sends data when replicas are available at the node. It has been implemented from our specification from Chapter 4. It's the implementation of the Storage model (Section 4.3.1).
  - Replication: This component is the implementation of the Replication model (Section 4.3.1). It implements the process in charge of performing the replication for a storage node.
- NMapReduce: This package contains the different components related to the computation distribution.

  - ComputationParser: This component is an implementation of the parser for the computation language. This component is extracted in Ocaml from the Coq specification (Section 3.5.3).
  - Compute: This component implements the management of a Compute node, activates the computation mode, responds to computation requests. It is the implementation of the Compute model (Section 4.3.2).
  - Processor: Implemented from the Processor model (Section 4.3.2), this is the component implementing the computation mechanism.

- Common : This package contains the different components shared between the DFS and the NMapReduce.

  - HeartbeatChecker: This component implements the heartbeat process used to check the availability of an element (for example a replica in NDFS and a map in NMapReduce). It's the implementation of the HeartbeatChecker Model (Section 4.3.1). Messages are sent periodically to detect whether replica or map are still alive. Responses are provided by a HeartbeatResponder.
  - HeartbeatResponder: This component implements the heartbeat process used to send a response about the availability of an element (this is the case for a replica in NDFS and a map in NMapReduce). It has been implemented using the HeartbeatResponder Model (Section 4.3.1). It responds to requests from HeartbeatChecker about availability of a replica or map.
  - ndn-cxx: This component is a C++14 library implementing Named Data Networking (NDN) primitives that can be used to write various NDN applications [143].

- Client: We also considered additional components for the different clients (ReplicationClient and ComputationClient).

  - ReplicationClient: Implemented from the ReplicationClient Model (Section 4.3.1), this component implements the process used to send a replication request to replicate data on the DFS.
  - ComputationClient: This component implements the process used to send a computation request on the data. It has been implemented from the ComputationClient (Section 4.3.2).

5.1.3 Scenarios

In this part, we highlight interactions between the administrator actions and the system. For each use case, we may have 3 parts. A normal course of events, which describes the ideal course of actions, where everything is fine. Alternative courses, which are about describing the different possible stages related to the choices of the administrator. This is the case for stages linked to conditions. Exceptions, when stages of the normal course of events could be disrupted due to abnormal events. We only present a sequence diagram for the normal course of events.

We abstract the communications with the forwarder (Section 2.4.4) which we consider outside of the scope of our system. For this purpose, we use the UML notation related to messages by presence of events, especially, the lost message and found message [144]. The lost messages are messages ending with a small black circle and represent messages that are sent to an unknown

object and can be interpreted as it has not reached its destination. The found messages are messages starting with a small black circle to specify that the messages have been received by an unknown object. It can be interpreted as the origin of the message is outside the scope of the current description.

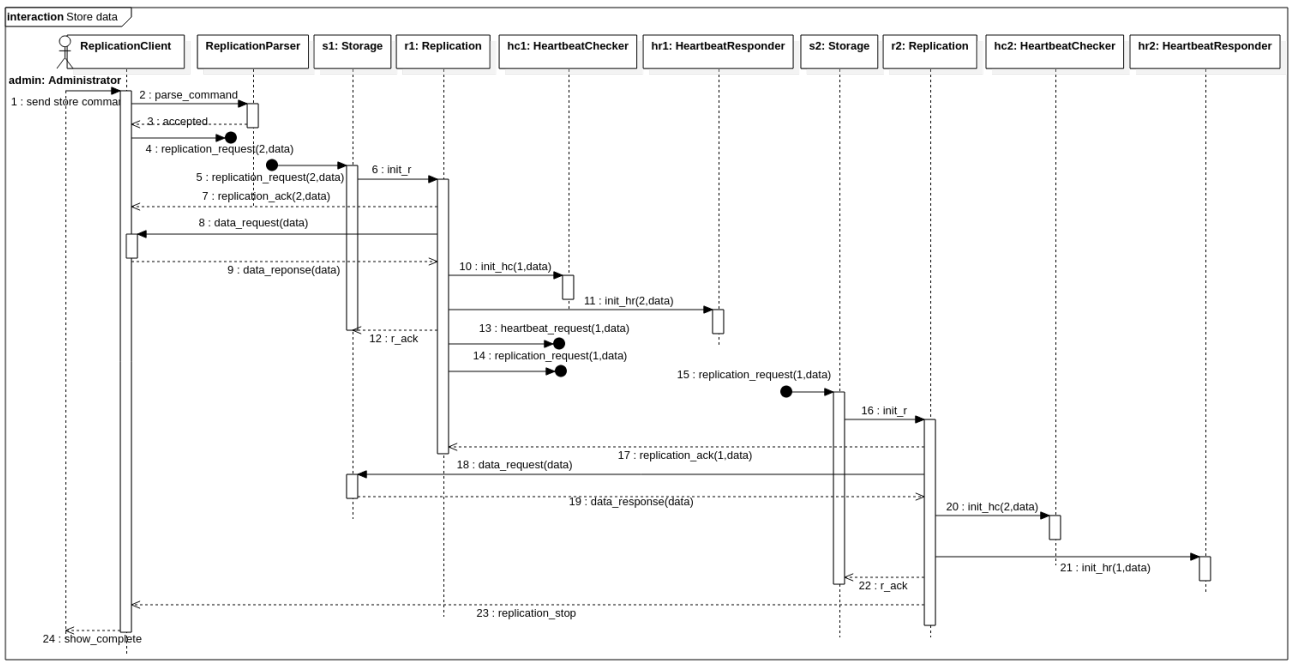

Figure 5.3: Store data sequence diagram

Store data

This scenario describes the Administrator exchanges with the system to store data on the DFS (Figure 5.3). As a precondition, the system has enough Storage nodes to replicate the data (the number of storage nodes is greater or equal to the number of replicas the administrator is looking for). As a post-condition, the data are stored on the DFS and the requested number of replicas is met.

- *Normal Course of Events*

1. An administrator sends a command to store data on the DFS
2. The ReplicationClient validates the command through the ReplicationParser.
3. The ReplicationClient sends a replication request *Interest* on the network
4. A storage (s1) receives the replication request *Interest*, sends an acknowledgment to the ReplicationClient, performs the replication by sending an *Interest* to retrieve a copy of the data and looks for the next replica by sending a new replication request (Figure 5.3).

5. Another storage (s2) receives the replication request *Interest*, sends an acknowledgment to the storage which had initiated the replication request, performs the replication by sending an *Interest* to retrieve a copy of the data. It then sends a message to the ReplicationClient to complete the replication process.

6. The administrator is then notified about the completion of the process.

- *Exceptions*

4.a No Storage node is available.

4.b After a timeout, an error *Interest* is sent to the ReplicationClient.

5.a The number of storages is less than the number of replicas requested by the Administrator.

5.b An error *Interest* is sent to the ReplicationClient.

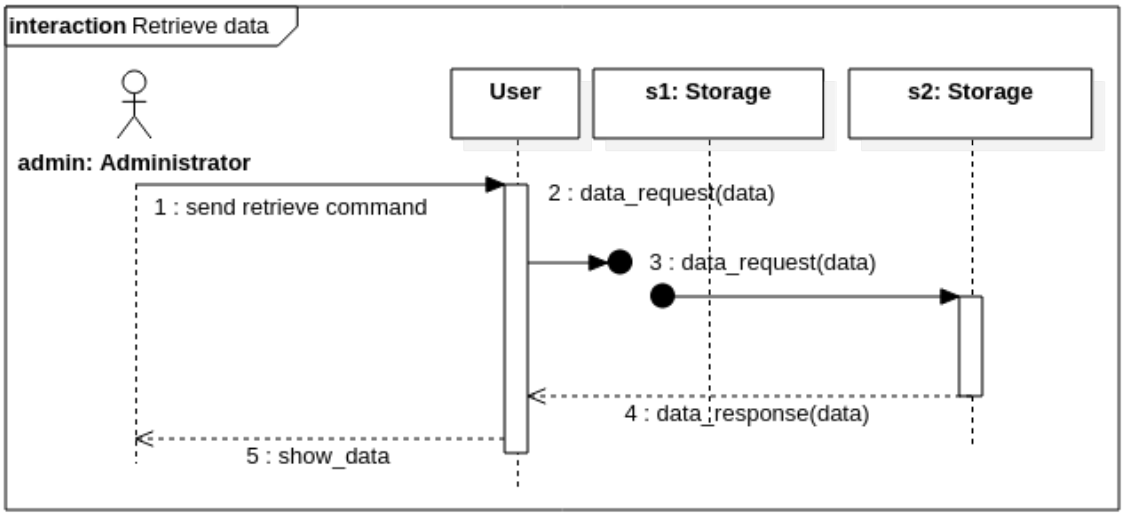

Figure 5.4: Retrieve data sequence diagram

Retrieve data

This scenario describes the Administrator exchanges with the system to retrieve data from the DFS (Figure 5.4). As a precondition, the data have been stored on the system before, and the administrator knows the name of the data he wants to retrieve. As a post-condition, the stored data are returned to the administrator.

- *Normal Course of Events*

1. An administrator sends a command to retrieve data on the DFS.

2. The User application sends a data request *Interest* on the network.

3. A storage receives the data request *Interest*, then sends the data to the User application which displays them to the administrator.

- *Alternative Courses*

2.a An intermediate node has the data available in cache. The data request *Interest* is not forwarded to a storage node holding a replica of the data. The data are then sent from this node to the User application.

- *Exceptions*

2.a No data are available for the specified name. An informative message is sent to the administrator.

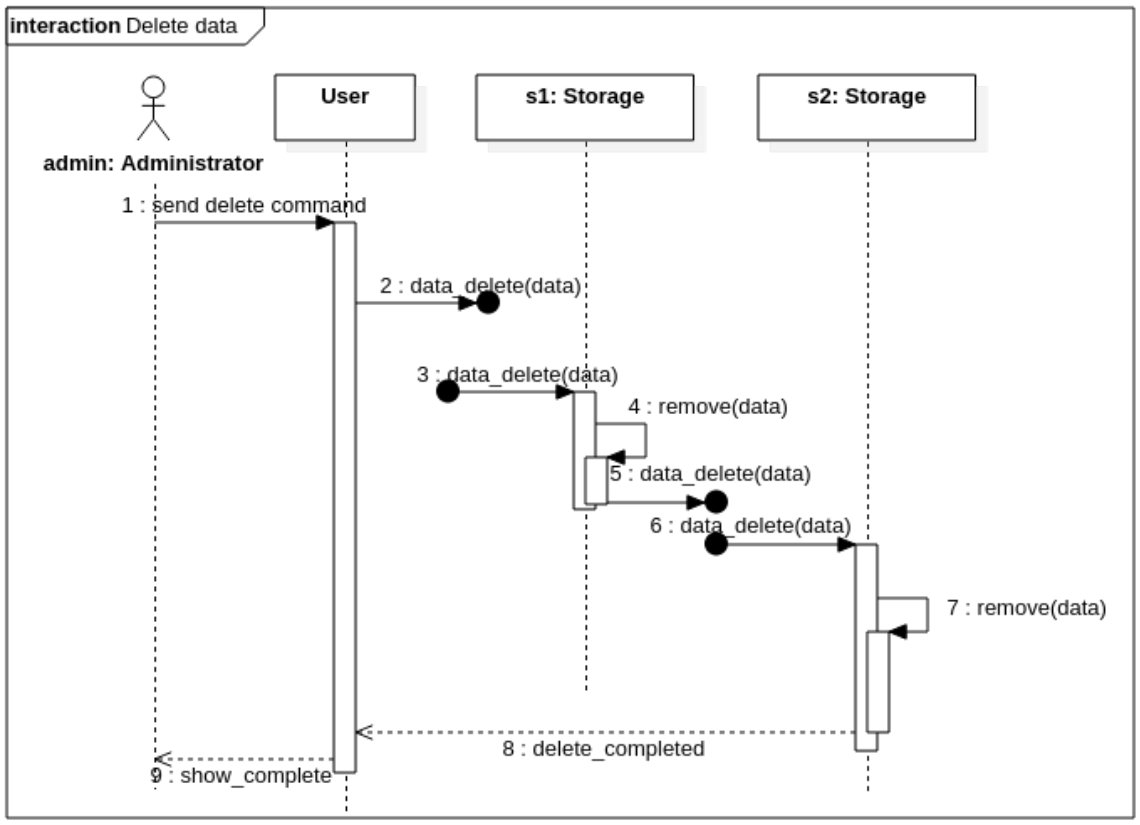

Figure 5.5: Delete data sequence diagram

Delete data

This scenario describes the Administrator exchanges with the system to remove data from the DFS (Figure 5.5). As a precondition, the data have been stored on the system before, and the

administrator knows the name of the data he wants to retrieve. As a post-condition, the list of all data is returned to the administrator.

- *Normal Course of Events*

1. An administrator sends a command to delete data on the DFS.
2. The User application sends a data removal *Interest* on the network.
3. A storage replicating that data, receives the data removal *Interest*, performs the removal and then sends a data removal *Interest* for the next replica.
4. Another storage node replicating the data, receives the *Interest*, performs the removal and then notifies the User application as it was the last replica.

- *Exceptions*

3.a No data are available for the specified name. An informative message is sent to the administrator.

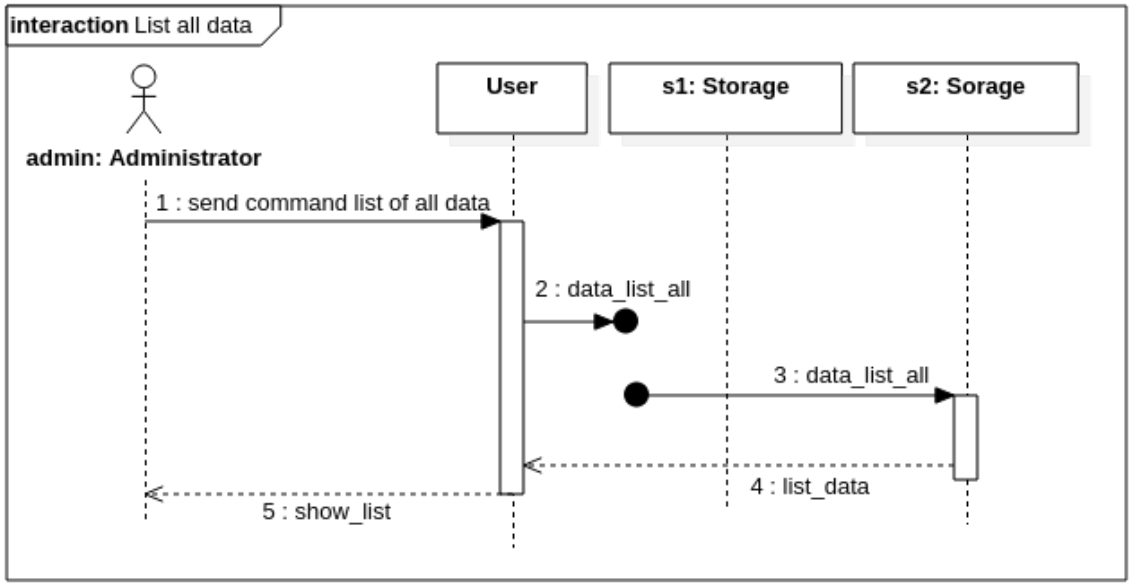

Figure 5.6: List all data sequence diagram

List all data

This scenario describes the Administrator exchanges with the system to know the list of all data that are stored on the DFS (Figure 5.6). As a post-condition, the list of all data is returned to the administrator.

- *Normal Course of Events*

1. An administrator sends a command to list all the data available on the DFS.
2. The User application sends a data list *Interest* on the network.
3. A storage receives the data list *Interest*, then sends the list to the User application which displays it to the administrator.

- *Alternative Courses*

3.a No data have been stored on the DFS before. An empty list is sent to the administrator.

Perform computation

This scenario describes the Administrator exchanges with the system to request a data computation (Figure 5.7). As a precondition, the data have been stored on the system before and also, the computation script is available on the network. As a post-condition, the result of the computation is returned to the administrator. The behavior of the Computes is quite identical. We only present two Computes on the sequence diagram to keep it readable, knowing that more Computes are involved. Maps start differently than Hadoop. A Compute node, when receiving a computation request, starts another map if needed, by sending a new computation request (Section 3.2).

- *Normal Course of Events*

1. An administrator sends a command to perform a computation on data stored on the DFS.
2. The ComputationClient validates the command through the ComputationParser.
3. The ComputationClient sends a computation request *Interest* on the network
4. A Compute (c1) receives the computation request *Interest*, in parallel, he sends an acknowledgment to the ComputationClient, performs the map computation for the first segment, and looks for another compute to perform the map for the next segment. These operations are repeated at each node receiving a computation request.
5. Another Compute (c2) receives the computation request *Interest*, sends an acknowledgment to the compute (c1) which had initiated the computation request, performs the map for the given segment.
6. One Compute object (c1) starts the reduce phase and sends *Interest* to retrieve the result for the different maps.
7. The compute object having performed the reduce computation sends the result back.

Alternative Courses

4.a The compute object doesn't have the data. It uses the "retrieve data" use case (Figure 5.4) to get a copy of the data.

4.b The compute object doesn't have the script for the computation. It sends an *Interest* to retrieve the script.

- *Exceptions*

4.a No Compute node is available.

4.b After a timeout, an error *Interest* is sent to the ComputationClient.

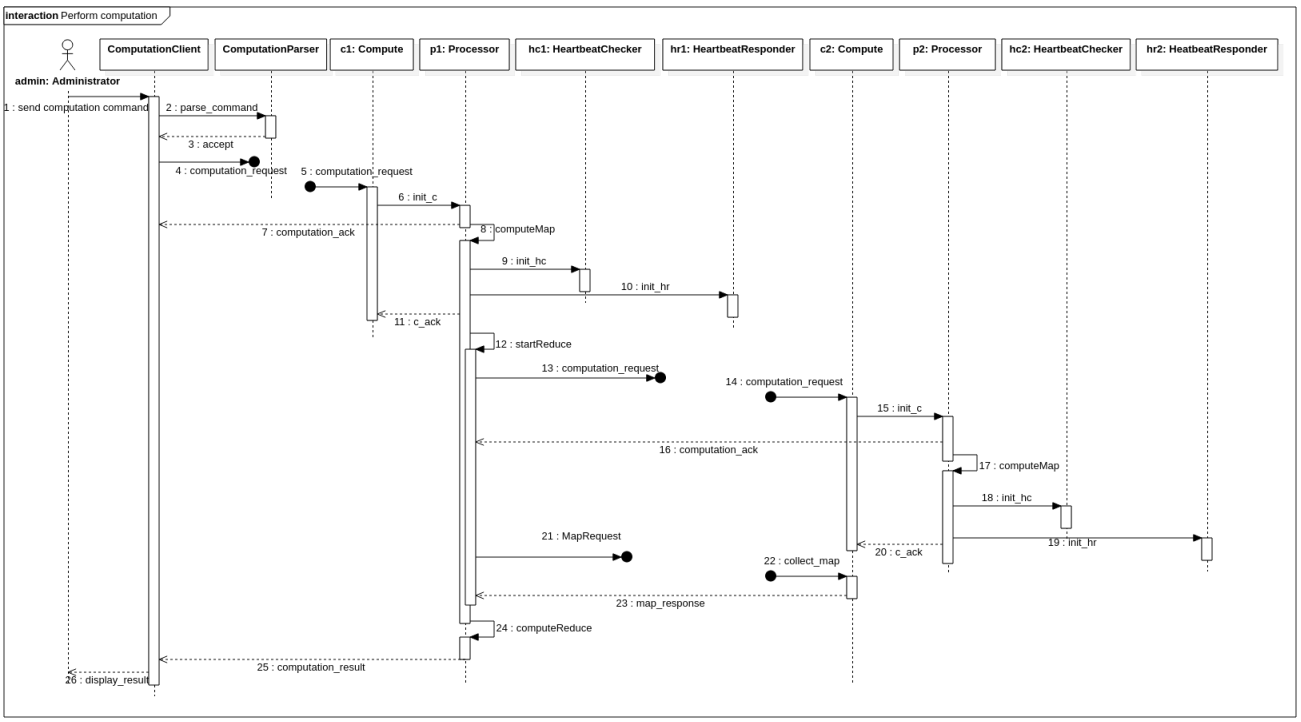

Figure 5.7: Perform computation sequence diagram

5.2 Implementation

The implementation is an important phase in the design of a software system. This part describes the implementation of the different versions of our system. We consider the different scenarios presented in Section 5.1.3. The prototype version is implemented to run within the ndnSIM simulator and evaluate the network communication aspects, while the concrete version is for the logic evaluation. Similarities for both versions are described in Section 5.2.1. Section 5.2.2 provides specificities about the prototype implementation. Technical details about our concrete implementation and a Command Line Interface (CLI) are provided in Section 5.2.3.

5.2.1 Model based approach

Our system has been modeled with timed automata, and properties about the system have been verified using UPPAAL model checker. The verified properties should be preserved during the implementation phase.

Automatic code generation is a technique that can be used in the implementation phase. It consists of using models of the system obtained during the specification, generates source code. The advantage of this technique is when the source code generator is formally proved, it ensures that the generated source code preserves properties verified during the modeling phase.

We were able to partially use the code generation approach for our implementations. When specifying our parsers (Section 3.5.3), the Coq specification has been used to generate the source code for these parsers in OCaml[2] which are then integrated with the other components of the system.

Even though this approach is interesting, we didn't have the chance to use it for our models from UPPAAL since this approach requires a tool to support the source code generation. In our case, a tool that would take as input a UPPAAL model and outputs a source code in our target languages (C++ and JavaScript). Moreover, that tool should be proven.

In the literature, this problem has been considered before. This is the case for Sidra Sultana, and Fahim Arif [145], which proposed automation of UPPAAL automaton into C++ code. Unfortunately, no source code has been provided to the community and our efforts to reach the authors remained unsuccessful. Another solution is found in [146], where a tool is presented to generates embedded C code from UPPAAL models. The target language of this tool is different from ours. Using this tool would require too much work for performing code refactoring to adapt the output code to our target language. For this reason we didn't use this tool for our implementation.

Not having found a tool which satisfy our needs, we could build a tool for this purpose, and prove that the code generator preserves properties when generating code. But, it would be very time consuming while we are lacking time. We decide to perform the implementation manually instead of investing our time into the building of a source code generator.

In our approach, we consider each automaton as a software component as described in Section 5.1.2 and shown in Figure 5.2. Based on the automaton definitions, we implement the logic behind each component.

Introduced in 1994 by the Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) [147], design patterns are generic design solutions responding to specific software development problems. The state design pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes, performs treatments based on the current state and looks as if the class of the object has changed. The state design pattern is used

[2]https://github.com/mistersound/thesis-source-code/tree/master/system/parser

where a state transitions diagram or state machines is possible. That is the case for our system modeled as a set of automata (Section 4.3).

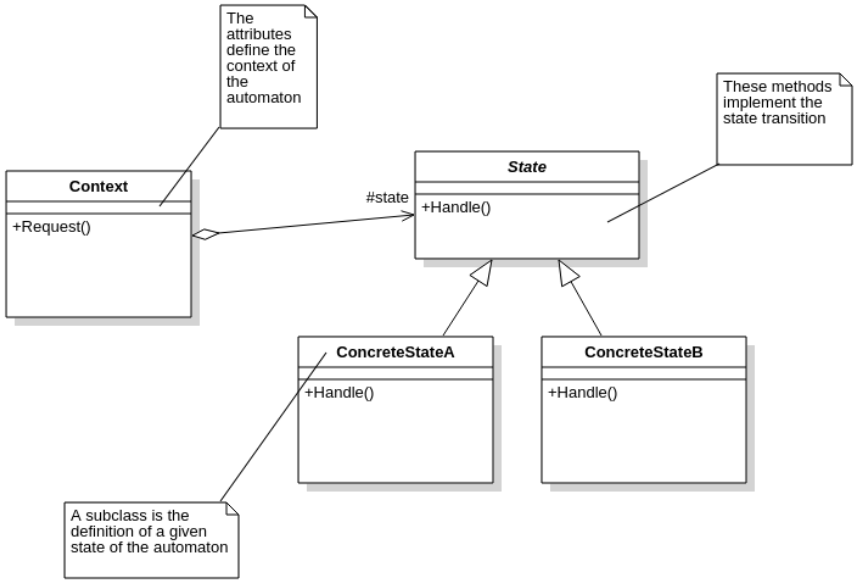

Figure 5.8: State design pattern structure

Figure 5.8 presents the structure of this design pattern:

- Create state classes that declare a common interface.
- Delegate operations depending on the states of the Context object to its current state object.
- Ensure that the Context object points to a state object that reflects its current state.

Figure 5.9 presents the use of the design pattern for the building of the Storage component, based on the Storage model (Section 4.3.1):

- Context (Storage) is a class which allows to use a state object and which manages an instance of an object ConcreteState.
- State (StateStorage) defines an interface that encapsulates the behavior associated with a particular state of Context.
- ConcreteState (Idle, Start, Ready, Replicating, ReplicationComplete, SendData) implements a behavior associated with the state of Context.

The different automata communicate using communication channels (Section 4.2.2). This ensures synchronization between them. For our implementation, a communication channel between two automata gives place to an interface between the corresponding components (Figure 5.2).

For instantaneous states also called committed states (Section 4.2.2), for which no delay is allowed, we have an immediate movement of the component concerned. Thus as soon as the

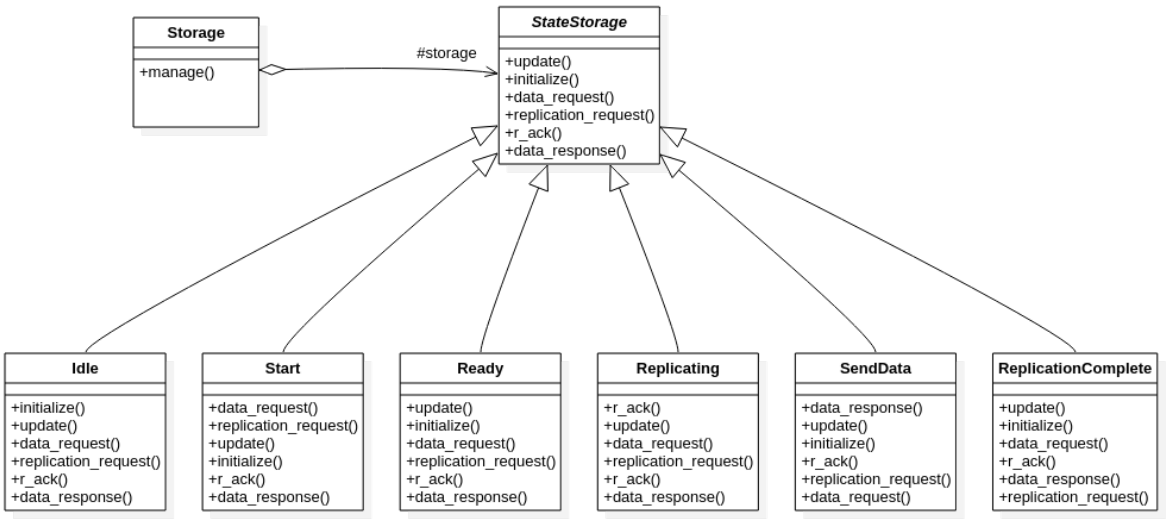

Figure 5.9: Storage using state design pattern structure

system reaches a committed state, the operations are performed and an update is automatically called to move to the next state.

Finally, the adoption of a structured coding approach helps to preserve properties from the specification.

5.2.2 Prototype version

Early in our project, we were looking for a way to quickly test our approach and perform an evaluation (Section 5.3). ndnSIM has been built for this purpose by the NDN community (Section 2.4.8). It helps to build test, and evaluate applications using the NDN architecture logics. The advantage of using ndnSIM is that the whole NDN stack and the core NDN protocol interactions such as receiving *Interest* and *Data* packets from upper and lower layers through Faces are already integrated as shown in Figure 5.10.

A prototype often implies a reduction of the specification. This is the case for our prototype where we only implemented the logic related to the different network messages exchanged by each component based on our approach defined in Section 5.2.1, and using the template[3] provided for building application using ndnSIM. The implementation language of ndnSIM is C++, and the template provided is in C++. Our prototype[4] has then been built in C++. Our prototype has been used for the simulation experiment described in Section 5.3, but also for the use cases in Section 6.3.

5.2.3 Concrete version

Later, to run our solution on physical devices and also being able to compare our approach to an existing solution, we decide to implement a concrete version (concrete because only this implemented version can be run on physical device) of our approach for this purpose. The

[3]https://github.com/named-data-ndnSIM/scenario-template
[4]https://github.com/mistersound/simulation/tree/master/ndnSIM/ns-3/src/ndnSIM/apps

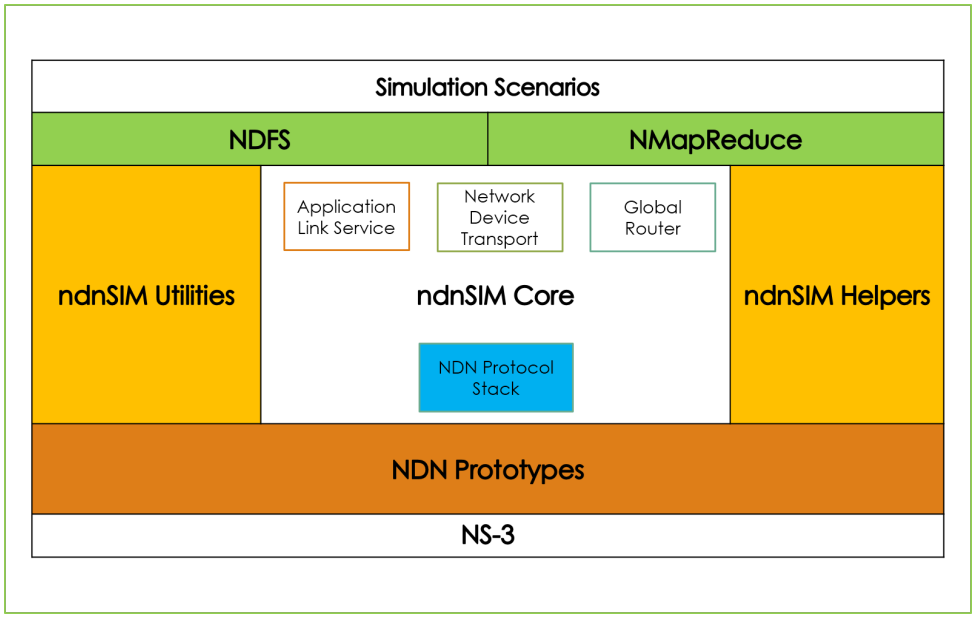

Figure 5.10: ndnSIM simulation package structure

implementation is based on the approach described in Section 5.2.1 and here, the main difference with the prototype version is the implementation of the business logic related to the replication, the computation, and the fact that it is built to run on a physical device. The NDN community has developed a library in JavaScript to ease the integration and the use of the NDN protocol by applications.

Node.js[5] is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser.

Using Node.js and the JavaScript library from NDN, we have built a concrete version[6] of the system. This version has been used for the system evaluation and to compare our approach with the Hadoop platform (Section 6.2).

Commands

This section presents the different commands that have been developed for our system. They have been developed based on the requirements in Section 5.1.1. The commands of the CLI are mainly used by the administrator to interact with the system.

- To store data:

```
$ node ReplicationClient.js /domain/storage/ReplicationFactor
/ReplicationIndex/domain/dataName
```

- To retrieve data:

```
$ node ndfs.js get /domain/dataName
```

[5]https://nodejs.org/en/docs/
[6]https://github.com/mistersound/thesis-source-code/tree/master/system

- To delete data:

```
$ node ndfs.js remove /domain/dataName
```

- To get the list of all data:

```
$ node ndfs.js get /all
```

- To request computation on data:

```
$ node ComputationClient.js /domain/compute/domain/data/domain/sourceCode
```

5.3 Simulation

This section describes the experiment used to evaluate the architecture in a simulated environment using the prototype version (Section 5.2.2). Moreover, the methodology used is presented. The goal is to determine and gain insight into the important parameters that impact our system and potentially affect its performance. This helps to better understand and characterize our system, but also better optimize it. Performance optimization is obtained by choosing the optimal values for the important parameters affecting our system.

5.3.1 Tools

Our experiment was conducted in several steps. The first step was the development of our prototype in C++, which is the language used by the simulator. We then designed the experiments to be run. The next step consisted in the analysis of the data collected during the executions. We present the simulation and the analysis tool used during this experiment.

Simulation tool

In the network domain, it is costly to deploy a complete testbed containing multiple nodes, routers, and data links to check a network algorithm. A network simulator such as ndnSIM [85] affords to test a network algorithm with various network topologies under a controlled environment. Nowadays, the list of ndnSIM related papers on simulations and measurements is impressive. ndnSIM implements the NDN protocol stack, to run simulations for a variety of network topologies and scenarios (Section 2.4.8). We use it to simulate the behavior of our architecture on different kind of topologies and provides tracers using libraries and packages to collect information such as in table 5.1. We have implemented a prototype version [148] of our architecture to run on ndnSIM.

Dataplot

Dataplot [149], is a statistical analysis tool developed and maintained by the National Institute of Standard and Technology (NIST). It is a powerful tool for data analysis which supports both quantitative and graphical statistical method. We have been introduced to dataplot during our visit as guest researcher at NIST. This is one of the reasons why we have chosen this tool for our analysis, but also because of its capability when dealing with sensitivity analysis.

Type of measurement	Description
InInterests	measurements of incoming Interests
OutInterests	measurements of outgoing Interests
InData	measurements of incoming Data
OutData	measurements of outgoing Data
SatisfiedInterests	measurements of satisfied Interests
TimedOutInterests	measurements of timed out Interests
InSatisfiedInterests	measurements of incoming satisfied Interests
InTimedOutInterests	measurements of incoming timed out Interests
OutSatisfiedInterests	measurements of outgoing satisfied Interests
OutTimedOutInterests	measurements of outgoing satisfied Interests

Table 5.1: ndnSIM Metrics

Factor	Name	Low (-1)	High (+1)
X1	Network Size	64	100
X2	Number of Storage Nodes	20	40
X3	Size of File	5 Gb	10 Gb
X4	Number of Replication	3	5
X5	Content Store Size	10 packets	1000 packets
X6	Node Capacity	2 Tb	5 Tb
X7	MeanFailureTime	300 s	700 s
X8	MeanFailureDuration	30 s	100 s
X9	Number Of Producer (Admin)	1	3
X10	Number Of Consumer Users	5	10
X11	Number Of Failure Nodes	1	4
X12	Links Speed	10 Mbps	1000 Mbps
X13	Links Delay	1 ms	10 ms
X14	Cache Policy	Lru	Fifo
X15	Consumer Request Distribution	Uniform	Exponential

Table 5.2: Input Parameters and value simulated

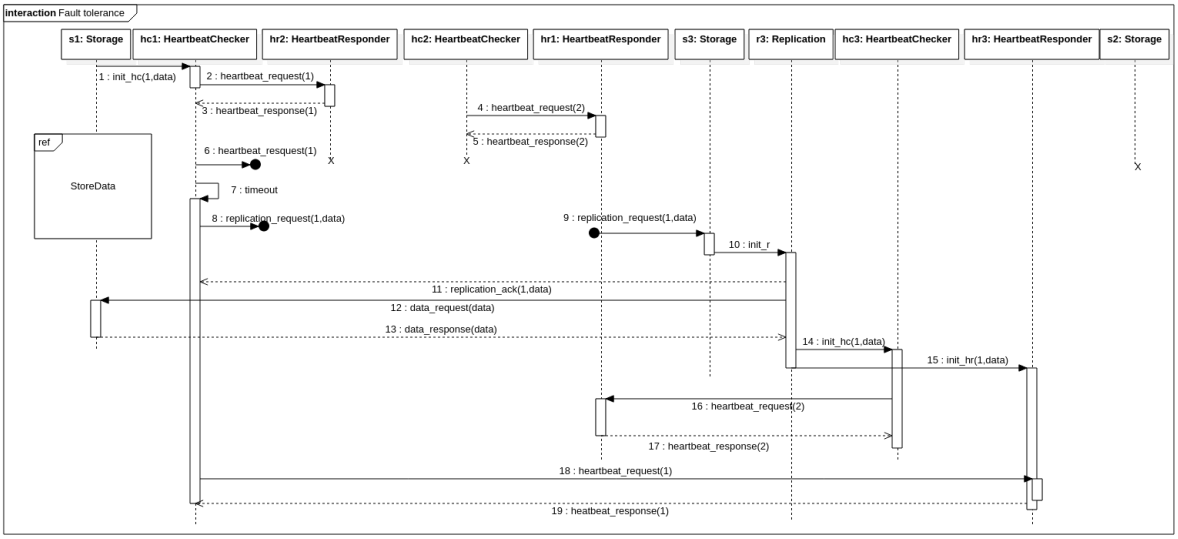

Figure 5.11: Fault tolerance sequence diagram

5.3.2 Experiment

The network topology used for this experiment is a grid topology that has to be implemented programmatically. The experiment consisted of using a uniform distribution, select some nodes on the network on which to start a set of ReplicationClients, Storage Nodes, and User application (Figure 5.3). The User application is used to play the role of an application needing the data from the DFS, such as NMapReduce application. The clients send replication requests to replicate data in the DFS which will be satisfied by the Storage nodes. Also, User applications are used to retrieve data from NDFS. The data retrieval is based on uniform distribution and an exponential distribution (factor X15 from Table 5.2). Also, we randomly selected some nodes to fail during the simulation. This was to test the fault tolerance of the system. Figure 5.11 presents a scenario with 3 storage nodes. Storage s1 and s2 are replicating the same data, and check each other periodically using they HeartbeatChecker and HeartbeatResponder through the heartbeat mechanism (Section 3.1.3) which has been initiated during the store data scenario of an admin (Figure 5.3). When storage s2 fails (failure of hr2 and hc2), it is detected by the HeartbeatChecker of storage s1 (hc1) which initiates a replication request to find a storage to take over the data replicated by s2. Storage s3 responds to the replication request, replicates the data and starts a heartbeat mechanism (hc3 and hr3) to check storage s1 and responds to heartbeat requests.

To evaluate the performance of our proposed DFS, we identified 15 input parameters (Table 5.2) and 10 response parameters (Table 5.3) to run our simulation experiments. These parameters and system responses have been identified by analyzing the available parameters in ndnSIM. This is to analyze the impact of these input parameters on the system. For each input parameter, we chose only two values: "low" and "high" [150]. Having 15 variables with two possible values implies 2^{15} possible combinations to run using a full factorial design (which consists of running all the possible combinations). Using a full factorial design will give more precise information. But this is difficult to run due to the time needed for a simulation to complete which requires approximately 40 minutes per execution. To reduce the number of runs, we used the method of

the fractional factorial [150] with a 2^{15-8} design (design specification). This consisted of using a part of the runs needed for a full factorial and still having good results. The 128 runs were chosen carefully to be representative of the whole possibilities. We then measured 10 system responses (Table 5.3) to capture the behavior of the DFS. After the execution of the simulations, the metrics are produced in files (3 files) on which R scripts (Appendix E) are applied to compute the response parameters. Finally, the results are analyzed using the statistical tool dataplot [149]. Routing configuration is a crucial aspect of these experiments. As the NLSR (Section 2.4.4) is not integrated into ndnSIM, the GlobalRoutingHelper [85] (a ndnSIM integrated function used to perform the routing) is used to calculate the route weight. Each time that the system creates a new route to the local application, a new origin route is added to the GlobalRoutingHelper and all the route weights are recalculated. At the beginning of the experiment, all the FIBs are empty. They are populated on-the-go by the ReplicationClient and the Storage applications. As the existing routing strategies are not suitable for our system because of the replication constraints, a custom forwarding strategy is installed on each node for the storage prefix. The other prefixes on the FIB are handled by the Best Route Strategy [91] that is used by default for the data prefixes (Section 2.4.4).

Response	**Name**	**Description**
Y1	Replication Time	Time between a storage Interest and the completion of the last replication
Y2	Mean Distance to data	Mean hop count before an Interest reaches a storage node
Y3	Mean Retrieval Time	Average time used by a user to get the data.
Y4	Total Replication request	Total number of replication requests
Y5	Number Of Exchanged Packets	Total of exchanged packets over the network (number of packets/s)
Y6	Data Rate	(Kilobits/s)
Y7	Number Of Incoming Interests	Total of Incoming Interest over the network
Y8	Number Of Outgoing Interests	Total of Outgoing Interest over the network
Y9	Number Of Outgoing Data	Total of Outgoing Data over the network
Y10	Number Of Incoming Data	Total of Incoming Data over the network

Table 5.3: System Responses

5.3.3 Results and discussion

This section describes the experimental results obtained using ndnSIM to measure the performance of the system.

A sensitivity analysis has been adopted to evaluate the behavior of our model. Performing this analysis permits us to determine for each system response, the factors which are the most important for that system response and the best and worst settings for these factors (main effect). We present the results for the replication time, the Mean distance to data and the Mean Retrieval time which have a great interest in our study, as their values indicated how the DFS performs. As soon as data are replicated, it can be involved in a computation process, also when an application wants to access data, one wants this access to be fast.

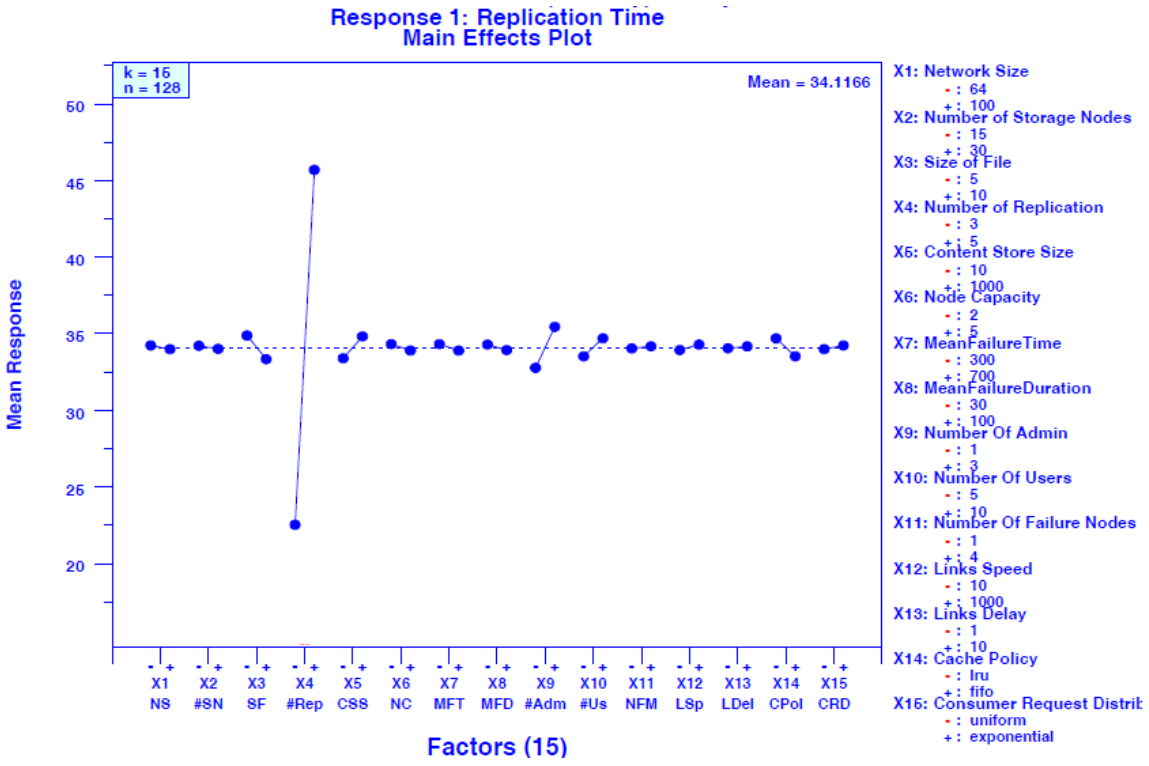

Figure 5.12: Main Effect plot - Mean Replication Time

Each factor was set at its high value for 64 runs and its low value for 64 runs within the 128-run design. The main effects plot graphed the average of these two sets. Figures give the main effect plot for the replication time, the mean distance to data and the mean retrieval time. The main effect plots have been produced using dataplot [149].

The mean plot is formed with:

- Horizontal Axis: we have the 15 factors and the two settings ("-" and "+") within each factor.
- Vertical Axis: we have the mean response for a given setting ("-" or "+") of a factor, for each of the 15 factors.

It is a sequence of 15 mean plots, with one mean plot for each factor. All of the mean plots are on the same scale to permit comparison and relative importance. The vertical axis of each mean plot is the mean response for each setting of the factor and the horizontal axis is the two settings of the factor: "-" and "+" (-1 and +1). A huge difference in the two means (for the two settings) implies that the factor is important while a small difference implies that the factor is not important. For each of the 15 factors, the mean values for that factor are connected with a line. The magnitude of that line indicates the factor effect. The longer line means that the factor has effects while the shorter line indicates the factor has not. The slope of the line shows whether there is an increasing or decreasing effect of the factor on the responses.

Figure 5.12 shows that the replication time (response Y1) is mainly affected by the number of replication (factor X4). Its mean value was 34.11s during our experiment. The time needed to perform the replication changes only with the number of replication. All the other factors don't

impact the time needed to replicate the data. This result contradicts our initial expectation, which was that the replication time would be influenced by the size of the network, the number of storage, the content store (cache) size or at least the size of the file to replicate. This result is important as it shows that the system can highly scale with no impact on the time needed to replicate data on the network. This also shows that the communication properties (Section 4.4.1) hold, as the replications are performed, meaning that the different nodes in charge of the request are exchanging data to perform the request.

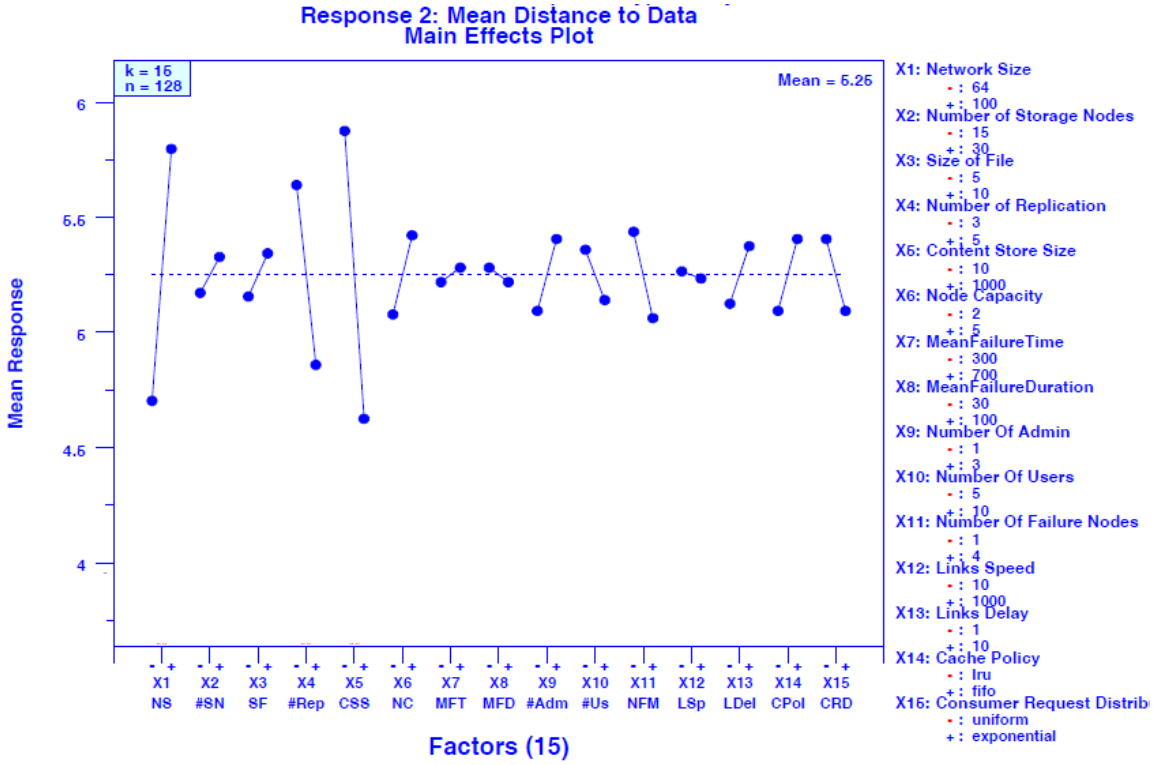

Figure 5.13: Main Effect plot - Mean distance to data

For the response Y2 (mean distance to data), we consider here how close are nodes running an application on the cluster to a replica. Figure 5.13 shows that factors X5, X1 and X4 are the most important. For the best settings X5 with a high value, X1 having low value and X4 a high value. Its mean value was 5 nodes, during our experiment. The size of the content store, the size of the network, and the number of replicas have an impact on the number of nodes that an *Interest* has to reach before this *Interest* has been satisfied. Having a small value for the mean distance to data means that the applications will quickly have access to the data. The size of the content store, the network size and the number of replicas might be chosen appropriately.

In Figure 5.14, we can see that Response Y3 (Mean Retrieval time) is mainly affected by Factors X13, X10, and X1. Its value was around 64 ms during our experiment. The size of the network, the delays on the links and the number of Users (applications accessing the data in the DFS) are affecting the time needed for the application to retrieve data from the DFS.

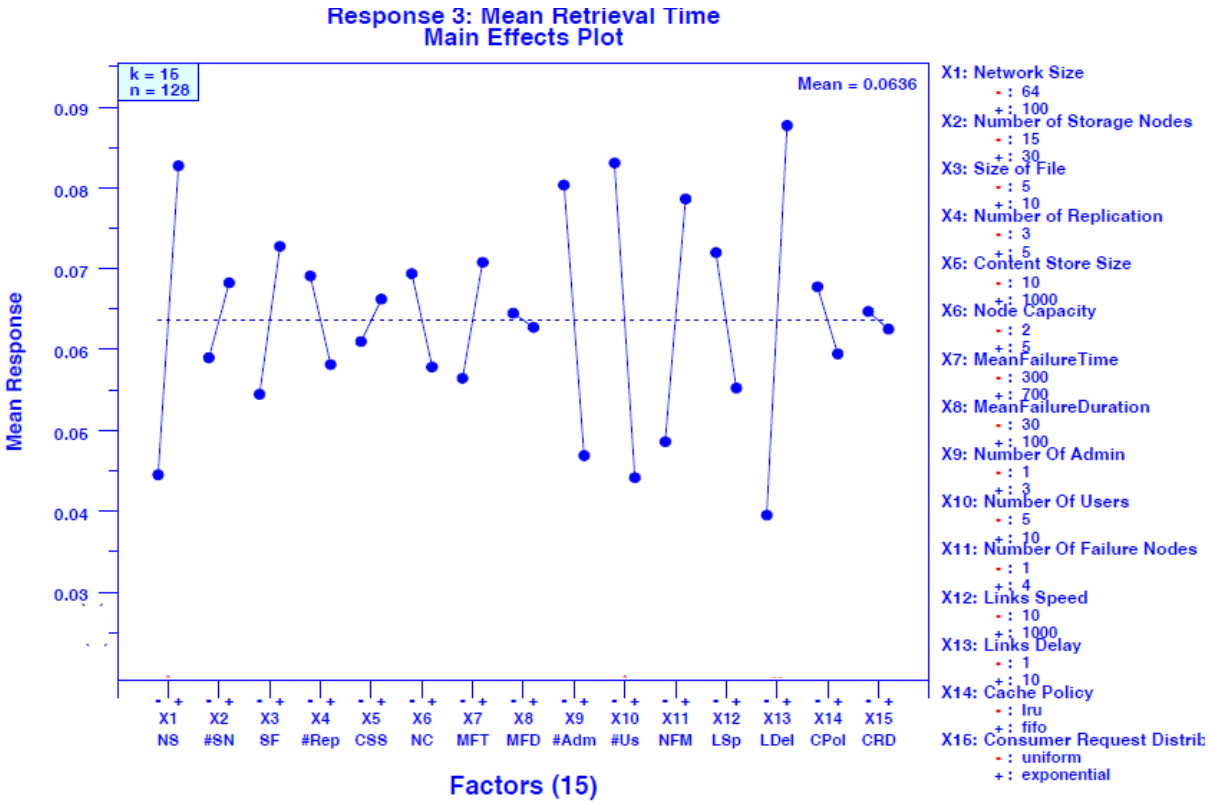

Figure 5.14: Main Effect plot - Mean Retrieval Time

The two first results show that the system behaves as expected (Chapter 3) and the last one shows how scalable the system can be (these satisfy our non-functional requirement related to the system scalability in Section 5.1.1). In fact, increasing the number of users (Factor X10) improves data retrieval time. Also, experiments show that having only one replica available in NDFS, the system can move to a stable state if there are sufficient Storage nodes to replace the failed ones. This shows that our recovery properties (Section 4.4.3), related to the fact that failed nodes are detected and their processing are managed by other nodes, hold.

5.4 Summary

In this chapter, we presented the software architecture that we have proposed for our solution in Chapter 3. We presented the different requirements the system must meet. Based on the modeling performed in Chapter 4, we described the different components of the system. We have then described our approach for the implementation in general and details about a prototype and a concrete implementation. We have also considered a simulation of the prototype version of our system and performed a sensitivity analysis to study the different factors impacting the system. We found that the system is highly scalable with no impact on the replication time. We have also shown that properties from Section 4.4 hold. This shows that the properties are preserved with our implementations, and our approach of implementation despite its simple aspect is successful.

In the next chapter, we use the concrete implementation for an evaluation phase. We then compare our solution to existing solutions from IP using classical metrics used to compare DFS

and computation distribution.

Chapter 6

Experimentation and Results

Contents

In Chapter 5, we have implemented our architecture for Big Data over NDN. Our architecture is composed by a Distributed File System (Section 3.1) and a Computation distribution (Section 3.2). In this chapter, we perform the evaluation of our architecture. We analyze different statistics by running experiments on our architecture and comparing the results with experiments from an Hadoop architecture. In Section 6.1, we first describe the experimental environment, then Section 6.2 gives the evaluation and comparison. Finally, we also consider in Section 6.3 three use cases where our architecture have been used to consider real life problem.

6.1 Experimental Platform

As we are considering the evaluation of our architecture, which is compared to another architecture, we have built two identical experimental platforms (partially presented in Figure 6.1). One for our architecture based on NDN and the second one for the Hadoop architecture based on TCP/IP. Our experimental platforms have been built using a cloud platform available in our laboratory.

6.1.1 NDN experimental platform

NDN possesses a testbed for applications evaluation and testing with real settings (Section 2.4.6). Unfortunately, our solution is difficult to test on the testbed because of the software stack that

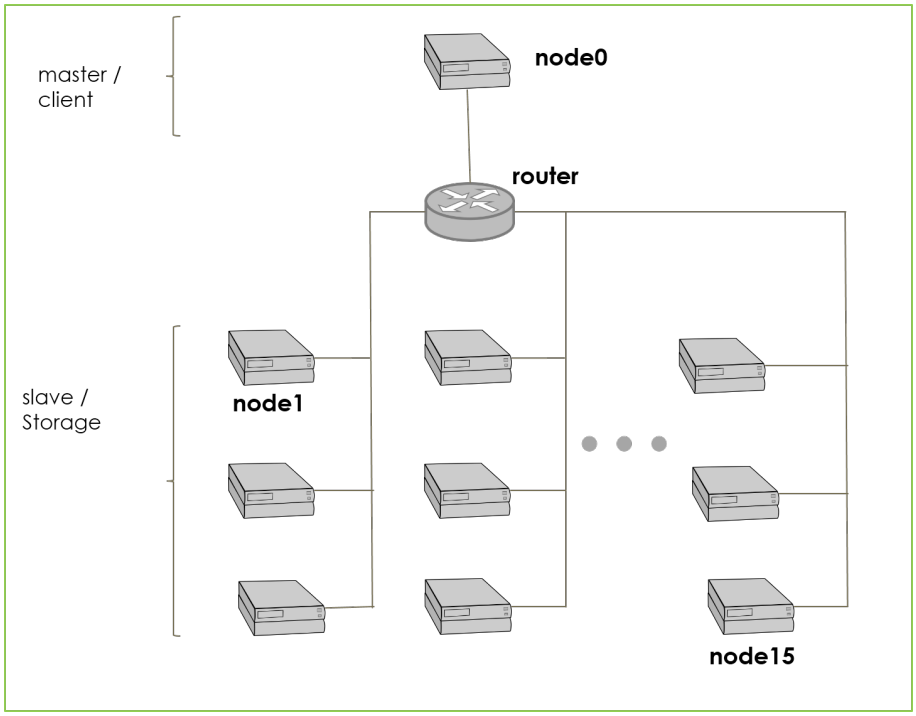

Figure 6.1: Cluster Architecture

has to be installed on the nodes. The management of a node in the testbed is operated by the organization which holds that particular node, so we only have administrative power on the LACL node. To get our environment as close as possible with what exist on the NDN testbed, we have adopted the configuration settings used for the nodes on the testbed. We have then built a mini testbed using a quarter of the number of nodes used in the current testbed. This mini testbed is used only as a test environment and has no connection with our link in the NDN testbed nor the NDN testbed itself. It is also good to mention that the number of nodes has also been chosen to have a meaning when building a Big Data cluster as recommended in best practices for building Big Data clusters [151].

The experimental cluster consisted of 16 nodes which are interconnected using a node called router. One node was used as a client and the 15 others were used as Storage and Compute nodes. The hardware information of the nodes is as follows:

- 15 nodes of storage / compute (receive replication, perform computation) with 8Go RAM, 50Go storage and 1 CPU
- 1 client (initiates replication, initiates computation) with 16Go RAM, 50Go storage and 2 CPU

At the software level, we have the following:

- OS is Ubuntu 16.04
- ndn-cxx has been installed [143]
- NFD has been installed [152]
- ndn-tools has been installed (essential command-line tools) [153]

- NLSR (NDN Link State Routing Protocol) has been installed [154] (it requires ChronoSync [155] and PSync [156])
- Nodejs has been installed [157].

Finally the JavaScript version of our Big Data framework was installed, that requires Nodejs (version >= 10.15.3). Both the client and the 15 nodes have the same software stack, except for our framework. The ReplicationClient and the ComputationClient were installed on the client while the master versions (Storage and Compute) were installed on the other nodes. The generated parsers (Section 3.5) were integrated into the deployed solution (Section 5.1.2) and were giving satisfactory results.

In order to allow researchers working on NDN to reproduce our work easily, and also because we have to repeat the experiments many times, we have built some scripts in order to automatize the different tasks for the cluster deployment, specially the installation of the software stack. The different scripts can be found in Appendix B, and the configuration script for the nodes on github (`https://github.com/mistersound/NDN-1/tree/master/configuration`).

6.1.2 Hadoop experimental platform

Our goal is to be able to compare our solution with the one from Hadoop. For this reason, we have to design the Hadoop experimental platform similar to the one we built for our framework. As previously, the experimental cluster consists of 16 nodes interconnected with a node called router. We consider a simple architecture with only one node as NameNode (master) and the other 15 nodes are designed to be DataNodes (slaves). We don't consider any backup, nor secondary NameNode. The master node manages the cluster and typically runs master components of distributed applications. Slave nodes coordinate data storage as part of the Hadoop Distributed File System (HDFS). The hardware information of the nodes is as follows:

- DataNodes: 8Go RAM, 50Go storage and 1 CPU
- NameNode: 16Go RAM, 50Go storage and 2 CPU

As mention in the previous section, the configuration is meaningful for a Hadoop cluster deployment [151]. We have used Ubuntu 16.04 as OS, the Hadoop-2.9.1 [158] and Java 7 [159] for HDFS and MapReduce. In this context, we also keep the experiments reproducibility constraint in mind while deploying the cluster. We then take this constraint into consideration by automating the deployment and providing a script for this automation (Appendix B.4). Our platform can easily be rebuilt.

6.2 Evaluation and Comparison

The evaluation is performed in two phases. Section 6.2.1 describes the evaluation of NDFS, which consists in sending replication requests from the ReplicationClient to replicate a file on NDFS. We then measure metrics such as the running time, and the throughput. This evaluation is performed considering the experiment parameters in Table 6.1, while increasing each time the size of the file and modifying the replication factor. Section 6.2.2 is for the evaluation of the NMapReduce. Based on the replication performed in Section 6.2.1 a computation request on the data is sent by the ComputationClient and we measure the runtime. In order to verify the reliability of the results as well as their adequacy to a normal behavior, we compare in each phase our result with the same experiment performed using the Hadoop framework.

Input	Experiment 1	Experiment 2	Experiment 3	Experiment 4
File size	500 Mo	500 Mo	5 Go	5 Go
Replication factor	3	15	3	15

Table 6.1: Evaluation parameters

6.2.1 NDFS vs HDFS

In this section, we present the results of the comparison between NDFS and HDFS for data replication. We present namely the replication time, and the throughput.

The execution of an experiment from Table 6.1 on NDFS consisted in performing a set of actions. On the ReplicationClient, we first load the file to be replicated using ndnputchunks [153].

This step is important as this permits to have the file available in NDN format. Then the command for the replication request is issued:

For Experiment 1 & 3:

```
$ ndnputchunks /lacl/data < file500Mo.txt
$ node ReplicationClient.js /upec/storage/3/3/lacl/data
```

For Experiment 2 & 4:

```
$ ndnputchunks /lacl/data < file5Go.txt
$ node ReplicationClient.js /upec/storage/15/15/lacl/data
```

The ReplicationClient.js corresponds to an implementation of the specification of the ReplicationClient Model (Section 4.3.1).

In order to perform measurements for the runtime, we have modified the default behavior of our application to include an aspect to detect the end of the replication. This aspect consisted for each Storage participating to a replication, to send an Interest to the ReplicationClient as an acknowledgement when they complete the replication (when the last segment of the file to be replicated is received). We then compute the difference between the start of the replication request and the reception of that acknowledgment

(`Runtime =` *last_acknowledgement_time* `-` *start_time*). For the second measurement which is the throughput, we have used an output provided by ndntools which provides metrics on each Storage node.

To perform the same experiment for HDFS, we have performed commands of the form:

```
$ time hdfs dfs -put (fileName)  (destinationHDFS)
```

For example for Experiment 1:

```
$ time hdfs dfs -put /home/hadoop/file300Mo.txt  /usr/jdongo/file300Mo.txt
```

In order to perform the measurements, we used the Linux `time` command. It is used when one wants to determine how long a given command takes to run. In our case, how long does the hadoop command takes to run. Each time, HDFS replication settings are adapted to the replication factor of the experiment.

Each experiment was performed many times to confirm that results are stable over each execution.

Runtime

This measures the time spent for replicating a file on the cluster using the parameters in Table 6.1.

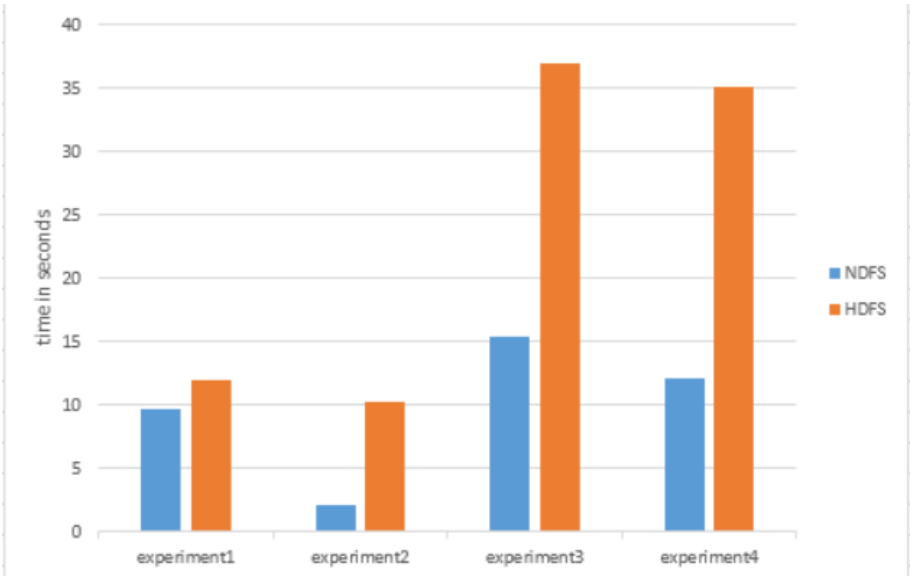

Figure 6.2: HDFS runtime vs NDFS

As we can see (Figure 6.2), for all the experiments, HDFS takes more time to perform the replication. Increasing the replication factor has a little impact on the replication time for HDFS, while it has a huge impact on the replication time for NDFS and decreases it considerably. This can be explained by the use of a cache in NDFS, which enables the retrieval of the data from many sources. We have also noticed that the client used for sending the replication in the NDFS experiments always terminates and the exact number of requested replica is created in the cluster. This confirms that our communication properties (Section 4.4.1) hold (P1, P2, P3) as the replication request is received by the storage nodes and the requested operation is performed. This also confirms that our property P7 which is about the completeness property (Section 4.4.2) of the replication process holds.

Throughput

This measures the amount of data passing through the system. It depends on both parameters (Table 6.1), the file size and the replication factor. For HDFS, increasing the replication factor increases the throughput (Table 6.2), while the opposite effect is observed with NDFS. The increase of the replication factor in NDFS decreases the throughput. One suspect of this behavior is the NFD as presented in [160]. NDFS still presents good throughput compared to HDFS, as for the same file size and same replication factor, the throughput from NDFS is almost two times the one from HDFS.

Having good throughput shows that there is a communication between the ReplicationClient and the different Storage nodes. Data to be replicated are first retrieved from the Replication-Client and then from a Storage, from a cache at an intermediate node or from the Replication-Client. This confirms that our communication properties set forth in Section 4.4.1 hold. This also confirms the holding of our consistency property (Section 3.6.3) about the behavior of our

replication parser which sends a NDN message when a replication command is accepted.

	NDFS	HDFS
experiment1	373,38	64,45
experiment2	316,17	84,1
experiment3	349,46	153,42
experiment4	276,3	162,05

Table 6.2: HDFS Throughput vs HDFS (Mbits/s)

6.2.2 Hadoop MapReduce vs NMapReduce

In this section, we present the results from the comparison between Hadoop MapReduce and NMapReduce while counting the number of times each word appears in a text data file. We present the time used for the computation. The different files replicated on the DFS are text files containing English books. After the replication, we perform two experiments using the result of experiment 1 and 3 from Table 6.1.

For NMapReduce, the experiment consisted in using a ComputationClient to send a computation request for the data. The following command was used:

```
$ time node ComputationClient.js /upec/compute/lacl/data/lacl/WordCount
```

The ComputationClient.js corresponds to an implementation of the specification of the ComputationClient Model (Section 4.3.1).

As mentioned in Section 3.2, the script here is also considered as a data available on the network. The script of the JavaScript program which was applied on the data is available in Appendix C

For the Hadoop MapReduce the following command was used after implementing a Java version of the WordCount program available in Appendix D

```
$ time hadoop jar wordCount.jar org.lacl.jdongo.WordCount /user/jdongo/file300Mo.txt
/user/jdongo/output
```

In order to perform the measurements, we used the Linux `time`. The time command is used when one wants to determine how long a given command takes to run. Here, how long the NMapReduce and MapReduce commands take to run.

Each experiment has been performed many times and the results are stable over the different executions.

Runtime

This is the time spent by the application to perform the computation. The computation has been performed using the parameters in Table 6.1. We only performed two experiments using settings of experiment 1 and 3. Results revealed that the execution time increases with the size of the input file (Table 6.3) in both applications. It can be noted that NMapReduce presented good execution time compared to MapReduce. This can be explained by the fact that NMapReduce somehow takes full advantage of data available in NDN cache. Getting the data quickly from a closed node helps to decrease the overall time, as it reduces the time needed to get the data when they are not available at the processing node. Having these results is a confirmation that

the computation process ended. This shows that our completeness property P8 (Section 4.4.2) related to the fact that the computation process completes, holds.

	NDFS	HDFS
experiment 1	310	380
experiment 2	1535	1850

Table 6.3: Hadoop MapReduce wordcount execution time vs NMapReduce (s)

Both approaches produced the same result for the word counting. This is an important aspect, as it shows the validity of our implementation. Other examples such as data extraction from a CSV file have been implemented but are not presented in this document.

6.3 Use Case

In this Section, we present three use cases considered to apply our approach. The first use case is from the IoT field and is described in Section 6.3.1. It helps us consider the high variability aspect of Big Data, where we process a large collection of small datasets.

The second use case is about Smart Grids and is described in Section 6.3.2. Nowadays, the Smart Grid presents new opportunities to apply Big Data. As a matter of fact, it helps to consider the volume aspect of Big Data as we have many nodes generating data to be stored and analyzed.

The third use case is about Building Management Systems and is described in Section 6.3.3. In a building environment, the transmitted data can easily vary from few kilobytes to several gigabytes a day going back and forth between the sensors, actuators, and control processes. That's the reason why we choose this use case to apply our Big Data approach.

The comparison performed in Section 6.2 was based on a JavaScript implementation, while all the use cases are performed using a C++ implementation. This is due to the fact that ndnSIM doesn't support JavaScript, and its simulation language is C++.

6.3.1 IoT

The development of big data is rapidly accelerating and affecting all areas of technologies and businesses. As the Internet of Things (IoT) development is following this trend and producing tremendous amount of data, it has played a major role on the big data landscape [161]. New challenges are presented by the capability to analyze and use huge amounts of IoT data, including applications in smart cities, smart transport and grid systems, energy smart meters, and remote patient healthcare monitoring devices. Rethinking big data to enable built-in data aggregation can benefit both the networking side, by reducing the network traffic, and the processing side by sharing the results between the compute infrastructure at the edge of the network.

Description

For our use case, we consider an IoT temperature monitoring system. We have a set of devices (smart temperature and humidity sensors) that are queried periodically to provide temperature information. In this application, we have redundancy when requesting temperature informations. In fact, all the nodes requesting a temperature value will directly query the IoT device, even if

the data had been requested before. This scenario is a good candidate for applying our big data approach (Section 3.1 and 3.2).

Let's take an example of a data name: /location/temperature/201803201000/TempY/ These data are for the temperature value on March 20, 2018 at 10am from the IoT device TempY located in a room in a building (location is formed using the building name and the room number). Requesting these data will trigger an Interest for replicating the data on the DFS located in the Big Data platform (Figure 6.3). The consequence of this replication is the fact that future request for these data can be retrieved from the replicas, but also from some caches along the network; as NDN enable in-network caching. This considerably alleviates the load of the IoT devices.

Having these temperature measures on the DFS, we want now to make a computation using hourly historical values for a month. The compute request is issued from the cloud data center and performed on the Big Data platform. With this approach, we mutualize the use of the infrastructure and avoid redundancy in the computation. To evaluate the behavior and validate our approach, we have implemented our use case in a simulator. Simulations have been performed using the ndnSIM simulator [85]. We simulated a case with 20 IoT devices, 10 edge-storage/compute nodes, and 100 servers at the cloud data center. The simulation has been performed using IP communication and then with our NDN Distributed File System approach.

The platform used for this simulation is different from the one presented in Section 6.1. This is first due to the fact that we needed more nodes in this experiment and were bound by some resource restrictions in the cluster in our lab. Also, this platform is more adapted to IoT Big data computation as it includes the use of IoT devices and enables edge computing. Using ndnSIM, we were able to easily perform simulations up to the number of desired nodes.

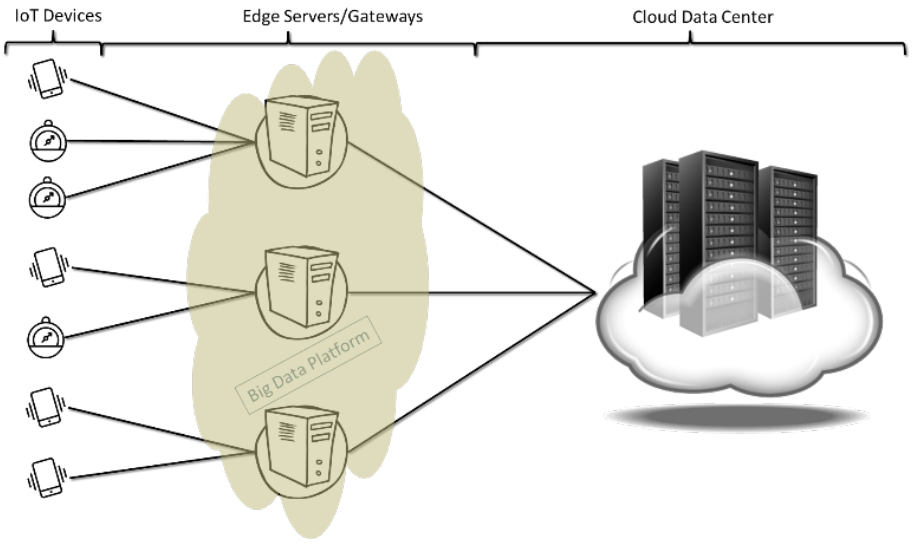

Figure 6.3: IoT Big Data Architecture

Results and discussions

The simulations have been performed for both the Distributed File System approach (Section 3.1) and the computation distribution (Section 3.2). We examined the packets exchanged on the network and the number of compute request Vs number of compute execution. NDN provides a set of tools. The dumping tool *ndndump* [162] was created to provide a tcpdump-like tool for Named Data Networking (NDN), for dumping data traffic between nodes over a NDN network. So for our approach, we used the *ndndump* tool (used during the comparison in Section 6.2), while using *tcpdump* for the IP based solution, to dump the traffic and perform analysis.

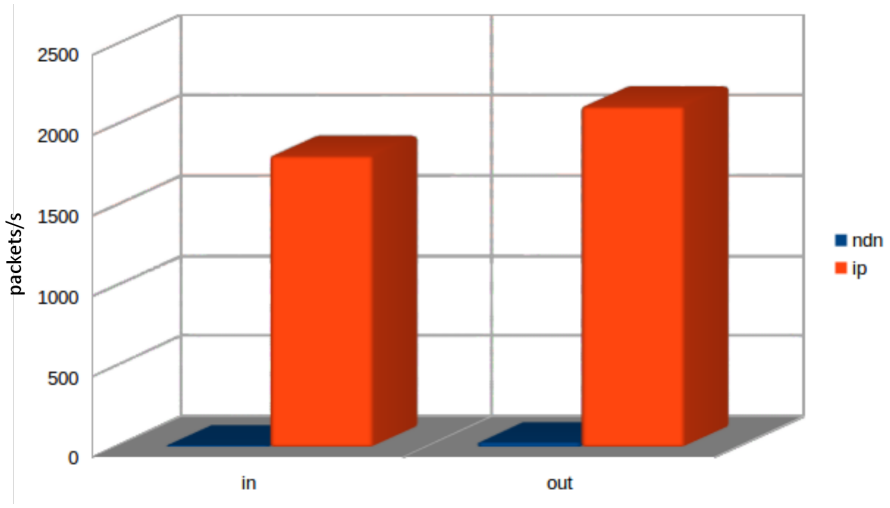

Figure 6.4: Average packet rate

Figure 6.4 shows the average packet rate on the network. The impact of replication factor on requesting the data from the source is linear. That means that the Interests are aggregated by NDN. That shows the efficiency of the system as only a minimal number of packets is transmitted.

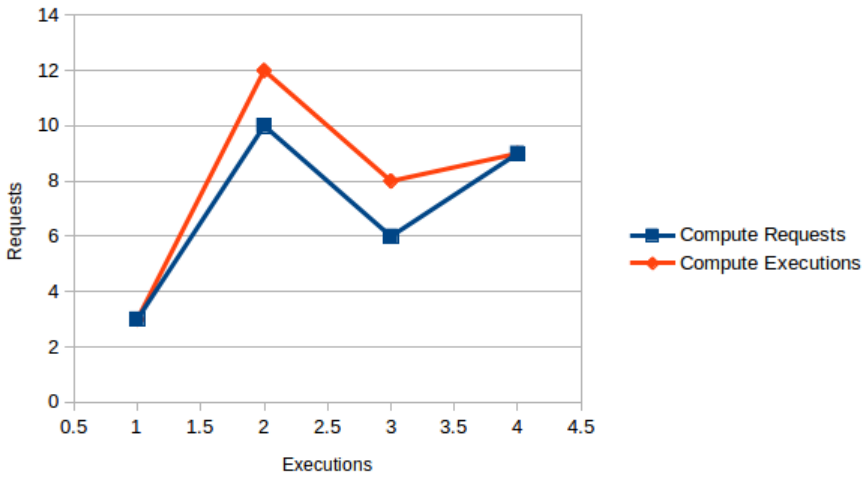

Figure 6.5: Compute request vs Execution - IP

Figure 6.5 shows that using an IP based solution, if a computation for a data is requested

for a number of time, that computation will be executed at least the same number of time and more if an error occurs during a computation. Our approach optimizes the computation and reuses already executed results. In this case, partial results for data chunks that have already evaluated and need to be reused in another evaluation, are not re-evaluated but the result is retrieved directly on the network.

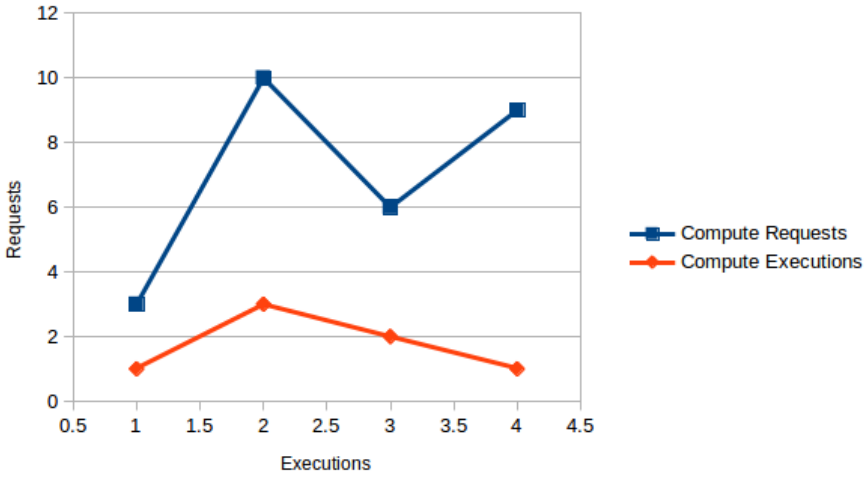

Figure 6.6: Compute request vs Execution - NDN

We can see in Figure 6.6 that based on our NDFS, the number of computation for the same content is almost always 1 with an exception when errors occur.

These results confirm that our communication properties (Section 4.4.1) hold as the protocols correctly transfer messages from ReplicationClient to Storage, from ComputationClient to Compute, between Storages and between Computes. We can see that with the different packets dumped.

Summary

This use case first strengthens our findings from Section 6.2, especially with the good throughput. This good throughput helps in the request for computation of already evaluated data. Also, the architecture considered in this use case highlights other benefits of our solution, which was not observed during the evaluation phase. For example, the fact that less packets are exchanged on the network using our approach.

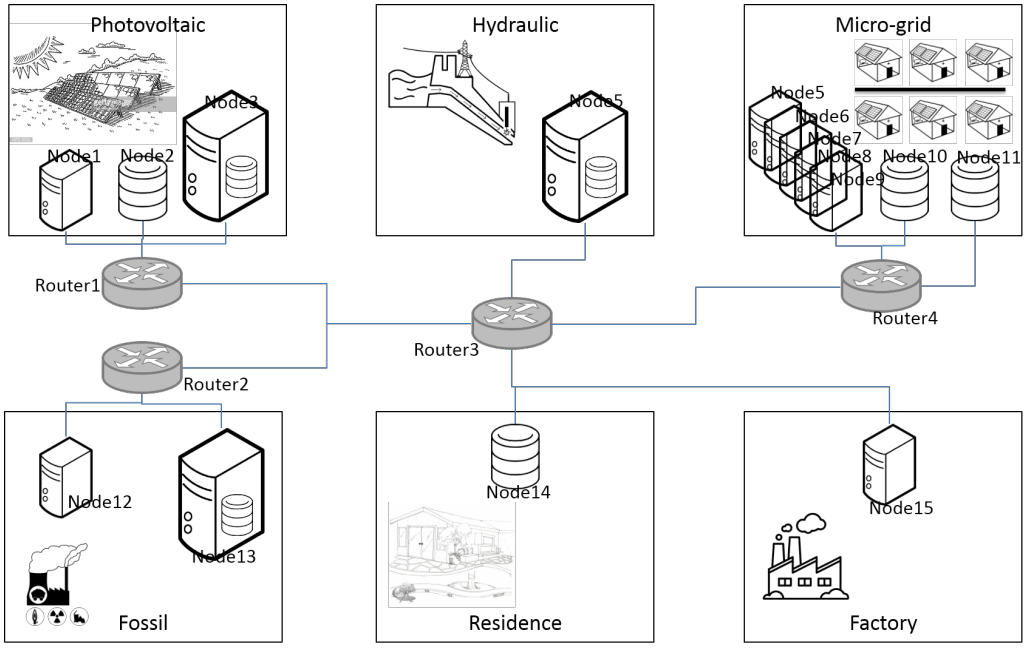

Figure 6.7: Smart Grid Scenario Architecture

6.3.2 Smart Grid

Since the emergence of the renewable energy sources, energy production scenario are evolving toward adoption of such an energy sources to address the foreseen oil shortage and the increasing demand for energy [163]. Therefore, stakeholders are changing their energy production and distribution processes to provide judicious uses of energy resources and a more "green" production [164]. This trend initiated a change in the model of the energy market. The transition is begun from a monopolistic single-provider model to a competitive model where several players (providers, vendors, and even consumers themselves) become producers in an open model [165]. Cheap photovoltaic panels and other affordable sources of renewable energy are heavily supporting this trend.

Description

The Smart Grid is defined as an electrical grid in which a set of operations and energy measures are performed for electricity production and distribution control [166]. It depends on measurements to provide an accurate state estimation. As state estimation guarantees the balance by controlling the production, gathering and processing these measurements is crucial to the Smart Grid. Measurement within the Smart Grid is based on two main devices. Phasor Measurement Units (PMUs), used to monitor and control the power system by measuring voltage angular and magnitude as well as active and reactive power.[167]. The Smart Meter, used to monitor the energy consumption at the consumer level (Residence in Figure 6.7) [168]. Communication is crucial in the Smart Grid to provide a reliable delivery of power from the sources to the consumers. Indeed, power delivery depends on reliable and real-time information sharing [169]. The Smart Grid is considered as a network of computers and power infrastructures that monitors and manages energy usage [170]. The use of advanced technologies and applications generates a huge amount of data that is very important for the Smart Grid. For example consumption data,

billing information etc. Data in the Smart Grid need to be saved, shared among many entities and need to be analyzed for control, real-time pricing etc. The data computation are repeated at many nodes and should be fast. Let's consider a network with a huge number $\mathcal{N}$ of subscribers. An information that needs to be processed at each node will be requested from the central source (measurement device) by the $\mathcal{N}$ subscribers and processed $\mathcal{N}$ times. New challenges are faced by the Smart Grid such as how to alleviate the load of the measurement device, to preserve the data in case of network failure and mutualise on the infrastructure for the data computation. Our Big Data approach is a good candidate in order to address these challenges.

For our use case, we consider the topology in Figure 6.7. In the Smart Grid, many Compute/Storage nodes are deployed at every Power plant, one storage/compute node at the microgrid level and a compute node at the User Home level. The compute nodes advertise */electricity/compute* to inform about the computation capability, while Storage nodes advertise */electricity/storage* for storage capability. When a PMU receives an Interest for a measure, it sends an interest to replicate that measure on the network. It defines the replication factor (number of replica that should be available on the network) for the data. Our experiment consisted in starting a set of Clients which hold a specific information such as the dynamic pricing of electricity that will be replicated. For example, using a name such as */electricity/prices/8pm/ProvX*, we define the prices at 8:00 pm for the provider ProvX. A set of nodes which are storage/compute nodes, will replicate the data after having received the replication interest */electricity/storage/RepFactor/Rank/prices/8pm/ProvX* (conforms with the language defined in Section 3.5). The replication factor (RepFactor, which is the number of replicas for a data) is dynamically determined by the Client. The rank (Rank) is essential for the heartbeat mechanism as discussed in Section 3.1.4 . We then start another set of nodes (located at the Residence level of Figure 6.7) that we will call User application which requested the replicated data. To evaluate the behavior and validate our approach, we have implemented our use case in a simulator.

Results and discussions

We only consider one aspect; the packets exchanged on the network. The simulations were concerning the Distributed File System approach. We compare our results with the ones from a scenario using NDN and in-network storage [171] in Table 6.3.2. This shows that the use of our approach significantly reduce the average number of packets exchanged on the network. We also compare these results with the theoretical bandwidth used in case of IP-Network [172][173]. This result once again confirms the network optimization introduced by our approach. In fact, the use of NDN cache, and the way replications are performed through our approach help a lot in obtaining data very closely to the requesting node. This reduces the number of packets that have to be forwarded to the producer of the data. These results confirm that our communication properties (Section 4.4.1) hold.

	NDN DFS	NDN with cache	IP
InData (kbits/s)	11.37	7.5	1810.9
OutData (kbits/s)	29.92	161.8	2117.5

Table 6.4: Average packet rate in the case of IP-network, NDN-Network with cache and NDN with DFS

Summary

The simulation of the architecture presented in this use case and the IP theoretical values confirmed our good throughput results presented in Section 6.2. Additionally, these results show that our architecture considered in Chapter 3 and implemented in Chapter 5 is performing well and has an added value to the NDN architecture. In fact the approach using our architecture presented better results compared to a default NDN approach.

6.3.3 Building Management System

The development of the Internet of Things (IoT) is rapidly growing and leading ways for changes and improvements in many areas of technology. Building Management System (BMS) is not an exception as IoT devices are widely used as lower energy consumption equipment to sense and actuate while reducing infrastructure costs. Moreover, those devices are enabler for interoperability between proprietary systems [174] in an environment where communication is crucial for the success of a solution.

The Edge Computing consists of placing the data collected by the sensors on the periphery of the IoT infrastructure. There are many solutions for processing data near the sensors. Based on event programming, a collector node does not necessarily send the data to the cloud and starts the analysis process directly afterward. The Edge Computing architecture and the IoT requirements reduce the processing time of sensor data placed on an industrial site. Because the data are stored on edge nodes near the production sites, the processing functions might be hosted directly on the local infrastructure. Such an approach allows manufacturers to obtain a near-real-time treatment of the information collected [175]. Sectors such as energy and smart building need this speed to handle huge amounts of often-critical data on productivity and security.

Description

In building management, several protocols are involved to build the control loop between the sensors and actuators. As illustrated in Figure 6.8, usually protocols such as BACnet [176] are used to implement the communications inter-controllers and protocols such as KNX [177] or simple IO are used to connect devices including sensors and actuators to those controllers. However, control is not limited to a single controller within its own control loop, but a control application involves several controllers. Indeed, for an HVAC application, temperature regulation might involve several local controllers regulating the temperature per room or per floor driven by a common objective represented by a setpoint to maintain.

Implementing such a control system requires analysis and processing of data coming from all over the system. That implies disseminating sensors generated data within the network. Our proposed architecture leverages the dissemination property of NDN to reduce the interactions between the sensors (Section 6.3.1). For the best exploitation of this property and to keep the compatibility with the brownfield deployments, we propose to introduce two new components to the architecture. The first component is the NDN-to-BACnet adapter. Its role on this architecture is to be an extension to the local controller in such a way that it adapts every request from/to the controller to NDN communication pattern. The second component is the edge NDN forwarder that are designed to be deployed at the floor level.

As illustrated in Figure 6.9, the way those components are introduced to the architecture is non-invasive to the established system. In addition to the classical provisioning path kept unchanged between the engineering station and the controllers, the two layers containing the adapters and the forwarders introduce the dissemination property to the system. The forwarding

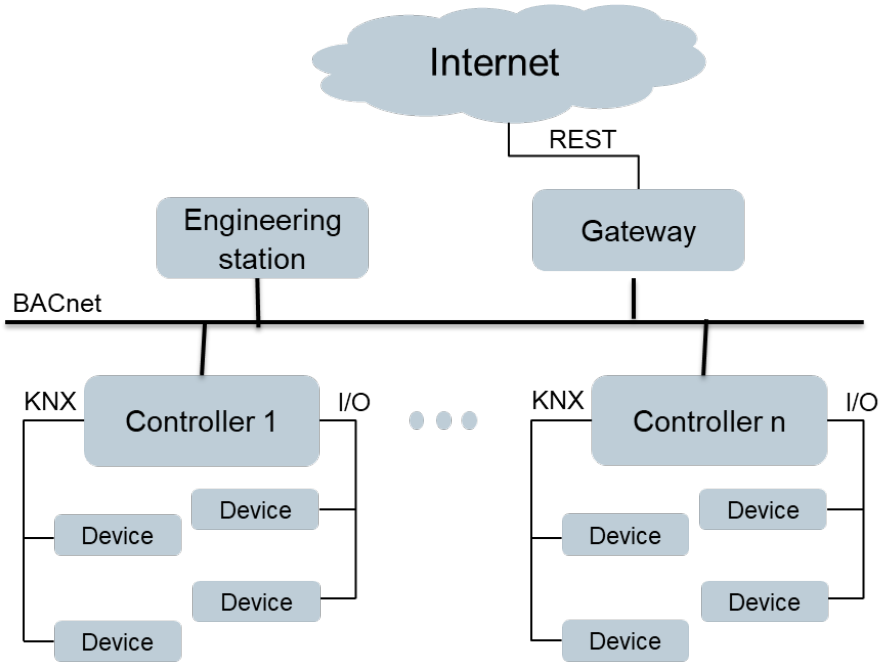

Figure 6.8: Architecture

layer optimizes the way data packets are transmitted. As the sensors data flows between the controllers, it keeps a copy on the Content Store within those floors' forwarders. Data are retrieved only once from the sensor using the NDN-to-BACnet adapter. After that the same chunk of data is retrieved only once per floor and served from the controllers local to the floor from the nearest common forwarder. This aspect of the architecture reduces drastically the amount of network exchanged (as presented in Section 6.3.2) and enables a new dimension of scalability to the system.

The processing decision at the edge might be driven by several techniques including stream processing where the data runs over several pipes and filters to produce the result. On our proposed architecture, we opted for a MapReduce model to process data and implement the central control mechanism. The assumption on this processing model is driven by the building management system style where a control application tries to orchestrate individual control loops towards the same objective. We use the two newly introduced component as infrastructure to run the map and the reduce operations. The NDN forwarders are used in this architecture as compute node at the edge that runs the map and reduce operations. The NDN-to-BACnet nodes are considered as storage nodes from which the compute nodes can retrieve the data chunks to be processed.The data chunks here are NDN segments.

For our use case, we consider the "Jeddah Tower". This building which will be the world's next tallest skyscraper is under construction in Jeddah, Saudi Arabia. It is expected to be 1 kilometer height, and is designed to include 168 floors and 2 basements with around 59 elevators. This building is a good candidate for applying our Building Management System approach.

Let's consider the 168 floors, using traditional IP based controller, we will need 2 controllers at each floor, which will result in 336 controllers for the whole building. This solution can be costly, and also a controller will be able to manage only the devices located at the floor where it is. Elevators are shared resources between all the floors of the building. This means that at some point in time, controllers have to share data information regarding devices located in the

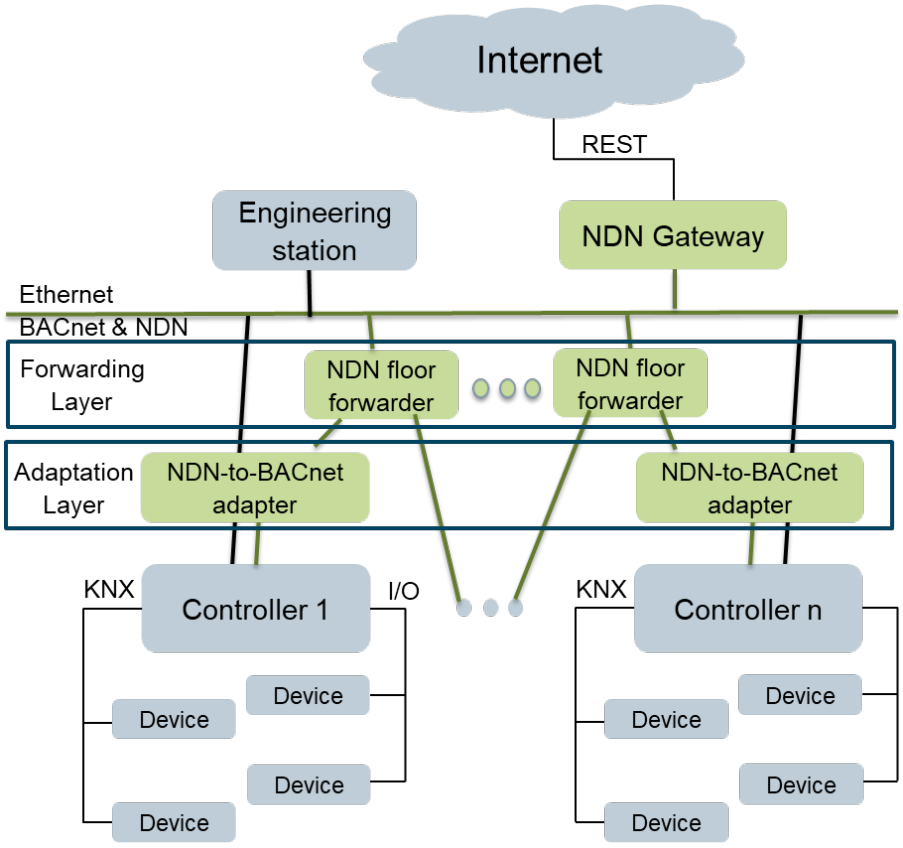

Figure 6.9: Architecture with NDN

elevators. The management design has to take this aspect into consideration. In a building like the "Jeddah Tower", elevator downtime is a crucial aspect. One wants to be able predict them and avoid them as much as possible or be able to schedule them during a period for which there is less traffic. Monitoring through the IoT devices can help improve. Using an approach based on NDN, We use the NDFS (Section 3.1) to replicate data from these elevators device over the network. When requesting data from an IoT device related to the elevator, the data generate an Interest for replicating that data on the DFS. The use of replication has a great impact, as future requests can be served from replicas, and from cache, thanks to NDN in-network caching.

We want to make a computation on a daily basis for the number of trips an elevator made during a period of time. Having data available in NDN, we can apply the NMapReduce (Section 3.2) for the computation. With this approach we are able to make part and the whole result available for another controller requesting the same computation or computation needing an already evaluated part.

Our use case has been implemented and evaluated using simulations. The platform considered in this simulation is the same as in Section 6.3.1. Based on the network simulator ns-3 [86], ndnSIM [85] is a discrete event based network simulator for Named Data Networking. ndnSIM matches the simulation platform with the latest advancements of NDN research, giving the opportunity to simulate code for real application.

Results and discussions

During the experimentation, we consider the number of requests compared with the number of cache hits. The requests concerned those for a data replicated and also request for already evaluated data in the case of a computation. Figure 6.10 shows the number of request and numbers of them have been satisfied by results available in cache. We can conclude that the use of NDN and our replication approach help to reduce bandwidth consumption. These results confirm that our communication properties (Section 4.4.1) hold.

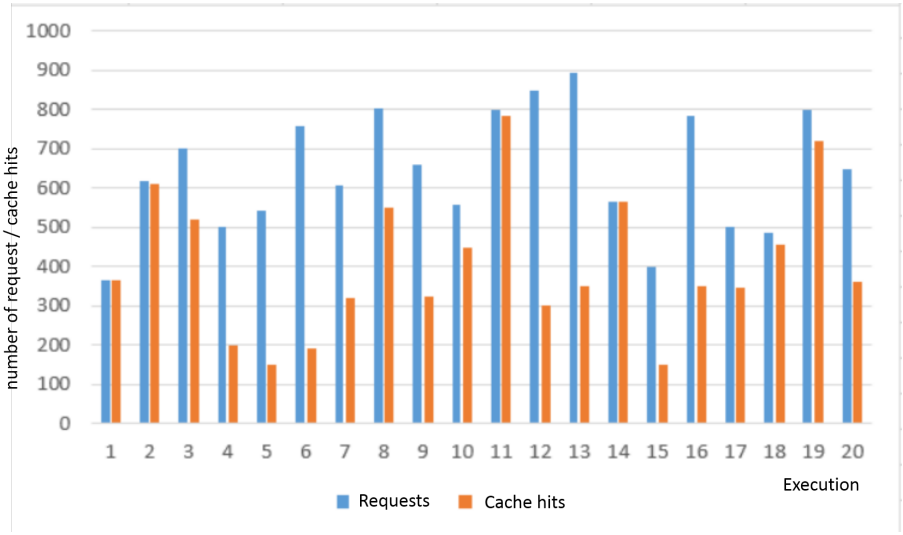

Figure 6.10: Number of request vs cache hit

Summary

This use case brought the light and explained the results from Section 6.3.1 and Section 6.3.2. In fact, we have presented one of the reasons why our architecture gave the previous results. This has been done by comparing the number of requests issued and the number of requests satisfied from a cache. In each case, most of the issued requests were satisfied from cache. This also explains the good throughput from Section 6.2.

6.4 Summary

In this Chapter, we described our experimental platform by presenting its hardware architectures, software as well as the test scenario which was executed in this environment. We also presented the experimentation used to evaluate our solution, at the DFS level as well as the computation distribution level. We found that the number of replica for a data can be scaled up without issue, as the time used for the replication of a data decreases with the increase of the replication factor. We have compared our results with experimentation results using the Hadoop architecture. We have found that our approach presents good results compared to Hadoop when considering replication time using a DFS and computation execution time for distributed computation.

Finally, we have considered three use cases in which we applied our solution to solve real life problems and showing the benefit of our solution. The First use case showed that our solution is suitable when considering Big Data variability aspect. The second one presented adequacy of

the solution when dealing with Big Data volume attribute. And the last use case highlighted one of the reason why our solution presented these good results.

With all these results of experiments, we can conclude that our architecture fulfills well the objectives that we had set ourselves, namely, to propose a Big Data architecture to NDN (Chapter 3) satisfying the properties defined in Chapter 4.

In the next chapter, we present our conclusions and also the perspectives offered by our study.

Chapter 7

Conclusion and Perspectives

Contents

The ICN paradigm is proposed as an architecture for the Future Internet in order to enable the Internet to fulfill its new role with the increase of usages. This is done by focusing on the data rather than on the hosts of the network, with the goal of moving data more efficiently to users. Several ICN architectures are proposed and from our point of view, the NDN architecture is the most advanced and brings together the largest community of researchers and companies for its development. By providing its nodes with a caching capability, the NDN network allows storing copies of the contents in the network to improve their distribution. The adoption of NDN as the Future Internet architecture will have a big impact on how applications are designed and implemented. This is due to the switching of the paradigm between IP based Network and NDN architecture based on data naming. Considering the OSI model, Big Data computations resides at the application layer. NDN enables the management of the data at the network layer thanks to the in-network caching, name-based routing, native support of multicast, and easy data access.

Our doctoral thesis work leads to the definition and implementation of a Big Data architecture in which data and script code are in motion to ease computations based on the NDN network architecture. In this chapter, we summarize our contributions and present future perspectives for the rest of our work. Finally, we list the publications made during this PhD thesis. This work fully integrates with the works on software mobility of our working group.

7.1 Summary of contributions

The main objective of this thesis is to provide a Big Data architecture for Named Data Networking architecture. We primarily gave an overview of the state of the art for Big Data, distributed system in general and distributed computation particularly, and presented the NDN paradigm. From there, we defined a formal model for our architecture and developed a framework for our architecture.

7.1.1 Formal definition of a software architecture

For our architecture, we have two property families to deal with. First, properties related to the parsers for our different languages and their implementation, and secondly, temporal properties related to the whole system. This lead to the adoption of two proof approaches; one using Coq and another one using timed automata.

Our formal specifications written in Coq provides a formal definition for the replication and the computation languages. Ours timed automata model components of a software architecture for parallel and distributed computing. The distributed file system derived from this architecture can handle different types of data files and also support a high scale in terms of users. For the computation, based on the traditional map-reduce model, we describe an architecture that can adapt to the number of available resources and fully use the data available at the network level to speed up the computations it has to deal with. All these properties have been formally proven.

7.1.2 Development of a framework

The formal modeling carried out in the first part of this thesis pilots the development of an architecture for Big Data computation based on NDN. Our framework is composed of two layers, a DFS, and a distributed computation mechanism. We have studied how to evolve a data layer in a big data context. We have defined a specific naming which enforces the features of our distributed file system: a replication factor and failover control. Using the specification of the system, we have considered an approach of implementation from specification, which enables us to implement a prototype and a concrete version of our solution. With the use of simulations, we have shown that the deployment of large data over an NDN network is achieved in suitable conditions under the control of a custom forwarding strategy. More importantly, we have shown that the proposed approach is highly scalable and fault-tolerant. In fact, increasing the number of user applications, decrease the meantime needed for these applications to access the data, and from one replica the DFS moves to a stable state. We have also addressed the computation part of the big data processing by providing a fully distributed and efficient approach on top of the DFS.

7.1.3 Measurements for evaluation

Measurements have been performed for the evaluation of the developed framework through experimentations. These measurements show that our approach is highly scalable, as the network size doesn't have a negative impact on the replication time, which decreases when increasing the number of replicas.

In Section 6.2.2, a Big Data computation has been performed using our framework and produces the same result as Hadoop but with less execution time.

Our Big Data approach is correctly responding when considering the Big Data variability, and volume aspects. It also guarantees consistency and availability regarding the CAP theorem.

To ensure reproducibility, each stage of the experimentations, have been controlled. Automation has been provided where possible and description otherwise. The reproducibility of this experimental approach has been evaluated and the source code of our work made available on github.

7.2 Future works

Our work presents many perspectives that could be considered. First, the aspect of the computation considered in this thesis is a batch processing approach. One future work could be the extension of the computation mechanism to include streaming processing. Also, for data to be replicated or computed in the network, these data have to be available in NDN packet format [7]. In this thesis, we relied on tools such as ndnputchunks (Ndn essential tools) to have the data available in NDN format. This tool has some bad performances. A possible work could be the development of a tool with better performances. Nowadays, Machine Learning (ML) is a hot topic and considered in almost all the domain. Another research possibility could be the extension of our framework to provide Machine Learning capability using NDN.

Definitely, we can consider presenting at meetings, looking for industrial partnerships to push this work as a standard when talking about Big Data in NDN, but also promote this solution to companies to take advantage of it when they look for a solution to gain insight from their data. I think we have to continue and look for partnerships to develop a complete approach of Big Data on NDN. This work can be continued on a post-doc where the Big Data aspects will be combined with predictive approaches for NDN network maintenance prediction.

Publications

- Distributed File System for NDN: an IoT Application
  Dongo J, Atik Y, Mahmoudi C, Mourlin F.
  In 2018 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT) (pp. 138-141).

- NDN Log Analysis Using Big Data Techniques: NFD Performance Assessment.
  Dongo J, Mahmoudi C, Mourlin F.
  In 2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService) (pp. 169-175).

- Service Abstraction Layer for Smart Grid Measurement
  Dongo J, Mahmoudi C, Mourlin F.
  In 2018 Renewable Energies, Power Systems & Green Inclusive Economy (REPS-GIE) (pp. 1-6).

- Distributed Edge Solution for IoT based Building Management System with NDN
  Dongo J, Foltete L, Mahmoudi C, Mourlin F.
  In 2019 Global Information Infrastructure and Networking Symposium (GIIS 2019)

- Elastic Gigabit NDN Forwarder for Big Data Applications
  Dongo J, Mahmoudi C, Mourlin F.
  In 2019 Global Information Infrastructure and Networking Symposium (GIIS 2019)

[7] https://irl.cs.ucla.edu/ cawka/spec/index.html

- NDFS: The Named Data Networking Distributed File System
  Dongo J, Mahmoudi C, Mourlin F.
  Submitted

Appendix A

Replication language parser in Coq

```
1
2  Add LoadPath "/home/lacl/Documents/coq_learning/ndfs".
3  Require Import ast.
4  Require Import String.
5
6  Require Import Notations.
7  Require Import Logic.
8  Require Import Specif.
9
10
11 From Coq.Lists Require Import List.
12 From Coq.Numbers.Cyclic.Int31 Require Import Int31.
13 From Coq.Program Require Import Syntax.
14 From MenhirLib Require Import Tuples.
15 From MenhirLib Require Import Alphabet.
16 From MenhirLib Require Grammar.
17 From MenhirLib Require Automaton.
18
19 Unset Elimination Schemes.
20
21 Module Import Gram <: Grammar.T.
22
23 Local Obligation Tactic := let x := fresh in intro x; case x; reflexivity.
24
25 Inductive terminal' : Set :=
26 | EOL't
27 | INT't
28 | SLASH't
29 | STORAGE't
30 | STRING't.
31 Definition terminal := terminal'.
32
33 Program Instance terminalNum : Numbered terminal :=
34   { inj := fun x => match x return _ with
35     | EOL't => Int31.On
36     | INT't => Int31.In
37     | SLASH't => (twice Int31.In)
38     | STORAGE't => (twice_plus_one Int31.In)
39     | STRING't => (twice (twice Int31.In))
40     end;
41     surj := (fun n => match Int31.phi n return _ with
42     | 0 => EOL't
```

```
43      | 1 => INT't
44      | 2 => SLASH't
45      | 3 => STORAGE't
46      | 4 => STRING't
47      | _ => EOL't
48      end)%Z;
49      inj_bound := 5%int31 }.
50  Instance TerminalAlph : Alphabet terminal := _.
51
52  Inductive nonterminal' : Set :=
53  | data'nt
54  | dataName'nt
55  | domain'nt
56  | firstSegement'nt
57  | lastSegment'nt
58  | main'nt
59  | replicationFactor'nt
60  | replicationIndex'nt
61  | replicationParameter'nt
62  | rootDomain'nt.
63  Definition nonterminal := nonterminal'.
64
65  Program Instance nonterminalNum : Numbered nonterminal :=
66    { inj := fun x => match x return _ with
67      | data'nt => Int31.On
68      | dataName'nt => Int31.In
69      | domain'nt => (twice Int31.In)
70      | firstSegement'nt => (twice_plus_one Int31.In)
71      | lastSegment'nt => (twice (twice Int31.In))
72      | main'nt => (twice_plus_one (twice Int31.In))
73      | replicationFactor'nt => (twice (twice_plus_one Int31.In))
74      | replicationIndex'nt => (twice_plus_one (twice_plus_one Int31.In))
75      | replicationParameter'nt => (twice (twice (twice Int31.In)))
76      | rootDomain'nt => (twice_plus_one (twice (twice Int31.In)))
77      end;
78      surj := (fun n => match Int31.phi n return _ with
79      | 0 => data'nt
80      | 1 => dataName'nt
81      | 2 => domain'nt
82      | 3 => firstSegement'nt
83      | 4 => lastSegment'nt
84      | 5 => main'nt
85      | 6 => replicationFactor'nt
86      | 7 => replicationIndex'nt
87      | 8 => replicationParameter'nt
88      | 9 => rootDomain'nt
89      | _ => data'nt
90      end)%Z;
91      inj_bound := 10%int31 }.
92  Instance NonTerminalAlph : Alphabet nonterminal := _.
93
94  Include Grammar.Symbol.
95
96  Definition terminal_semantic_type (t:terminal) : Type:=
97    match t with
98    | STRING't =>         (string)%type
99    | STORAGE't => unit%type
100   | SLASH't => unit%type
101   | INT't =>          (nat)%type
```

```
  | EOL't => unit%type
  end.

Definition nonterminal_semantic_type (nt:nonterminal) : Type:=
  match nt with
  | rootDomain'nt =>       (string)%type
  | replicationParameter'nt =>       (replicationParameter)%type
  | replicationIndex'nt =>       (nat)%type
  | replicationFactor'nt =>       (nat)%type
  | main'nt =>         (ast)%type
  | lastSegment'nt =>       (nat)%type
  | firstSegement'nt =>       (nat)%type
  | domain'nt =>       (string)%type
  | dataName'nt =>       (dataName)%type
  | data'nt =>       (string)%type
  end.

Definition symbol_semantic_type (s:symbol) : Type:=
  match s with
  | T t => terminal_semantic_type t
  | NT nt => nonterminal_semantic_type nt
  end.

Inductive production' : Set :=
| Prod'rootDomain'0
| Prod'replicationParameter'0
| Prod'replicationIndex'0
| Prod'replicationFactor'0
| Prod'main'0
| Prod'lastSegment'0
| Prod'firstSegement'0
| Prod'domain'0
| Prod'dataName'0
| Prod'data'0.
Definition production := production'.

Program Instance productionNum : Numbered production :=
  { inj := fun x => match x return _ with
    | Prod'rootDomain'0 => Int31.On
    | Prod'replicationParameter'0 => Int31.In
    | Prod'replicationIndex'0 => (twice Int31.In)
    | Prod'replicationFactor'0 => (twice_plus_one Int31.In)
    | Prod'main'0 => (twice (twice Int31.In))
    | Prod'lastSegment'0 => (twice_plus_one (twice Int31.In))
    | Prod'firstSegement'0 => (twice (twice_plus_one Int31.In))
    | Prod'domain'0 => (twice_plus_one (twice_plus_one Int31.In))
    | Prod'dataName'0 => (twice (twice (twice Int31.In)))
    | Prod'data'0 => (twice_plus_one (twice (twice Int31.In)))
    end;
    surj := (fun n => match Int31.phi n return _ with
    | 0 => Prod'rootDomain'0
    | 1 => Prod'replicationParameter'0
    | 2 => Prod'replicationIndex'0
    | 3 => Prod'replicationFactor'0
    | 4 => Prod'main'0
    | 5 => Prod'lastSegment'0
    | 6 => Prod'firstSegement'0
    | 7 => Prod'domain'0
    | 8 => Prod'dataName'0
```

```
161     | 9 => Prod'data'0
162     | _ => Prod'rootDomain'0
163     end)%Z;
164     inj_bound := 10%int31 }.
165 Instance ProductionAlph : Alphabet production := _.
166 
167 Definition prod_contents (p:production) :
168   { p:nonterminal * list symbol &
169     arrows_left (map symbol_semantic_type (rev (snd p)))
170                 (symbol_semantic_type (NT (fst p))) }
171  :=
172   let box := existT (fun p =>
173     arrows_left (map symbol_semantic_type (rev (snd p)))
174                 (symbol_semantic_type (NT (fst p))))
175   in
176   match p with
177   | Prod'data'0 => box
178     (data'nt, [T STRING't])
179     (fun da =>
180     ( da )
181 )
182   | Prod'dataName'0 => box
183     (dataName'nt, [NT lastSegment'nt; T SLASH't; NT firstSegement'nt; T SLASH't;
          NT data'nt; T SLASH't; NT domain'nt; T SLASH't])
184     (fun seg2 _7 seg1 _5 dat _3 dom _1 =>
185     ( Name dom dat seg1 seg2 )
186 )
187   | Prod'domain'0 => box
188     (domain'nt, [T STRING't])
189     (fun d =>
190     ( d )
191 )
192   | Prod'firstSegement'0 => box
193     (firstSegement'nt, [T INT't])
194     (fun seg1 =>
195     ( seg1 )
196 )
197   | Prod'lastSegment'0 => box
198     (lastSegment'nt, [T INT't])
199     (fun seg2 =>
200     ( seg2 )
201 )
202   | Prod'main'0 => box
203     (main'nt, [T EOL't; NT dataName'nt; NT replicationParameter'nt; T STORAGE't; T
          SLASH't; NT rootDomain'nt; T SLASH't])
204     (fun _7 dn rp _4 _3 r _1 =>
205      ( Main r rp dn )
206 )
207   | Prod'replicationFactor'0 => box
208     (replicationFactor'nt, [T INT't])
209     (fun r1 =>
210     ( r1 )
211 )
212   | Prod'replicationIndex'0 => box
213     (replicationIndex'nt, [T INT't])
214     (fun r2 =>
215     ( r2 )
216 )
217   | Prod'replicationParameter'0 => box
```

```
218     (replicationParameter'nt, [NT replicationIndex'nt; T SLASH't; NT
            replicationFactor'nt; T SLASH't])
219     (fun index _3 factor _1 =>
220     ( Parameters factor index )
221 )
222   | Prod'rootDomain'0 => box
223     (rootDomain'nt, [T STRING't])
224     (fun s =>
225     ( s )
226 )
227   end.
228
229 Definition prod_lhs (p:production) :=
230   fst (projT1 (prod_contents p)).
231 Definition prod_rhs_rev (p:production) :=
232   snd (projT1 (prod_contents p)).
233 Definition prod_action (p:production) :=
234   projT2 (prod_contents p).
235
236 Include Grammar.Defs.
237
238 End Gram.
239
240 Module Aut <: Automaton.T.
241
242 Local Obligation Tactic := let x := fresh in intro x; case x; reflexivity.
243
244 Module Gram := Gram.
245 Module GramDefs := Gram.
246
247 Definition nullable_nterm (nt:nonterminal) : bool :=
248   match nt with
249   | rootDomain'nt => false
250   | replicationParameter'nt => false
251   | replicationIndex'nt => false
252   | replicationFactor'nt => false
253   | main'nt => false
254   | lastSegment'nt => false
255   | firstSegement'nt => false
256   | domain'nt => false
257   | dataName'nt => false
258   | data'nt => false
259   end.
260
261 Definition first_nterm (nt:nonterminal) : list terminal :=
262   match nt with
263   | rootDomain'nt => [STRING't]
264   | replicationParameter'nt => [SLASH't]
265   | replicationIndex'nt => [INT't]
266   | replicationFactor'nt => [INT't]
267   | main'nt => [SLASH't]
268   | lastSegment'nt => [INT't]
269   | firstSegement'nt => [INT't]
270   | domain'nt => [STRING't]
271   | dataName'nt => [SLASH't]
272   | data'nt => [STRING't]
273   end.
274
275 Inductive noninitstate' : Set :=
```

```
276 | | Nis'26
277 | | Nis'25
278 | | Nis'24
279 | | Nis'23
280 | | Nis'22
281 | | Nis'21
282 | | Nis'20
283 | | Nis'19
284 | | Nis'18
285 | | Nis'17
286 | | Nis'16
287 | | Nis'15
288 | | Nis'14
289 | | Nis'13
290 | | Nis'12
291 | | Nis'11
292 | | Nis'10
293 | | Nis'9
294 | | Nis'8
295 | | Nis'7
296 | | Nis'6
297 | | Nis'5
298 | | Nis'4
299 | | Nis'3
300 | | Nis'2
301 | | Nis'1.
302 | Definition noninitstate := noninitstate'.
303 |
304 | Program Instance noninitstateNum : Numbered noninitstate :=
305 |   { inj := fun x => match x return _ with
306 |     | Nis'26 => Int31.On
307 |     | Nis'25 => Int31.In
308 |     | Nis'24 => (twice Int31.In)
309 |     | Nis'23 => (twice_plus_one Int31.In)
310 |     | Nis'22 => (twice (twice Int31.In))
311 |     | Nis'21 => (twice_plus_one (twice Int31.In))
312 |     | Nis'20 => (twice (twice_plus_one Int31.In))
313 |     | Nis'19 => (twice_plus_one (twice_plus_one Int31.In))
314 |     | Nis'18 => (twice (twice (twice Int31.In)))
315 |     | Nis'17 => (twice_plus_one (twice (twice Int31.In)))
316 |     | Nis'16 => (twice (twice_plus_one (twice Int31.In)))
317 |     | Nis'15 => (twice_plus_one (twice_plus_one (twice Int31.In)))
318 |     | Nis'14 => (twice (twice (twice_plus_one Int31.In)))
319 |     | Nis'13 => (twice_plus_one (twice (twice_plus_one Int31.In)))
320 |     | Nis'12 => (twice (twice_plus_one (twice_plus_one Int31.In)))
321 |     | Nis'11 => (twice_plus_one (twice_plus_one (twice_plus_one Int31.In)))
322 |     | Nis'10 => (twice (twice (twice (twice Int31.In))))
323 |     | Nis'9 => (twice_plus_one (twice (twice (twice Int31.In))))
324 |     | Nis'8 => (twice (twice_plus_one (twice (twice Int31.In))))
325 |     | Nis'7 => (twice_plus_one (twice_plus_one (twice (twice Int31.In))))
326 |     | Nis'6 => (twice (twice (twice_plus_one (twice Int31.In))))
327 |     | Nis'5 => (twice_plus_one (twice (twice_plus_one (twice Int31.In))))
328 |     | Nis'4 => (twice (twice_plus_one (twice_plus_one (twice Int31.In))))
329 |     | Nis'3 => (twice_plus_one (twice_plus_one (twice_plus_one (twice Int31.In))))
330 |     | Nis'2 => (twice (twice (twice (twice_plus_one Int31.In))))
331 |     | Nis'1 => (twice_plus_one (twice (twice (twice_plus_one Int31.In))))
332 |     end;
333 |     surj := (fun n => match Int31.phi n return _ with
334 |     | 0 => Nis'26
```

```
335       | 1 => Nis'25
336       | 2 => Nis'24
337       | 3 => Nis'23
338       | 4 => Nis'22
339       | 5 => Nis'21
340       | 6 => Nis'20
341       | 7 => Nis'19
342       | 8 => Nis'18
343       | 9 => Nis'17
344       | 10 => Nis'16
345       | 11 => Nis'15
346       | 12 => Nis'14
347       | 13 => Nis'13
348       | 14 => Nis'12
349       | 15 => Nis'11
350       | 16 => Nis'10
351       | 17 => Nis'9
352       | 18 => Nis'8
353       | 19 => Nis'7
354       | 20 => Nis'6
355       | 21 => Nis'5
356       | 22 => Nis'4
357       | 23 => Nis'3
358       | 24 => Nis'2
359       | 25 => Nis'1
360       | _ => Nis'26
361      end)%Z;
362      inj_bound := 26%int31 }.
363  Instance NonInitStateAlph : Alphabet noninitstate := _.
364
365  Definition last_symb_of_non_init_state (noninitstate:noninitstate) : symbol :=
366    match noninitstate with
367    | Nis'1 => T SLASH't
368    | Nis'2 => T STRING't
369    | Nis'3 => NT rootDomain'nt
370    | Nis'4 => T SLASH't
371    | Nis'5 => T STORAGE't
372    | Nis'6 => T SLASH't
373    | Nis'7 => T INT't
374    | Nis'8 => NT replicationFactor'nt
375    | Nis'9 => T SLASH't
376    | Nis'10 => T INT't
377    | Nis'11 => NT replicationIndex'nt
378    | Nis'12 => NT replicationParameter'nt
379    | Nis'13 => T SLASH't
380    | Nis'14 => T STRING't
381    | Nis'15 => NT domain'nt
382    | Nis'16 => T SLASH't
383    | Nis'17 => T STRING't
384    | Nis'18 => NT data'nt
385    | Nis'19 => T SLASH't
386    | Nis'20 => T INT't
387    | Nis'21 => NT firstSegement'nt
388    | Nis'22 => T SLASH't
389    | Nis'23 => T INT't
390    | Nis'24 => NT lastSegment'nt
391    | Nis'25 => NT dataName'nt
392    | Nis'26 => T EOL't
393    end.
```

```
394
395 Inductive initstate' : Set :=
396 | Init'0.
397 Definition initstate := initstate'.
398
399 Program Instance initstateNum : Numbered initstate :=
400   { inj := fun x => match x return _ with
401     | Init'0 => Int31.On
402     end;
403     surj := (fun n => match Int31.phi n return _ with
404     | 0 => Init'0
405     | _ => Init'0
406     end)%Z;
407     inj_bound := 1%int31 }.
408 Instance InitStateAlph : Alphabet initstate := _.
409
410 Include Automaton.Types.
411
412 Definition start_nt (init:initstate) : nonterminal :=
413   match init with
414   | Init'0 => main'nt
415   end.
416
417 Definition action_table (state:state) : action :=
418   match state with
419   | Init Init'0 => Lookahead_act (fun terminal:terminal =>
420     match terminal return lookahead_action terminal with
421     | SLASH't => Shift_act Nis'1 (eq_refl _)
422     | _ => Fail_act
423     end)
424   | Ninit Nis'1 => Lookahead_act (fun terminal:terminal =>
425     match terminal return lookahead_action terminal with
426     | STRING't => Shift_act Nis'2 (eq_refl _)
427     | _ => Fail_act
428     end)
429   | Ninit Nis'2 => Default_reduce_act Prod'rootDomain'0
430   | Ninit Nis'3 => Lookahead_act (fun terminal:terminal =>
431     match terminal return lookahead_action terminal with
432     | SLASH't => Shift_act Nis'4 (eq_refl _)
433     | _ => Fail_act
434     end)
435   | Ninit Nis'4 => Lookahead_act (fun terminal:terminal =>
436     match terminal return lookahead_action terminal with
437     | STORAGE't => Shift_act Nis'5 (eq_refl _)
438     | _ => Fail_act
439     end)
440   | Ninit Nis'5 => Lookahead_act (fun terminal:terminal =>
441     match terminal return lookahead_action terminal with
442     | SLASH't => Shift_act Nis'6 (eq_refl _)
443     | _ => Fail_act
444     end)
445   | Ninit Nis'6 => Lookahead_act (fun terminal:terminal =>
446     match terminal return lookahead_action terminal with
447     | INT't => Shift_act Nis'7 (eq_refl _)
448     | _ => Fail_act
449     end)
450   | Ninit Nis'7 => Default_reduce_act Prod'replicationFactor'0
451   | Ninit Nis'8 => Lookahead_act (fun terminal:terminal =>
452     match terminal return lookahead_action terminal with
```

```
453       | SLASH't => Shift_act Nis'9 (eq_refl _)
454       | _ => Fail_act
455       end)
456     | Ninit Nis'9 => Lookahead_act (fun terminal:terminal =>
457       match terminal return lookahead_action terminal with
458       | INT't => Shift_act Nis'10 (eq_refl _)
459       | _ => Fail_act
460       end)
461     | Ninit Nis'10 => Default_reduce_act Prod'replicationIndex'0
462     | Ninit Nis'11 => Default_reduce_act Prod'replicationParameter'0
463     | Ninit Nis'12 => Lookahead_act (fun terminal:terminal =>
464       match terminal return lookahead_action terminal with
465       | SLASH't => Shift_act Nis'13 (eq_refl _)
466       | _ => Fail_act
467       end)
468     | Ninit Nis'13 => Lookahead_act (fun terminal:terminal =>
469       match terminal return lookahead_action terminal with
470       | STRING't => Shift_act Nis'14 (eq_refl _)
471       | _ => Fail_act
472       end)
473     | Ninit Nis'14 => Default_reduce_act Prod'domain'0
474     | Ninit Nis'15 => Lookahead_act (fun terminal:terminal =>
475       match terminal return lookahead_action terminal with
476       | SLASH't => Shift_act Nis'16 (eq_refl _)
477       | _ => Fail_act
478       end)
479     | Ninit Nis'16 => Lookahead_act (fun terminal:terminal =>
480       match terminal return lookahead_action terminal with
481       | STRING't => Shift_act Nis'17 (eq_refl _)
482       | _ => Fail_act
483       end)
484     | Ninit Nis'17 => Default_reduce_act Prod'data'0
485     | Ninit Nis'18 => Lookahead_act (fun terminal:terminal =>
486       match terminal return lookahead_action terminal with
487       | SLASH't => Shift_act Nis'19 (eq_refl _)
488       | _ => Fail_act
489       end)
490     | Ninit Nis'19 => Lookahead_act (fun terminal:terminal =>
491       match terminal return lookahead_action terminal with
492       | INT't => Shift_act Nis'20 (eq_refl _)
493       | _ => Fail_act
494       end)
495     | Ninit Nis'20 => Default_reduce_act Prod'firstSegement'0
496     | Ninit Nis'21 => Lookahead_act (fun terminal:terminal =>
497       match terminal return lookahead_action terminal with
498       | SLASH't => Shift_act Nis'22 (eq_refl _)
499       | _ => Fail_act
500       end)
501     | Ninit Nis'22 => Lookahead_act (fun terminal:terminal =>
502       match terminal return lookahead_action terminal with
503       | INT't => Shift_act Nis'23 (eq_refl _)
504       | _ => Fail_act
505       end)
506     | Ninit Nis'23 => Default_reduce_act Prod'lastSegment'0
507     | Ninit Nis'24 => Default_reduce_act Prod'dataName'0
508     | Ninit Nis'25 => Lookahead_act (fun terminal:terminal =>
509       match terminal return lookahead_action terminal with
510       | EOL't => Shift_act Nis'26 (eq_refl _)
511       | _ => Fail_act
```

```
512     end)
513   | Ninit Nis'26 => Default_reduce_act Prod'main'0
514   end.
515
516 Definition goto_table (state:state) (nt:nonterminal) :=
517   match state, nt return option { s:noninitstate | NT nt =
          last_symb_of_non_init_state s } with
518   | Init Init'0, main'nt => None  | Ninit Nis'1, rootDomain'nt => Some (exist _
          Nis'3 (eq_refl _))
519   | Ninit Nis'5, replicationParameter'nt => Some (exist _ Nis'12 (eq_refl _))
520   | Ninit Nis'6, replicationFactor'nt => Some (exist _ Nis'8 (eq_refl _))
521   | Ninit Nis'9, replicationIndex'nt => Some (exist _ Nis'11 (eq_refl _))
522   | Ninit Nis'12, dataName'nt => Some (exist _ Nis'25 (eq_refl _))
523   | Ninit Nis'13, domain'nt => Some (exist _ Nis'15 (eq_refl _))
524   | Ninit Nis'16, data'nt => Some (exist _ Nis'18 (eq_refl _))
525   | Ninit Nis'19, firstSegement'nt => Some (exist _ Nis'21 (eq_refl _))
526   | Ninit Nis'22, lastSegment'nt => Some (exist _ Nis'24 (eq_refl _))
527   | _, _ => None
528   end.
529
530 Definition past_symb_of_non_init_state (noninitstate:noninitstate) : list symbol
        :=
531   match noninitstate with
532   | Nis'1 => []
533   | Nis'2 => []
534   | Nis'3 => [T SLASH't]
535   | Nis'4 => [NT rootDomain'nt; T SLASH't]
536   | Nis'5 => [T SLASH't; NT rootDomain'nt; T SLASH't]
537   | Nis'6 => []
538   | Nis'7 => []
539   | Nis'8 => [T SLASH't]
540   | Nis'9 => [NT replicationFactor'nt; T SLASH't]
541   | Nis'10 => []
542   | Nis'11 => [T SLASH't; NT replicationFactor'nt; T SLASH't]
543   | Nis'12 => [T STORAGE't; T SLASH't; NT rootDomain'nt; T SLASH't]
544   | Nis'13 => []
545   | Nis'14 => []
546   | Nis'15 => [T SLASH't]
547   | Nis'16 => [NT domain'nt; T SLASH't]
548   | Nis'17 => []
549   | Nis'18 => [T SLASH't; NT domain'nt; T SLASH't]
550   | Nis'19 => [NT data'nt; T SLASH't; NT domain'nt; T SLASH't]
551   | Nis'20 => []
552   | Nis'21 => [T SLASH't; NT data'nt; T SLASH't; NT domain'nt; T SLASH't]
553   | Nis'22 => [NT firstSegement'nt; T SLASH't; NT data'nt; T SLASH't; NT domain'nt
          ; T SLASH't]
554   | Nis'23 => []
555   | Nis'24 => [T SLASH't; NT firstSegement'nt; T SLASH't; NT data'nt; T SLASH't;
          NT domain'nt; T SLASH't]
556   | Nis'25 => [NT replicationParameter'nt; T STORAGE't; T SLASH't; NT rootDomain'
          nt; T SLASH't]
557   | Nis'26 => [NT dataName'nt; NT replicationParameter'nt; T STORAGE't; T SLASH't;
           NT rootDomain'nt; T SLASH't]
558   end.
559 Extract Constant past_symb_of_non_init_state => "fun _ -> assert false".
560
561 Definition state_set_1 (s:state) : bool :=
562   match s with
563   | Init Init'0 => true
```

```
564   | _ => false
565   end.
566 Extract Inlined Constant state_set_1 => "assert false".
567
568 Definition state_set_2 (s:state) : bool :=
569   match s with
570   | Ninit Nis'1 => true
571   | _ => false
572   end.
573 Extract Inlined Constant state_set_2 => "assert false".
574
575 Definition state_set_3 (s:state) : bool :=
576   match s with
577   | Ninit Nis'3 => true
578   | _ => false
579   end.
580 Extract Inlined Constant state_set_3 => "assert false".
581
582 Definition state_set_4 (s:state) : bool :=
583   match s with
584   | Ninit Nis'4 => true
585   | _ => false
586   end.
587 Extract Inlined Constant state_set_4 => "assert false".
588
589 Definition state_set_5 (s:state) : bool :=
590   match s with
591   | Ninit Nis'5 => true
592   | _ => false
593   end.
594 Extract Inlined Constant state_set_5 => "assert false".
595
596 Definition state_set_6 (s:state) : bool :=
597   match s with
598   | Ninit Nis'6 => true
599   | _ => false
600   end.
601 Extract Inlined Constant state_set_6 => "assert false".
602
603 Definition state_set_7 (s:state) : bool :=
604   match s with
605   | Ninit Nis'8 => true
606   | _ => false
607   end.
608 Extract Inlined Constant state_set_7 => "assert false".
609
610 Definition state_set_8 (s:state) : bool :=
611   match s with
612   | Ninit Nis'9 => true
613   | _ => false
614   end.
615 Extract Inlined Constant state_set_8 => "assert false".
616
617 Definition state_set_9 (s:state) : bool :=
618   match s with
619   | Ninit Nis'12 => true
620   | _ => false
621   end.
622 Extract Inlined Constant state_set_9 => "assert false".
```

```
623
624 Definition state_set_10 (s:state) : bool :=
625   match s with
626   | Ninit Nis'13 => true
627   | _ => false
628   end.
629 Extract Inlined Constant state_set_10 => "assert false".
630
631 Definition state_set_11 (s:state) : bool :=
632   match s with
633   | Ninit Nis'15 => true
634   | _ => false
635   end.
636 Extract Inlined Constant state_set_11 => "assert false".
637
638 Definition state_set_12 (s:state) : bool :=
639   match s with
640   | Ninit Nis'16 => true
641   | _ => false
642   end.
643 Extract Inlined Constant state_set_12 => "assert false".
644
645 Definition state_set_13 (s:state) : bool :=
646   match s with
647   | Ninit Nis'18 => true
648   | _ => false
649   end.
650 Extract Inlined Constant state_set_13 => "assert false".
651
652 Definition state_set_14 (s:state) : bool :=
653   match s with
654   | Ninit Nis'19 => true
655   | _ => false
656   end.
657 Extract Inlined Constant state_set_14 => "assert false".
658
659 Definition state_set_15 (s:state) : bool :=
660   match s with
661   | Ninit Nis'21 => true
662   | _ => false
663   end.
664 Extract Inlined Constant state_set_15 => "assert false".
665
666 Definition state_set_16 (s:state) : bool :=
667   match s with
668   | Ninit Nis'22 => true
669   | _ => false
670   end.
671 Extract Inlined Constant state_set_16 => "assert false".
672
673 Definition state_set_17 (s:state) : bool :=
674   match s with
675   | Ninit Nis'25 => true
676   | _ => false
677   end.
678 Extract Inlined Constant state_set_17 => "assert false".
679
680 Definition past_state_of_non_init_state (s:noninitstate) : list (state -> bool) :=
681   match s with
```

```
682 |   | Nis'1 => [ state_set_1 ]
683 |   | Nis'2 => [ state_set_2 ]
684 |   | Nis'3 => [ state_set_2; state_set_1 ]
685 |   | Nis'4 => [ state_set_3; state_set_2; state_set_1 ]
686 |   | Nis'5 => [ state_set_4; state_set_3; state_set_2; state_set_1 ]
687 |   | Nis'6 => [ state_set_5 ]
688 |   | Nis'7 => [ state_set_6 ]
689 |   | Nis'8 => [ state_set_6; state_set_5 ]
690 |   | Nis'9 => [ state_set_7; state_set_6; state_set_5 ]
691 |   | Nis'10 => [ state_set_8 ]
692 |   | Nis'11 => [ state_set_8; state_set_7; state_set_6; state_set_5 ]
693 |   | Nis'12 => [ state_set_5; state_set_4; state_set_3; state_set_2; state_set_1 ]
694 |   | Nis'13 => [ state_set_9 ]
695 |   | Nis'14 => [ state_set_10 ]
696 |   | Nis'15 => [ state_set_10; state_set_9 ]
697 |   | Nis'16 => [ state_set_11; state_set_10; state_set_9 ]
698 |   | Nis'17 => [ state_set_12 ]
699 |   | Nis'18 => [ state_set_12; state_set_11; state_set_10; state_set_9 ]
700 |   | Nis'19 => [ state_set_13; state_set_12; state_set_11; state_set_10;
        state_set_9 ]
701 |   | Nis'20 => [ state_set_14 ]
702 |   | Nis'21 => [ state_set_14; state_set_13; state_set_12; state_set_11;
        state_set_10; state_set_9 ]
703 |   | Nis'22 => [ state_set_15; state_set_14; state_set_13; state_set_12;
        state_set_11; state_set_10; state_set_9 ]
704 |   | Nis'23 => [ state_set_16 ]
705 |   | Nis'24 => [ state_set_16; state_set_15; state_set_14; state_set_13;
        state_set_12; state_set_11; state_set_10; state_set_9 ]
706 |   | Nis'25 => [ state_set_9; state_set_5; state_set_4; state_set_3; state_set_2;
        state_set_1 ]
707 |   | Nis'26 => [ state_set_17; state_set_9; state_set_5; state_set_4; state_set_3;
        state_set_2; state_set_1 ]
708 |   end.
709 | Extract Constant past_state_of_non_init_state => "fun _ -> assert false".
710 |
711 | Definition lookahead_set_1 : list terminal :=
712 |   [STRING't; STORAGE't; SLASH't; INT't; EOL't].
713 | Extract Inlined Constant lookahead_set_1 => "assert false".
714 |
715 | Definition lookahead_set_2 : list terminal :=
716 |   [SLASH't].
717 | Extract Inlined Constant lookahead_set_2 => "assert false".
718 |
719 | Definition lookahead_set_3 : list terminal :=
720 |   [EOL't].
721 | Extract Inlined Constant lookahead_set_3 => "assert false".
722 |
723 | Definition items_of_state_0 : list item :=
724 |   [ {| prod_item := Prod'main'0; dot_pos_item := 0; lookaheads_item :=
        lookahead_set_1 |} ].
725 | Extract Inlined Constant items_of_state_0 => "assert false".
726 |
727 | Definition items_of_state_1 : list item :=
728 |   [ {| prod_item := Prod'main'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_1 |};
729 |     {| prod_item := Prod'rootDomain'0; dot_pos_item := 0; lookaheads_item :=
          lookahead_set_2 |} ].
730 | Extract Inlined Constant items_of_state_1 => "assert false".
731 |
```

```
732 Definition items_of_state_2 : list item :=
733   [ {| prod_item := Prod'rootDomain'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_2 |} ].
734 Extract Inlined Constant items_of_state_2 => "assert false".
735
736 Definition items_of_state_3 : list item :=
737   [ {| prod_item := Prod'main'0; dot_pos_item := 2; lookaheads_item :=
        lookahead_set_1 |} ].
738 Extract Inlined Constant items_of_state_3 => "assert false".
739
740 Definition items_of_state_4 : list item :=
741   [ {| prod_item := Prod'main'0; dot_pos_item := 3; lookaheads_item :=
        lookahead_set_1 |} ].
742 Extract Inlined Constant items_of_state_4 => "assert false".
743
744 Definition items_of_state_5 : list item :=
745   [ {| prod_item := Prod'main'0; dot_pos_item := 4; lookaheads_item :=
        lookahead_set_1 |};
746     {| prod_item := Prod'replicationParameter'0; dot_pos_item := 0;
          lookaheads_item := lookahead_set_2 |} ].
747 Extract Inlined Constant items_of_state_5 => "assert false".
748
749 Definition items_of_state_6 : list item :=
750   [ {| prod_item := Prod'replicationFactor'0; dot_pos_item := 0; lookaheads_item
        := lookahead_set_2 |};
751     {| prod_item := Prod'replicationParameter'0; dot_pos_item := 1;
          lookaheads_item := lookahead_set_2 |} ].
752 Extract Inlined Constant items_of_state_6 => "assert false".
753
754 Definition items_of_state_7 : list item :=
755   [ {| prod_item := Prod'replicationFactor'0; dot_pos_item := 1; lookaheads_item
        := lookahead_set_2 |} ].
756 Extract Inlined Constant items_of_state_7 => "assert false".
757
758 Definition items_of_state_8 : list item :=
759   [ {| prod_item := Prod'replicationParameter'0; dot_pos_item := 2;
        lookaheads_item := lookahead_set_2 |} ].
760 Extract Inlined Constant items_of_state_8 => "assert false".
761
762 Definition items_of_state_9 : list item :=
763   [ {| prod_item := Prod'replicationIndex'0; dot_pos_item := 0; lookaheads_item :=
        lookahead_set_2 |};
764     {| prod_item := Prod'replicationParameter'0; dot_pos_item := 3;
          lookaheads_item := lookahead_set_2 |} ].
765 Extract Inlined Constant items_of_state_9 => "assert false".
766
767 Definition items_of_state_10 : list item :=
768   [ {| prod_item := Prod'replicationIndex'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_2 |} ].
769 Extract Inlined Constant items_of_state_10 => "assert false".
770
771 Definition items_of_state_11 : list item :=
772   [ {| prod_item := Prod'replicationParameter'0; dot_pos_item := 4;
        lookaheads_item := lookahead_set_2 |} ].
773 Extract Inlined Constant items_of_state_11 => "assert false".
774
775 Definition items_of_state_12 : list item :=
776   [ {| prod_item := Prod'dataName'0; dot_pos_item := 0; lookaheads_item :=
        lookahead_set_3 |};
```

```
777     {| prod_item := Prod'main'0; dot_pos_item := 5; lookaheads_item :=
          lookahead_set_1 |} ].
778 Extract Inlined Constant items_of_state_12 => "assert false".
779
780 Definition items_of_state_13 : list item :=
781   [ {| prod_item := Prod'dataName'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_3 |};
782     {| prod_item := Prod'domain'0; dot_pos_item := 0; lookaheads_item :=
          lookahead_set_2 |} ].
783 Extract Inlined Constant items_of_state_13 => "assert false".
784
785 Definition items_of_state_14 : list item :=
786   [ {| prod_item := Prod'domain'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_2 |} ].
787 Extract Inlined Constant items_of_state_14 => "assert false".
788
789 Definition items_of_state_15 : list item :=
790   [ {| prod_item := Prod'dataName'0; dot_pos_item := 2; lookaheads_item :=
        lookahead_set_3 |} ].
791 Extract Inlined Constant items_of_state_15 => "assert false".
792
793 Definition items_of_state_16 : list item :=
794   [ {| prod_item := Prod'data'0; dot_pos_item := 0; lookaheads_item :=
        lookahead_set_2 |};
795     {| prod_item := Prod'dataName'0; dot_pos_item := 3; lookaheads_item :=
          lookahead_set_3 |} ].
796 Extract Inlined Constant items_of_state_16 => "assert false".
797
798 Definition items_of_state_17 : list item :=
799   [ {| prod_item := Prod'data'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_2 |} ].
800 Extract Inlined Constant items_of_state_17 => "assert false".
801
802 Definition items_of_state_18 : list item :=
803   [ {| prod_item := Prod'dataName'0; dot_pos_item := 4; lookaheads_item :=
        lookahead_set_3 |} ].
804 Extract Inlined Constant items_of_state_18 => "assert false".
805
806 Definition items_of_state_19 : list item :=
807   [ {| prod_item := Prod'dataName'0; dot_pos_item := 5; lookaheads_item :=
        lookahead_set_3 |};
808     {| prod_item := Prod'firstSegement'0; dot_pos_item := 0; lookaheads_item :=
          lookahead_set_2 |} ].
809 Extract Inlined Constant items_of_state_19 => "assert false".
810
811 Definition items_of_state_20 : list item :=
812   [ {| prod_item := Prod'firstSegement'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_2 |} ].
813 Extract Inlined Constant items_of_state_20 => "assert false".
814
815 Definition items_of_state_21 : list item :=
816   [ {| prod_item := Prod'dataName'0; dot_pos_item := 6; lookaheads_item :=
        lookahead_set_3 |} ].
817 Extract Inlined Constant items_of_state_21 => "assert false".
818
819 Definition items_of_state_22 : list item :=
820   [ {| prod_item := Prod'dataName'0; dot_pos_item := 7; lookaheads_item :=
        lookahead_set_3 |};
```

```
821     {| prod_item := Prod'lastSegment'0; dot_pos_item := 0; lookaheads_item :=
           lookahead_set_3 |} ].
822 Extract Inlined Constant items_of_state_22 => "assert false".
823
824 Definition items_of_state_23 : list item :=
825   [ {| prod_item := Prod'lastSegment'0; dot_pos_item := 1; lookaheads_item :=
        lookahead_set_3 |} ].
826 Extract Inlined Constant items_of_state_23 => "assert false".
827
828 Definition items_of_state_24 : list item :=
829   [ {| prod_item := Prod'dataName'0; dot_pos_item := 8; lookaheads_item :=
        lookahead_set_3 |} ].
830 Extract Inlined Constant items_of_state_24 => "assert false".
831
832 Definition items_of_state_25 : list item :=
833   [ {| prod_item := Prod'main'0; dot_pos_item := 6; lookaheads_item :=
        lookahead_set_1 |} ].
834 Extract Inlined Constant items_of_state_25 => "assert false".
835
836 Definition items_of_state_26 : list item :=
837   [ {| prod_item := Prod'main'0; dot_pos_item := 7; lookaheads_item :=
        lookahead_set_1 |} ].
838 Extract Inlined Constant items_of_state_26 => "assert false".
839
840 Definition items_of_state (s:state) : list item :=
841   match s with
842   | Init Init'0 => items_of_state_0
843   | Ninit Nis'1 => items_of_state_1
844   | Ninit Nis'2 => items_of_state_2
845   | Ninit Nis'3 => items_of_state_3
846   | Ninit Nis'4 => items_of_state_4
847   | Ninit Nis'5 => items_of_state_5
848   | Ninit Nis'6 => items_of_state_6
849   | Ninit Nis'7 => items_of_state_7
850   | Ninit Nis'8 => items_of_state_8
851   | Ninit Nis'9 => items_of_state_9
852   | Ninit Nis'10 => items_of_state_10
853   | Ninit Nis'11 => items_of_state_11
854   | Ninit Nis'12 => items_of_state_12
855   | Ninit Nis'13 => items_of_state_13
856   | Ninit Nis'14 => items_of_state_14
857   | Ninit Nis'15 => items_of_state_15
858   | Ninit Nis'16 => items_of_state_16
859   | Ninit Nis'17 => items_of_state_17
860   | Ninit Nis'18 => items_of_state_18
861   | Ninit Nis'19 => items_of_state_19
862   | Ninit Nis'20 => items_of_state_20
863   | Ninit Nis'21 => items_of_state_21
864   | Ninit Nis'22 => items_of_state_22
865   | Ninit Nis'23 => items_of_state_23
866   | Ninit Nis'24 => items_of_state_24
867   | Ninit Nis'25 => items_of_state_25
868   | Ninit Nis'26 => items_of_state_26
869   end.
870 Extract Constant items_of_state => "fun _ -> assert false".
871
872 End Aut.
873
874
```

```
875 From MenhirLib Require Import Main.
876
877 Module Parser := Main.Make Aut.
878 Theorem safe:
879   Parser.safe_validator () = true.
880 Proof eq_refl true<:Parser.safe_validator () = true.
881
882 Theorem complete:
883   Parser.complete_validator () = true.
884 Proof eq_refl true<:Parser.complete_validator () = true.
885
886 Definition main := Parser.parse safe Aut.Init'0.
887
888 Theorem main_correct iterator buffer:
889   match main iterator buffer with
890   | Parser.Inter.Parsed_pr sem buffer_new =>
891     exists word,
892       buffer = Parser.Inter.app_str word buffer_new /\
893       inhabited (Gram.parse_tree (NT main'nt) word sem)
894   | _ => True
895   end.
896 Proof. apply Parser.parse_correct. Qed.
897
898 Theorem main_complete (iterator:nat) word buffer_end (output:       (ast)):
899   forall tree:Gram.parse_tree (NT main'nt) word output,
900   match main iterator (Parser.Inter.app_str word buffer_end) with
901   | Parser.Inter.Fail_pr => False
902   | Parser.Inter.Parsed_pr output_res buffer_end_res =>
903     output_res = output /\ buffer_end_res = buffer_end  /\
904     le (Gram.pt_size tree) iterator
905   | Parser.Inter.Timeout_pr => lt iterator (Gram.pt_size tree)
906   end.
907 Proof. apply Parser.parse_complete with (init:=Aut.Init'0); exact complete. Qed.
908
909 (*adding stuff to prove message sending *)
910
911
912 Inductive message : Type :=
913 |Replication
914 |Stop
915 .
916
917 Inductive data : Type :=
918 |Data
919 .
920
921 (* Function used to decrease the replication factor *)
922 Definition decrease_replication_factor := fun x => x - 1.
923
924
925 (* Function to determine when a replication is needed according to the replication
        factor *)
926 Definition replication_needed (n:nat) : bool :=
927    match  decrease_replication_factor n with
928   | O => false
929   | _ => true
930   end.
931
932 (* send the correct message according to the need for replication or not*)
```

```
933 Definition send_message(a:ast): message :=
934    match  a with
935    | Main c d e => match  d with
936                      | Parameters n m => match  replication_needed m with
937                                           | true =>  Replication
938                                           | false => Stop
939                                         end
940                    end
941 end.
942
943 (* Always send a data request *)
944 Definition send_data_request(a:ast): data :=
945    match  a with
946    | Main c d e => Data
947 end.
948
949
950 (*This normally is created by coq when defining the type message, but don't know
        why it's not.
951   So Copy paste from another terminal where it's created, the result of Check
        message_ind
952 *)
953 Axiom message_ind
954      : forall P : message -> Prop, P Replication -> P Stop -> forall m : message,
           P m
955 .
956
957 (* any message type is necessarily one of the least known.*)
958
959 Theorem message_equal:
960 forall m :message,
961       m = Replication \/ m = Stop.
962 Proof.
963 intro m. pattern m.
964 apply message_ind.
965 induction m. auto. auto. auto.
966 Qed.
967
968
969 (* If a word is in the ast(accepted by the parser), then a message is sent and a
        data request is sent*)
970 Theorem accepted_implies_message iterator buffer (m:       (message)):
971   match main iterator buffer with
972   | Parser.Inter.Parsed_pr sem buffer_new =>
973     forall sem:ast,
974          (send_message sem = Replication \/ send_message sem = Stop)  /\ (
               send_data_request sem = Data)
975 | _ => True
976   end.
977
978
979 Proof.
980  intros.
981  induction main.
982  auto. auto.
983  intro sem.
984  constructor.
985  pattern sem.
986  induction sem.
```

```
987 apply message_equal.
988 pattern sem.
989 induction sem.
990 constructor.
991 Qed.
```

Appendix B

Automation scripts

B.1 Framework installation

```
#INSTALL NODEJS

#download it from https://nodejs.org/en/download/
#unzip the binary archive to /usr/local/lib/nodejs
sudo mkdir -p /usr/local/lib/nodejs
sudo tar -xJvf node-v10.15.3-linux-x64.tar.xz -C /usr/local/lib/nodejs

#set the environment variable ~/.profile, add below to the end
export PATH=/usr/local/lib/nodejs/node-v10.15.3-linux-x64/bin:$PATH

#test installation
node -v
npm version
npx -v

#(optionnal) to create sudo link (because "export" is not persistent)
sudo ln -s /usr/local/lib/nodejs/node-v10.15.3-linux-x64/bin/node /usr/bin/node
sudo ln -s /usr/local/lib/nodejs/node-v10.15.3-linux-x64/bin/npm /usr/bin/npm
sudo ln -s /usr/local/lib/nodejs/node-v10.15.3-linux-x64/bin/npx /usr/bin/npx

#download the framework
git clone https://github.com/mistersound/ndfs-evaluation.git

#create two directory for the app
cd ndfs-evaluation/
mkdir conf  #for conf files
mkdir store #to stock replication files

#make shure "nfd-start" is running
#client advertising for /lacl

#ON NODE 1
#the advertising must be /upec/storage on nodes
node storage.js

#ON CLIENT
#ndnputchunks fileName < file
#node client.js /domain/storage/repF/repI/fileName
ndnputchunks /lacl/data/0/9 < hello.txt
node client.js /upec/storage/2/2/lacl/data/0/9
```

B.2 Install NDN node from source

```
1
2  //install git
3  apt-get install git
4
5  //download ndn-cxx (main dependence of NFD)
6  git clone https://github.com/named-data/ndn-cxx
7
8  //download NFD
9  git clone --recursive https://github.com/named-data/NFD
10
11 //download ndn-tools (ping, dump...)
12 git clone https://github.com/named-data/ndn-tools
13
14 //install ndn-cxx library and its requirements
15 apt-get install build-essential pkg-config libboost-all-dev libsqlite3-dev libssl-
       dev libpcap-dev libsystemd-dev
16
17 //(optional) install valgrind
18 apt-get install valgrind valgrind-dbg kcachegrind alleyoop valkyrie
19
20 //to build manpages and API documentation
21 apt-get install doxygen graphviz python-sphinx
22
23 //to build ndn-cxx
24 cd ndn-cxx/
25 ./waf configure
26 ./waf
27 sudo ./waf install
28
29 //if ndn-cxx library is installed into a non-standard path
30 export PKG_CONFIG_PATH=/user/local/lib/pkgconfig/
31
32 #to build NFD
33 cd NFD/
34 ./waf configure
35 ./waf
36 sudo ./waf install
37 sudo ldconfig
38
39 #to build ndn-tools
40 cd ndn-tools/
41 ./waf configure
42 ./waf
43 sudo ./waf install
44 sudo ldconfig
45
46 #to create proper config file
47 cp /usr/local/etc/ndn/nfd.conf.sample /usr/local/etc/ndn.nfd.conf
```

B.3 Install NLSR from source

```
1
2  #NFD and its requirements must be install
3  #INSTALLATION PART
4
5  #install requirements
6  git clone https://github.com/named-data/ChronoSync
7  cd ChronoSync/
8  ./waf configure
9  ./waf
10 sudo ./waf install
11
12 git clone https://github.com/named-data/PSync
13 cd PSync/
14 ./waf configure
15 ./waf
16 sudo ./waf install
17
18 #install NLSR (Named Data Link State Routing)
19 git clone https://github.com/named-data/NLSR
20 cd NLSR/
21 ./waf configure
22 ./waf
23 sudo ./waf install
24
25 sudo ldconfig
26
27 #to create dir for nlsr file (include nlsr.conf maked by nlsr)
28 mkdir /var/lib/nlsr
29
30 #to get rights to nlsr to write in
31 chmod 777 /var/lib/nlsr
32
33 #SETTING UP THE SECURITY
34
35 #generate the router key
36 ndnsec-key-gen /ndn/edu/uaslp/%C1.Router/routerX > routerX.key
37
38 #generate the certificate for the router key
39 ndnsec-cert-dump -i /ndn/edu/uaslp/%C1.Router/routerX > routerX.cert
40
41 #install the router certificate
42 ndnsec-cert-install -f routerX.cert
43
44 #to verify that the certificates have been installed
45 ndnsec-list
46
47 #CONFIGURING NFD
48
49 #test physical network configuration
50 ping 192.168.10.2
51
52 #remember to start "nfd-start" in an other terminal
53 nfd-status
54
55 #configure each face that a computer uses to connect to a neighboring computer
56 nfdc face create udp4://192.168.10.2
57
58 #to display the face id
```

```
nfdc face list

#to verify the status of the face
nfdc face show id 265 #because <face-id> = 265

#SETTING UP THE CONFIGURATION FILE

#see https://named-data.net/doc/NLSR/current/ROUTER-CONFIG.html
touch nlsr.conf

#STARTING NLSR

#general command (recommended to open in other terminal)
nlsr -f nlsr.conf

#to verify what is NLSR doing
export NDN_LOG=nlsr.*=TRACE && nlsr -f nlsr.conf

#TURNING EVERYTHING OFF

#1 - stop nlsr processus by pressing Ctrl-C
#2 - destroy the face to the remote computers
nfdc face destroy 265

#3 - stop NFD
nfd-stop
```

B.4 Deploy Hadoop cluster

```
#!/bin/bash
namenode=192.168.20.254
d1=192.168.20.1
d2=192.168.20.2
d3=192.168.20.3
d4=192.168.20.4
d5=192.168.20.5
d6=192.168.20.6
d7=192.168.20.7
d8=192.168.20.8
d9=192.168.20.9
d10=192.168.20.10
d11=192.168.20.11
d12=192.168.20.12
d13=192.168.20.13
d14=192.168.20.14
d15=192.168.20.15
d16=192.168.20.16

##INSTALLATION

#check where java is installed
path=`which java`

#remove the last two components (bin/java) of the path
path=`readlink -f $path | rev | cut -d / -f 3- | rev`/

```

```
28 #set the environment variable (add this line to ~/.bashrc to get it permanently)
29 export JAVA_HOME=$path
30
31 #download the Hadoop 2.9.1 version
32 wget -O ~/hadoop-2.9.1.tar.gz http://mirror.ibcp.fr/pub/apache/hadoop/common/
       hadoop-2.9.1/hadoop-2.9.1.tar.gz
33
34 #extract files
35 tar xzf ~/hadoop-2.9.1.tar.gz
36
37 ##CONFIGURATION
38
39 #add property to core-site.xlm
40 file=~/hadoop-2.9.1/etc/hadoop/core-site.xml
41 head -n -3 $file > temp.txt
42 mv temp.txt $file
43 echo "
44 <configuration>
45     <property>
46         <name>fs.default.name</name>
47         <value>hdfs://$namenode:9000</value>
48     </property>
49 </configuration>" >> $file
50
51 #create directory who contains data
52 mkdir data
53 mkdir data/namenode
54 mkdir data/datanode
55
56 #edit hdfs-site.xml
57 file=~/hadoop-2.9.1/etc/hadoop/hdfs-site.xml
58 head -n -4 $file > temp.txt
59 mv temp.txt $file
60 echo "
61 <configuration>
62     <property>
63         <name>dfs.namenode.name.dir</name>
64         <value>home/user/data/nameNode</value>
65     </property>
66
67     <property>
68         <name>dfs.datanode.data.dir</name>
69         <value>home/user/data/dataNode</value>
70     </property>
71
72     <property>
73         <name>dfs.replication</name>
74         <value>3</value>
75     </property>
76
77  <property>
78   <name>dfs.namenode.datanode.registration.ip-hostname-check</name>
79   <value>false</value>
80  </property>
81 </configuration>" >> $file
82
83 #the file "workers" is used to start required daemons on all nodes
84 file=~/hadoop-2.9.1/etc/hadoop/workers
85 echo "$d1
```

```
$d2
$d3
$d4
$d5
$d6
$d7
$d8
$d9
$d10
$d11
$d12
$d13
$d14
$d15
$d16" > $file

#duplicate config files on each node
ssh-keygen
for node in $d1 $d2 $d3 $d4 $d5 $d6 $d7 $d8 $d9 $d10 $d11 $d12 $d13 $d14 $d15 $d16
    ; do
        ssh-copy-id -i ~/.ssh/id_rsa.pub user@$node
        ssh-add
        scp hadoop-*.tar.gz $node:~
        ssh user@$node 'tar -xzf hadoop-2.9.1.tar.gz'
        scp ~/hadoop-2.9.1/etc/hadoop/* $node:~/hadoop-2.9.1/etc/hadoop/;
done

#HDFS needs to be formatted like any classical file system
~/hadoop-2.9.1/bin/hdfs namenode -format
```

Appendix C

NMapReduce WordCount script (JavaScript)

```
1
2  // mapper function
3  var map = function (str) {
4      return str.split(" ").map(function(str) {
5          return [str, [1]];
6      }).filter(function(array){
7        return array[1][0] > 0;
8      });
9  }
10
11 // Shuflle function
12 var shuffle = function (arr) {
13   var sortedArray = arr.sort();
14   var size = sortedArray.length - 1;
15   for(var i = 0; i < size; i++) {
16     if (sortedArray[i][0] == sortedArray[i+1][0]) {
17       sortedArray[i][1].push(1);
18       sortedArray.splice(i+1, 1);
19       size--;
20       i--;
21     }
22   }
23   return sortedArray
24 }
25
26 // reducer function
27 var reduce = function (arr){
28   for (var j = 0; j < shuffledArray.length; j++)
29   {
30     shuffledArray[j][1] = shuffledArray[j][1].reduce(function(x, y) {
31       return x + y;
32     });
33   }
34 }
```

Appendix D

MapReduce WordCount source code (Java)

D.1 Mapper Class Code

```
1  public static class Map extends MapReduceBase implements Mapper {
2      private final static IntWritable one = new IntWritable(1);
3      private Text word = new Text();
4      public void map(LongWritable key, Text value, OutputCollector output, Reporter
           reporter)
5      throws IOException {
6          String line = value.toString();
7          StringTokenizer tokenizer = new StringTokenizer(line);
8          while (tokenizer.hasMoreTokens()) {
9              word.set(tokenizer.nextToken());
10             output.collect(word, one);
11         }
12     }
13 }
```

D.2 Reducer Class Code

```
1
2  public static class Reduce extends MapReduceBase implements Reducer {
3
4      public void reduce(Text key, Iterator values, OutputCollector output,
5          Reporter reporter) throws IOException {
6
7          int sum = 0;
8          while (values.hasNext()) {
9              sum += values.next().get();
10         }
11         output.collect(key, new IntWritable(sum));
12     }
13 }
```

D.3 Main Class Code

```
1
2  public static void main(String[] args) throws Exception {
3    JobConf conf = new JobConf(WordCount.class);
4    conf.setJobName("WordCount");
5
6    conf.setOutputKeyClass(Text.class);
7    conf.setOutputValueClass(IntWritable.class);
8
9    conf.setMapperClass(Map.class);
10   //conf.setCombinerClass(Reduce.class);
11   conf.setReducerClass(Reduce.class);
12
13   conf.setInputFormat(TextInputFormat.class);
14   conf.setOutputFormat(TextOutputFormat.class);
15
16   FileInputFormat.setInputPaths(conf, new Path(args[0]));
17   FileOutputFormat.setOutputPath(conf, new Path(args[1]));
18
19   JobClient.runJob(conf);
20  }
21 }
```

Appendix E

R script for simulation response computation

```
# install.packages('ggplot2')
library(ggplot2)

library(plyr)
library(stringr)
library(dplyr)

n = 57

df <- data.frame(experiment = numeric(n+1), ReplicationTime  =
                   numeric(n+1), nbNodeBeforeRes = numeric(n+1), RetrieveTime =
                   numeric(n+1), TotalReplication = numeric(n+1) )

#import files

for(exp in 0:n){

  pathElems <- c("~/data/dump-trace-", ".txt")
  #pathElems <- c("/home/lacl/Dropbox/simulation/dump-trace-", ".txt")



  file <- paste(pathElems, collapse=toString(exp))

  if (file.exists(file) && file.size(file) > 0){ #check file existence
    data = read.table(file, header=T, sep=";")







    # Y1 Replication time


    storLine <- grep("storage", data$Name) #store storages

    stopLine <- grep("stop", data$Name) #store stop

```

```
40      part1 <- data[storLine,]
41      part2 <- data[stopLine,]
42
43      #global <- rbind(part1,part2)
44
45      #results <- cbind(part,str_split_fixed(part$V3, "/", Inf))
46
47
48      #res <- c("Time","ID","Value") # will generate a warning due to NA when
            looping
49      res1 <- data.frame(character(),character(),character())
50      res2 <- data.frame(character(),character(),character())
51      for(i in 1:nrow(part1)){
52
53        #keep only rows corresponding to start and stop replication command from an
              admin node lacl-AdminID
54        if(length(grep(paste0("lacl",part1[i,2],"/"),part1[i,4])) == 1){
55          res1 <- rbind(res1, part1[i,])
56          #res <- rbind(res, dump.trace.1[grep(paste0("lacl",i),part[i,3]),])
57        }
58
59      }
60
61      for(i in 1:nrow(part2)){
62
63        #keep only rows corresponding to start and stop replication command from an
              admin node lacl-AdminID
64        if(length(grep(paste0("lacl",part2[i,2],"/"),part2[i,4])) == 1){
65          res2 <- rbind(res2, part2[i,])
66          #res <- rbind(res, dump.trace.1[grep(paste0("lacl",i),part[i,3]),])
67        }
68
69      }
70
71      #res <- res[-1,] #remove the first row wich contains the NA
72
73
74    if(nrow(res2)!=0){
75      X <- split(res1$Time, res1$Node) #keep only the min and max time which
            represent the start and completion time
76      Y <- split(res2$Time, res2$Node)
77      val <- data.frame(repTime = numeric(min(length(X),length(Y)))) #value for the
            replication Time for every request in the experiment
78      for(i in 1:min(length(X),length(Y))){
79
80        #maxi = as.numeric(max(X[[i]]))
81        #mini = as.numeric(min(X[[i]]))
82        #dif = maxi-mini
83        # print(dif)
84        val$repTime[i] <- as.numeric(min(Y[[i]])) - as.numeric(min(X[[i]]))
85
86        moy = mean(val$repTime)
87      }
88    }else
89      {
90        moy = NA
91     }
92
93
```

```
94
95
96
97      df$experiment[exp+1] = exp
98      df$ReplicationTime[exp+1] = moy
99
100     # Mean distance before an interest reach a Storage
101
102     dataInterest <- filter(data, !grepl("Interest", data$Name)) #keep only lines
            from data request interest
103
104     dataInterest <- filter(dataInterest, !grepl("data", dataInterest$Type)) #added
            4/21 remove the data lines
105
106     data_count <- ddply(dataInterest, c("Name"), summarise,
107                         nbNodes=length(Name))
108
109     t = mean(as.numeric(data_count$nbNodes))
110
111     df$nbNodeBeforeRes[exp+1] = round(t)
112
113     #begin mean retrieve
114     dat <- filter(data, !grepl("stop", data$Name))
115     dat <- filter(dat, !grepl("storage", dat$Name))
116     dat <- filter(dat, !grepl("heartbeat", dat$Name))
117
118     v <- dat[c(1,3,4)]
119
120     re <- data.frame(character(),character(),character())
121     re <- rbind(re,v[1,])
122
123
124     current <- v[1,2]
125     for(i in 1:nrow(v)){
126
127       if(v[i,2]!=current){
128
129         re <- rbind(re,v[i,])
130         current <- v[i,2]
131       }
132
133     }
134
135     cmp <- data.frame(rep=numeric(nrow(re)))
136     for(i in 1:nrow(re)){
137       if(i %%2 == 0) {
138         cmp$rep[i]=NA
139         next()
140       }
141       cmp$rep[i] = re[1+i,1] - re[i,1]
142
143     }
144     cmp <- cmp[complete.cases(cmp), ]
145
146     #end mean retrieve
147
148     a = mean(cmp)
149
150
```

```
151       df$RetrieveTime[exp+1] = a
152       #TotalReplication Total number of  replication requests
153       part1 = part1[grep("interest",part1$Type),]
154       tDt = as.numeric(length(unique(part1$Name)))
155       df$TotalReplication[exp+1]  = tDt
156
157
158
159     }# end if
160     else
161     {
162       df$experiment[exp+1] = exp
163       df$ReplicationTime[exp+1] = NA
164       df$nbNodeBeforeRes[exp+1] = NA
165       df$RetrieveTime[exp+1] = NA #
166       df$TotalReplication[exp+1] = NA
167
168     }
169
170 }
171
172
173 for(exp in 0:n){
174
175   #pathElemsRate <- c("/home/lacl/work_ndn/dump-trace-", ".txt")
176     pathElemsRate <- c("~/data/rate-trace-", ".txt")
177
178     fileRate <- paste(pathElemsRate, collapse=toString(exp))
179
180     if (file.exists(fileRate) && file.size(fileRate) > 0){ #check file existence
181       dataRate = read.delim(fileRate, header=T, sep="\t")
182
183       dataRate$Node = factor(dataRate$Node)
184       dataRate$FaceId <- factor(dataRate$FaceId)
185       dataRate$Kilobits <- dataRate$Kilobytes * 8
186       dataRate$Type = factor(dataRate$Type)
187       dataRate$Time = factor(dataRate$Time)
188
189       idata = subset(dataRate, Type %in% c("InInterests", "OutInterests",
190                                            "InData", "OutData"))
191       idata = subset(idata, FaceDescr %in% c("appFace://"))
192
193       df$experiment[exp+1] = exp
194
195       df$NbPackets[exp+1] <-  sum(idata$Packets) #number of packets per sec
196
197       df$Rate[exp+1] <-  sum(idata$Kilobits) #Kilobits/s
198
199       #df$ReplicationFactor[exp+1] <- i
200
201       ################################################################################
202
203       Test = subset(dataRate, Type %in%
204                       c("InSatisfiedInterests","OutSatisfiedInterests","
                            InTimedOutInterests","OutTimedOutInterests"))
205       Test = subset(Test, FaceDescr %in% c("appFace://"))
206       Test = subset(Test,Packets>0)
207
208
```

```
209
210     out = subset(idata,Packets>0)
211
212     #inInterest
213     inInterest = subset(out,Type %in% c("InInterests"))
214     df$nbInInterest[exp+1] = length(inInterest$Type)
215
216     #outInterest
217     outInterest = subset(out,Type %in% c("OutInterests"))
218     df$nbOutInterest[exp+1] = length(outInterest$Type)
219
220
221     #outData
222     outData = subset(out,Type %in% c("OutData"))
223     df$nbOutData[exp+1] = length(outData$Type)
224
225     #inData
226     inData = subset(out,Type %in% c("InData"))
227     df$nbInData[exp+1] = length(inData$Type)
228
229     #inSatisfiedInterests
230     inSati = subset(Test,Type %in% c("InSatisfiedInterests"))
231     length(inSati$Type)
232
233     #outSatisfiedInterests
234     outSati = subset(Test,Type %in% c("OutSatisfiedInterests"))
235     length(outSati$Type)
236
237     #inTimedOutInterests
238     inTime = subset(Test,Type %in% c("InTimedOutInterests"))
239     length(inTime$Type)
240
241     #out TimedOutInterests
242     outTime = subset(Test,Type %in% c("OutTimedOutInterests"))
243     length(outTime$Type)
244
245
246     ######################################3
247
248   }#end if
249   else
250   {
251     df$experiment[exp+1] = exp
252     df$NbPackets[exp+1] = NA
253     df$Rate[exp+1] = NA
254     df$nbInInterest[exp+1] = NA
255     df$nbOutInterest[exp+1] = NA
256     df$nbOutData[exp+1] = NA
257     df$nbInData[exp+1] = NA
258
259   }
260
261 }# end for
262
263 write.csv(x = df,file = "~/result.csv", sep = ";")
```

Bibliography

[1] J. Wang, C. Peng, C. Li, E. Osterweil, R. Wakikawa, P.-c. Cheng, and L. Zhang, "Implementing instant messaging using named data," in *Proceedings of the Sixth Asian Internet Engineering Conference*. ACM, 2010, pp. 40–47.

[2] J. Burke, "Video streaming over named data networking," *E-LETTER*, 2013.

[3] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[4] J. Manyika, "Big data: The next frontier for innovation, competition, and productivity," *http://www. mckinsey. com/Insights/MGI/Research/Technology_ and_ Innovation/Big_ data_ The_ next_ frontier_ for_ innovation*, 2011.

[5] T. H. Davenport and J. G. Harris, *Analytics and Big Data: The Davenport Collection (6 Items)*. Harvard Business Review Press, 2014.

[6] B. Staff, "Big data isn't a concept — it's a problem to solve," https://datascience.berkeley.edu/blog/what-is-big-data/, accessed: 2019-10-20.

[7] E. T. from the arXiv, "The big data conundrum: How to define it?" https://www.technologyreview.com/s/519851/the-big-data-conundrum-how-to-define-it/, accessed: 2019-10-20.

[8] GARTNER, "Big data - gartner glossary," https://www.gartner.com/en/information-technology/glossary/big-data, accessed: 2019-10-20.

[9] Talend, "Stream processing defined - talend real-time open source data integration software," https://www.talend.com/resources/stream-processing-defined/, accessed: 2019-10-20.

[10] J. Kreps, "Questioning the lambda architecture," https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[11] A. Foundation, "Apache spark- unified analytics engine for big data," https://spark.apache.org/, accessed: 2019-10-20.

[12] ——, "Apache flink — stateful computations over data streams," https://flink.apache.org/, accessed: 2019-10-20.

[13] J. L. Potter, *The massively parallel processor*. MIT press, 1985.

[14] J. Eiloart, "Big data in banking: How bnp paribas is answering questions with its data," https://www.globalbankingandfinance.com/big-data-in-banking-how-bnp-paribas-is-answering-questions-with-its-data/, accessed: 2019-10-20.

[15] M. Terekhova, "Jpmorgan takes ai use to the next level," https://www.businessinsider.com/jpmorgan-takes-ai-use-to-the-next-level-2017-8, accessed: 2019-10-20.

[16] S. Kessler, "Mckinsey: Robots can do about 30won't necessarily take jobs," https://qz.com/1034873/mckinsey-robots-can-do-about-30-of-the-work-at-banks-but-they-wont-necessarily-take-jobs/, accessed: 2019-10-20.

[17] Z. Peng, "Stocks analysis and prediction using big data analytics," in *2019 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*. IEEE, 2019, pp. 309–312.

[18] B. Marr, "Big data in healthcare: Paris hospitals predict admission rates using machine learning," https://www.forbes.com/sites/bernardmarr/2016/12/13/big-data-in-healthcare-paris-hospitals-predict-admission-rates-using-machine-learning/#1521146079a2, accessed: 2019-10-20.

[19] H. Elshazly, A. T. Azar, A. El-Korany, and A. E. Hassanien, "Hybrid system for lymphatic diseases diagnosis," in *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2013, pp. 343–347.

[20] E. Rees, V. Ng, P. Gachon, A. Mawudeku, D. McKenney, J. Pedlar, D. Yemshanov, J. Parmely, and J. Knox, "Early detection and prediction of infectious disease outbreaks," *CCDR*, vol. 45, p. 5, 2019.

[21] S. Akter and S. F. Wamba, "Big data analytics in e-commerce: a systematic review and agenda for future research," *Electronic Markets*, vol. 26, no. 2, pp. 173–194, 2016.

[22] L. Carrel, "Listening to customers makes big change at expedia," https://www.investors.com/news/management/leaders-and-success/chad-richison-paycom-turns-trending-needs-big-business/, accessed: 2019-10-20.

[23] MapR, "Government use cases," https://mapr.com/solutions/industry/government-use-cases/, accessed: 2019-10-20.

[24] N. A. Ghani, S. Hamid, I. A. T. Hashem, and E. Ahmed, "Social media big data analytics: A survey," *Computers in Human Behavior*, vol. 101, pp. 417–428, 2019.

[25] M. Moessner, J. Feldhege, M. Wolf, and S. Bauer, "Analyzing big data in social media: text and network analyses of an eating disorder forum," *International Journal of Eating Disorders*, vol. 51, no. 7, pp. 656–667, 2018.

[26] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7. Portland, OR, 2000.

[27] D. Agrawal, A. El Abbadi, S. Antony, and S. Das, "Data management challenges in cloud computing infrastructures," in *International Workshop on Databases in Networked Information Systems*. Springer, 2010, pp. 1–10.

[28] T. A. S. Foundation, "Apache hbase," http://hbase.apache.org/, accessed: 2019-10-20.

[29] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[30] Y. Demchenko, C. De Laat, and P. Membrey, "Defining architecture components of the big data ecosystem," in *2014 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2014, pp. 104–112.

[31] S. Bergamaschi, F. Guerra, M. Orsini, C. Sartori, and M. Vincini, "A semantic approach to etl technologies," *Data & Knowledge Engineering*, vol. 70, no. 8, pp. 717–731, 2011.

[32] A. Foundation, "Apache flume," https://flume.apache.org/, accessed: 2019-10-20.

[33] C. White, "Data integration: Using etl, eai, and eii tools to create an integrated enterprise," *Business Intelligence Journal*, vol. 10, no. I, 2005.

[34] J. S. Saltz, "The need for new processes, methodologies and tools to support big data teams and improve big data project effectiveness," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 2066–2071.

[35] D. Loshin, *Big data analytics: from strategic planning to enterprise integration with tools, techniques, NoSQL, and graph*. Elsevier, 2013.

[36] Talend, "Talend - a cloud data integration leader (modern etl)," https://www.talend.com/, accessed: 2019-10-20.

[37] T. A. S. Foundation, "Apache sqoop," http://sqoop.apache.org/, accessed: 2019-10-20.

[38] TIBCO, "Tibco cloud integration - connect," https://www.tibco.com/products/cloud-integration/connect?source=scribesoft.com, accessed: 2019-10-20.

[39] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi, "Big data and its technical challenges," *Communications of the ACM*, vol. 57, no. 7, pp. 86–94, 2014.

[40] J. Zhao and J. Pjesivac-Grbovic, "Mapreduce: The programming model and practice," 2009, tutorial. [Online]. Available: https://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf

[41] A. Foundation, "Apache hive tm," https://hive.apache.org/, accessed: 2019-10-20.

[42] A. Cuzzocrea, "Privacy and security of big data: current challenges and future research perspectives," in *Proceedings of the First International Workshop on Privacy and Secuirty of Big Data*. ACM, 2014, pp. 45–47.

[43] V. N. Inukollu, S. Arsi, and S. R. Ravuri, "Security issues associated with big data in cloud computing," *International Journal of Network Security & Its Applications*, vol. 6, no. 3, p. 45, 2014.

[44] Y. Li, K. Gai, L. Qiu, M. Qiu, and H. Zhao, "Intelligent cryptography approach for secure distributed big data storage in cloud computing," *Information Sciences*, vol. 387, pp. 103–115, 2017.

[45] V. C. Hu, T. Grance, D. F. Ferraiolo, and D. R. Kuhn, "An access control scheme for big data processing," in *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2014, pp. 1–7.

[46] R. Zuech, T. M. Khoshgoftaar, and R. Wald, "Intrusion detection and big heterogeneous data: a survey," *Journal of Big Data*, vol. 2, no. 1, p. 3, 2015.

[47] T. A. S. Foundation, "Apache ranger," https://ranger.apache.org/.

[48] ——, "Apache knox," https://knox.apache.org/.

[49] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[50] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," 2003.

[51] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[52] E. Levy and A. Silberschatz, "Distributed file systems: Concepts and examples," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 321–374, 1990.

[53] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The itc distributed file system: Principles and design," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, pp. 35–50, 1985.

[54] B. Jena, M. K. Gourisaria, S. S. Rautaray, and M. Pandey, "Name node performance enlarging by aggregator based hadoop framework," in *I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC), 2017 International Conference on*. IEEE, 2017, pp. 112–116.

[55] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[56] S. Gupta and V. Giri, "Ensure high availability of data lake," in *Practical Enterprise Data Lake Insights*. Springer, 2018, pp. 261–295.

[57] J. Kumari, T. Biswas, and S. Vuppala, "Enhancing replica synchronization in hadoop distributed file system," in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2018, pp. 1–5.

[58] L. Jiang, B. Li, and M. Song, "The optimization of hdfs based on small files," in *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*. IEEE, 2010, pp. 912–915.

[59] J. A. Sacristán and T. Dilla, "No big data without small data: learning health care systems begin and end with the individual patient," *Journal of evaluation in clinical practice*, vol. 21, no. 6, pp. 1014–1017, 2015.

[60] M. Raynal, *Distributed algorithms for message-passing systems*. Springer, 2013, vol. 500.

[61] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley Reading, 2000, vol. 11.

[62] P. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[63] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in openmp," in *International Workshop on OpenMP*. Springer, 2007, pp. 1–12.

[64] Z. Yang and K. Duddy, "Corba: a platform for distributed object computing," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 2, pp. 4–31, 1996.

[65] J. McCarthy, S. Russell, T. P. Hart, M. Levin, A. Arc, and C. L. Clojure, "Lisp programming language," 1985.

[66] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[67] V. Jacobson, "A new way to look at networking," *Google Tech Talk*, vol. 30, 2006.

[68] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.

[69] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 181–192, 2007.

[70] N. Fotiou, P. Nikander, D. Trossen, and G. C. Polyzos, "Developing information networking further: From psirp to pursuit," in *International Conference on Broadband Communications, Networks and Systems*. Springer, 2010, pp. 1–13.

[71] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl, "Network of information (netinf)–an information-centric networking architecture," *Computer Communications*, vol. 36, no. 7, pp. 721–735, 2013.

[72] V. S. Mai, S. Ioannidis, D. Pesavento, and L. Benmohamed, "Optimal cache allocation under network-wide capacity constraint," in *2019 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2019, pp. 816–820.

[73] D. Nguyen, K. Sugiyama, and A. Tagami, "Congestion price for cache management in information-centric networking," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2015, pp. 287–292.

[74] M. Dehghan, L. Massoulie, D. Towsley, D. S. Menasché, and Y. C. Tay, "A utility optimization approach to network cache design," *IEEE/ACM Transactions on Networking*, 2019.

[75] T. Berners-Lee, R. Fielding, L. Masinter *et al.*, "Uniform resource identifiers (uri): Generic syntax," 1998.

[76] Z. Zhang, Y. Yu, H. Zhang, E. Newberry, S. Mastorakis, Y. Li, A. Afanasyev, and L. Zhang, "An overview of security support in named data networking," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 62–68, 2018.

[77] V. Lehman, A. Gawande, B. Zhang, L. Zhang, R. Aldecoa, D. Krioukov, and L. Wang, "An experimental investigation of hyperbolic routing with a smart forwarding plane in ndn," in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. IEEE, 2016, pp. 1–10.

[78] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 2013, pp. 15–20.

[79] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos *et al.*, "Named data networking (ndn) project," *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.

[80] H. Zhang, Z. Wang, C. Scherb, C. Marxer, J. Burke, L. Zhang, and C. Tschudin, "Sharing mhealth data via named data networking," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 142–147.

[81] G. Grassi, D. Pesavento, G. Pau, R. Vuyyuru, R. Wakikawa, and L. Zhang, "Vanet via named data networking," in *2014 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 410–415.

[82] M. Gibbens, C. Gniady, L. Ye, and B. Zhang, "Hadoop on named data networking: Experience and results," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, pp. 2:1–2:21, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3084439

[83] S. Chen, J. Cao, and L. Zhu, "Ndss: A named data storage system," in *2015 International Conference on Cloud and Autonomic Computing*, Sept 2015, pp. 196–199.

[84] W. Shang, Z. Wen, Q. Ding, A. Afanasyev, and L. Zhang, "Ndnfs: An ndn-friendly file system," *NDN Technical Report NDN-0027, Revision 1*, 2014.

[85] A. Afanasyev, I. Moiseenko, L. Zhang *et al.*, "ndnsim: Ndn simulator for ns-3," *University of California, Los Angeles, Tech. Rep*, vol. 4, 2012.

[86] M. Lacage and T. R. Henderson, "Yet another network simulator," in *Proceeding from the 2006 workshop on ns-2: the IP network simulator*. ACM, 2006, p. 12.

[87] B. Etefia, M. Gerla, and L. Zhang, "Supporting military communications with named data networking: An emulation analysis," in *MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*. IEEE, 2012, pp. 1–6.

[88] O. P. Team, "Opnet network simulator," http://opnetprojects.com/opnet-network-simulator/.

[89] H. Taleb, S. Hamrioui, P. Lorenz, and A. Bilami, "Integration of energy aware wsns in cloud computing using ndn approach," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, 2017, pp. 188–192.

[90] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8. Boston, MA, USA, 2010, p. 9.

[91] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "Nfd developer's guide," *Technical report, NDN-0021, NDN*, 2014.

[92] A. V. Aho and J. D. Ullman, "The theory of languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 97–125, 1968.

[93] M. A. Harrison, *Introduction to formal language theory.* Addison-Wesley Longman Publishing Co., Inc., 1978.

[94] G. Rozenberg and A. Salomaa, *Handbook of Formal Languages: Volume 3 Beyond Words.* Springer Science & Business Media, 2012.

[95] J. E. Hopcroft, J. D. Ullman, M. Rabin, and D. Scott, "Introduction to automata theory, languages, and computation," *IBM Journal of Research and Development*, vol. 3, pp. 114–125.

[96] INRIA, "The coq reference manual," https://coq.inria.fr/distrib/current/refman/, accessed: 2019-05-20.

[97] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, "Software foundations," *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[98] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[99] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2013.

[100] É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain, "A3pat, an approach for certified automated termination proofs," 2010.

[101] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, "A modular integration of sat/smt solvers to coq through proof witnesses," in *International Conference on Certified Programs and Proofs.* Springer, 2011, pp. 135–150.

[102] B. Chetali and Q.-H. Nguyen, "About the world-first smart card certificate with eal7 formal assurances," *Slides 9th ICCC, Jeju, Korea (September 2008), www. commoncriteriaportal. org/iccc/9iccc/pdf B*, vol. 2404, 2008.

[103] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 42–54.

[104] H. Geuvers, F. Wiedijk, and J. Zwanenburg, "A constructive proof of the fundamental theorem of algebra without using the rationals," in *International Workshop on Types for Proofs and Programs.* Springer, 2000, pp. 96–111.

[105] G. Gonthier, "Advances in the formalization of the odd order theorem," in *International Conference on Interactive Theorem Proving.* Springer, 2011, pp. 2–2.

[106] I. D. Team, "Isabelle," http://isabelle.in.tum.de/, accessed: 2019-05-20.

[107] H. P. project, "Hol interactive theorem prover," https://hol-theorem-prover.org/, accessed: 2019-05-20.

[108] P. D. Team, "Pvs specification and verification system," http://pvs.csl.sri.com/, accessed: 2019-05-20.

[109] Y. Marquer, L. Maignan, and J.-B. Yunès, "Proving formally a field-based fssp solution," 2018.

[110] F. Loulergue and J. Tesson, "Certified Parallel Program Calculation in Coq: A Tutorial," in *International Conference on High Performance Computing and Simulation (HPCS)*, ser. HPCS. Bologna, Italy: IEEE, 2014. [Online]. Available: https://hal.inria.fr/hal-00966632

[111] INRIA, "The gallina specification language coq 8.9.1 documentation," https://coq.inria.fr/refman/language/gallina-specification-language.html, accessed: 2019-05-20.

[112] A. V. Aho and J. D. Ullman, *The theory of parsing, translation, and compiling.* Prentice-Hall Englewood Cliffs, NJ, 1972, vol. 1.

[113] T. Parr and K. Fisher, "Ll (*): the foundation of the antlr parser generator," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 425–436.

[114] T. Anderson, J. Eve, and J. J. Horning, "Efficientlr (1) parsers," *Acta Informatica*, vol. 2, no. 1, pp. 12–39, 1973.

[115] Y. R.-G. François Pottier, "Menhir reference manual (version 20181113)," http://gallium.inria.fr/~fpottier/menhir/manual.html, accessed: 2019-05-20.

[116] J.-H. Jourdan, F. Pottier, and X. Leroy, "Validating lr (1) parsers," in *European Symposium on Programming.* Springer, 2012, pp. 397–416.

[117] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert-a formally verified optimizing compiler," 2016.

[118] INRIA, "Psatz," https://coq.inria.fr/refman/addendum/micromega.html, accessed: 2019-05-20.

[119] C. Baier and J.-P. Katoen, *Principles of model checking.* MIT press, 2008.

[120] M. Magnin, "Réseaux de petri à chronomètres: temps dense et temps discret," Ph.D. dissertation, Nantes, 2007.

[121] M. BENDIAF, "Spécification et vérification des systèmes embarqués temps réel en utilisant la logique de réécriture," Ph.D. dissertation, UNIVERSITE MOHAMED KHIDER BISKRA, 2018.

[122] R. Alur and D. Dill, "Automata for modeling real-time systems," in *International Colloquium on Automata, Languages, and Programming.* Springer, 1990, pp. 322–335.

[123] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[124] A. N. Prior, *Past, present and future.* Clarendon Press Oxford, 1967, vol. 154.

[125] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[126] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[127] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and computation*, vol. 111, no. 2, pp. 193–244, 1994.

[128] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[129] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller *et al.*, "Uppaal-now, next, and future," in *Summer School on Modeling and Verification of Parallel Processes*. Springer, 2000, pp. 99–124.

[130] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal implementation secrets," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2002, pp. 3–22.

[131] A. David, G. Behrmann, K. G. Larsen, and W. Yi, "New uppaal architecture," in *Workshop on Real-Time Tools, Uppsala University Technical Report Series*, 2002.

[132] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal smc tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[133] D. Basile, M. H. ter Beek, and V. Ciancia, "Statistical model checking of a moving block railway signalling scenario with uppaal smc," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 372–391.

[134] C. Nigro, L. Nigro, and P. F. Sciammarella, "Modelling and analysis of multi-agent systems using uppaal smc," *International Journal of Simulation and Process Modelling*, vol. 13, no. 1, pp. 73–87, 2018.

[135] J. Arias, M. Desainte-Catherine, and C. Rueda, "Exploiting parallelism in fpgas for the real-time interpretation of interactive multimedia scores," 2015.

[136] K. G. Larsen, B. Steffen, and C. Weise, "Continuous modeling of real-time and hybrid systems: from concepts to tools," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 64–85, 1997.

[137] J. Byg, K. Y. Jørgensen, and J. Srba, "Tapaal: Editor, simulator and verifier of timed-arc petri nets," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2009, pp. 84–89.

[138] S. Yovine, "Kronos: A verification tool for real-time systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 123–133, 1997.

[139] T. A. Henzinger and P.-H. Ho, "Hytech: The cornell hybrid technology tool," in *International Hybrid Systems Workshop*. Springer, 1994, pp. 265–293.

[140] C. Mahmoudi, "Orchestration d'agents mobiles en communauté," Ph.D. dissertation, 2014.

[141] C. Dumont, "Système d'agents mobiles pour les architectures de calculs auto-adaptatifs," Ph.D. dissertation, 2014.

[142] G. L. Djiken, "La mobilité du code dans les systèmes embarqués," Ph.D. dissertation, Paris Est, 2018.

[143] N. Team, "ndn-cxx: Ndn c++ library with experimental extensions," https://github.com/named-data/ndn-cxx, accessed: 2019-11-2.

[144] D. J. Mala, *Object Oriented Analysis and Design Using UML.* Tata McGraw-Hill Education, 2013.

[145] S. Sultana and F. Arif, "From verification to implementation: Uppaal to c++," *American Journal of Engineering Research (AJER), e-ISSN*, pp. 2320–0847, 2016.

[146] arieleiz, "Uppaal2c," https://github.com/arieleiz/UPPAAL2C.

[147] E. Gamma, *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[148] J. Dongo, "Prototype version ndfs and nmapreduce," https://github.com/mistersound/simulation/tree/master/ndnSIM.

[149] N. Heckert and J. J. Filliben, "Nist handbook 148: Dataplot reference manual, volume i: Commands," *National Institute of Standards and Technology Handbook Series*, 2003.

[150] K. Mills, J. Filliben, and C. Dabrowski, *Assessing Effects of Asymmetries, Dynamics, and Failures on a Cloud Simulator*, 2015.

[151] D. S. Summit, "Real world archirecture and deployment best practices," https://www.slideshare.net/HadoopSummit/real-world-archirecture-and-deployment-best-practices, accessed: 2019-10-15.

[152] N. Team, "Named data networking forwarding daemon," https://github.com/named-data/NFD, accessed: 2019-11-2.

[153] ——, "Ndn essential tools," https://github.com/named-data/ndn-tools, accessed: 2019-11-2.

[154] ——, "Nlsr - named data link state routing protocol," https://github.com/named-data/NLSR, accessed: 2019-11-2.

[155] ——, "sync library for multiuser realtime applications for ndn," https://github.com/named-data/ChronoSync, accessed: 2019-11-2.

[156] ——, "Psync," https://github.com/named-data/PSync, accessed: 2019-11-2.

[157] O. Foundation, "Getting started guide," https://nodejs.org/en/docs/guides/getting-started-guide/, accessed: 2019-10-15.

[158] A. S. Foundation, "Hadoop apache hadoop 2.9.2," https://hadoop.apache.org/docs/r2.9.2/, accessed: 2019-09-12.

[159] O. Technology, "Java se 7 archive downloads," https://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html, accessed: 2019-09-12.

[160] J. Dongo, C. Mahmoudi, and F. Mourlin, "Ndn log analysis using big data techniques: Nfd performance assessment," in *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2018, pp. 169–175.

[161] K. Hwang and M. Chen, *Big-data analytics for cloud, IoT and cognitive computing*. John Wiley & Sons, 2017.

[162] N. Team, *ndndump*, https://github.com/named-data/ndn-tools/tree/master/tools/dump.

[163] J. Twidell and T. Weir, *Renewable energy resources*. Routledge, 2015.

[164] Y. Yan, Y. Qian, H. Sharif, and D. Tipper, "A survey on smart grid communication infrastructures: Motivations, requirements and challenges," *IEEE communications surveys & tutorials*, vol. 15, no. 1, pp. 5–20, 2013.

[165] J. Sanz, G. Matute, H. Bludszuweit, and E. Laporta, "Microgrids, a new business model for the energy market," in *International Conference on Renewable Energies and Power Quality*, 2014.

[166] M. E. El-Hawary, "The smart grid—state-of-the-art and future trends," *Electric Power Components and Systems*, vol. 42, no. 3-4, pp. 239–250, 2014.

[167] A. G. Phadke and J. S. Thorp, "Phasor measurement units and phasor data concentrators," in *Synchronized Phasor Measurements and Their Applications*. Springer, 2017, pp. 83–109.

[168] X. Liu, L. Golab, W. Golab, I. F. Ilyas, and S. Jin, "Smart meter data analytics: systems, algorithms, and benchmarking," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 1, p. 2, 2017.

[169] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke, "Smart grid technologies: Communication technologies and standards," *IEEE transactions on Industrial informatics*, vol. 7, no. 4, pp. 529–539, 2011.

[170] S. Borlase, *Smart grids: infrastructure, technology, and solutions*. CRC press, 2017.

[171] H. Bilil, C. Mahmoudi, and M. Maaroufi, "Named Data Networking for Smart Grid Information Sharing," in *International Renewable and Sustainable Energy Conference*, 2017.

[172] Cisco, "Bandwidth, packets per second, and other network performance metrics," https://tools.cisco.com/security/center/resources/network_performance_metrics.

[173] I. E. T. F. (IETF), "Internet protocols for the smart grid," https://tools.ietf.org/html/rfc6272, accessed: 2018-05-5.

[174] A. McGibney, S. Rea, and J. Ploennigs, "Open bms-iot driven architecture for the internet of buildings," in *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2016, pp. 7071–7076.

[175] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[176] D. G. Holmberg and D. Evans, *BACnet wide area network security threat assessment.* US Department of Commerce, National Institute of Standards and Technology, 2003.

[177] W. S. Lee and S. H. Hong, "Implementation of a knx-zigbee gateway for home automation," in *2009 IEEE 13th International Symposium on Consumer Electronics.* IEEE, 2009, pp. 545–549.