Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Practical Framework for Byzantine Fault-tolerant Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Adrian Djokic
Matrikelnummer 0928197

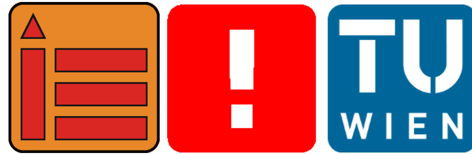
an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Nitin Vaidya (University of Illinois)

Wien, 17. März 2014

(Unterschrift Verfasser/In) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at



Practical Framework for Byzantine Fault-tolerant Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Adrian Djokic
Registration Number 0928197

elaborate at the
Institut of Computer Aided Automation
Reseach Group for Industrial Software
to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig
Assistance: Nitin Vaidya (University of Illinois)

Vienna, March 17, 2014

Statement by Author

Adrian Djokic
Alser Straße 43/19, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

____________________________ ____________________________

(Place, Date) (Signature of Author)

Acknowledgements

Thanks for all the love and support from my family and friends: my dear sister, parents, grandparents, Elif, Clara, my Vienna & U-C Crews, my office mates, Figen & Ghazale, thanks for staying & working late into the evenings, my library crew, Mr. Ogner, Sami, Ștefan, Lazar, and Dušan.
For all of my academic support, I would like to thank Brigitte, Prof. Thomas Grechenig, and Prof. Nitin Vaidya.
This would not have been possible without you.

Kurzfassung

Ein verteiltes System führt ein Programm oder einen Algorithmus auf mehreren Knoten aus, die über ein Netzwerk miteinander verbunden sind. Mit den heutzutage immer schneller wachsenden Computernetzwerk-Systemen wird es zunehmend schwieriger, deren Robustheit zu gewährleisten. Einige Probleme, die in einem Computernetzwerk auftreten können, sind: übernommene Systeme nach einer Hacker-Attacke, unzuverlässige Daten-Übertragungsmedien sowie Hardware-Ausfälle.

Das byzantinische Fehlermodell umfasst alle möglichen Fehler eines Systems. Darunter gehören Abstürze sowie Timing-Fehler und verlorengegangene Übertragungsdaten. Die stetige Entwicklung neuer Technologien macht es zwingend notwendig, dass Fehlertoleranz in unternehmens- und überlebenswichtigen Systemen richtig erfasst wird. Wir präsentieren ein System, das erleichtertes Testen und Benchmarking eines verteilten Systems ermöglicht. Dieses Framework bietet eine Basis an, die einen aussagekräftigen Leistungsvergleich von verteilten fehlertoleranten Protokollen erlaubt.

Die Praktikabilität des Frameworks ist von vorrangiger Bedeutung: Es soll einfach sein, das Framework einzusetzen, auszuführen und zu warten. Ein modulares Design ermöglicht eine einfache Erweiterung des Quellcodes und der Konfiguration. Darüber hinaus ist die Dokumentation des Frameworks sehr umfangreich. Sie beinhaltet eine umfassende Code-Dokumentation und visuelle Darstellungen wie Klassendiagramme, die das Verständnis des Frameworks erleichtern. Die Umsetzung des Frameworks in Java ermöglicht die Bereitstellung und Ausführung des Programms auf meheren Plattformen sowie ein objektorientiertes Design, was einen großen Vorzug für Modularität bietet.

Das modulare Framework ermöglicht es, verteilte Algorithmen zu testen. Zusätzlich sind wir in der Lage, neue Ideen für Storage-Netzwerke, wie z.B. Netzwerkkodierung, einzusetzen. Netzwerkkodierung ist eine Möglichkeit, um die Effizienz eines Netzwerkes und die Widerstandsfähigkeit gegen Angriffe zu steigern. Das Kodierungsverfahren stellt auch eine alternative Methode dar, um byzantinische Fehlertoleranz zu gewährleisten, während andere Methoden nur auf Kryptographie basieren. Netzwerkkodierung ist im wesentlichen ein Verfahren, das ein größeres Paket in kleinere Päckchen zerstückelt, oder diese wieder in die ursprüngliche Form zurückversetzt.

Obwohl der aktuelle Stand des Frameworks keinen Kodierungssalgorithmus anbietet, integriert die Standardimplementierung des fehlertoleranten-Protokolls die Methodenaufrufe zu einer abstrakten Java-Klasse, die zum Codieren und Decodieren der Daten dient. Die Erweiterung des Frameworks mit einem Codierungsschema, wie z.B. Reed-Solomon, würde somit die volle Funktionalität anbieten.

Schlüsselwörter

Byzantinische Fehlertoleranz, Byzantinisches fehlertolerantes System, BFT, Netzwerk-Coding, verteilte Algorithmen, verteilte Systeme

Abstract

A distributed system executes one program or algorithm on multiple networked nodes. As computer network systems continue growing, it becomes increasingly complex to maintain their robustness. Some of the issues that have to be dealt with in a network are: subverted systems following an adversarial attack, unreliable transmission of data, as well as hardware failures.

The Byzantine fault model encompasses all possible faults in a system, which, among others, include crash, timing, and omission failures. Continuous discoveries of new technologies make it imperative that fault tolerance is addressed properly in mission-critical systems. We present a system that provides functionality for testing and benchmarking on a distributed system. This framework offers a common ground to be able to make a fair performance comparison of distributed fault-tolerant protocols.

The framework's practicality is of primary concern. This means that it should be easily deployed, run, and maintained. Its modular design allows the code baseline to be extended and configured with ease. Moreover, the documentation of the framework, which includes comprehensive code documentation, as well as visual representations such as class diagrams, facilitate an easier understanding at both a lower and higher level. Implementing the framework in Java allows for cross-platform deployment and execution, as well as an object-oriented design, which eases modularity.

Having a modular framework to test distributed algorithms enables the use of novel ideas for storage networks, such as network coding, for Byzantine fault-tolerant algorithms. Network coding is a way to improve a network's efficiency and resilience to attacks, and poses an alternative methodology to provide Byzantine fault tolerance compared to those solely relying on cryptography. Coding essentially breaks up data into smaller packets or combines data packets into the original form.

Although the current state of the framework does not provide a coding algorithm, the default fault-tolerant protocol implementation integrates the calls to an abstract coding scheme to encode and decode data. Extending the framework with a coding scheme, such as Reed-Solomon, would enable the full use of this functionality.

Keywords

Byzantine fault-tolerance, Byzantine fault-tolerant system, BFT, network coding, distributed algorithms, distributed systems

Contents

1 Introduction **1**
1.1 Problem Description 1
1.2 Motivation 2
1.3 Aim of Work 2
1.4 Structure of Work 3

2 Fundamentals **4**
2.1 Byzantine Fault-Tolerance 4
2.2 Coding 5
2.3 State-of-the-Art 6

3 Methodological Approach **7**
3.1 Design 7
3.2 Models 13
3.3 Implementation 16
3.4 Testing 17
3.4.1 Environment 17
3.4.2 Plan 17
3.4.3 Expected Results 18
3.4.4 Execution and Analysis 18

4 Algorithms **33**
4.1 DefaultFTProtocol 33
4.1.1 Pseudocode 33
4.1.2 Graphical Example 33

5 Implementation Results **46**

6 Discussion **47**

7 Conclusion **49**

Bibliography **50**
References 50
Online References 53

A Appendix **54**
A.1 Source Code 54

List of Figures

2.1 Cluster with three nodes and one faulty primary node. 4
2.2 Cluster with three nodes and one faulty secondary node. 5
2.3 Cluster with six nodes and one faulty primary and secondary node. 5

3.1 Framework class diagram — AbstractPackageHandler 8
3.2 Framework class diagram — AbstractStorageEntity 9
3.3 Framework class diagram — StoragePackage 10
3.4 Framework sequence diagram — Client-side 14
3.5 Framework sequence diagram — Server-side 15
3.6 Write time for data packages without hashing, grouped by cluster. 21
3.7 Write time for data packages without hashing using corrected data, grouped by cluster. 21
3.8 Write time for 1 kB-100 kB data packages without hashing, grouped by cluster. . . . 21
3.9 Write time for 1 kB-100 kB data packages without hashing using corrected data, grouped by cluster. 22
3.10 Write time for 100 kB-10000 kB data packages without hashing, grouped by cluster. 22
3.11 Write time for 100 kB-10000 kB data packages without hashing using corrected data, grouped by cluster. 23
3.12 Write time for data packages with hashing, grouped by cluster. 23
3.13 Write time for data packages with hashing using corrected data, grouped by cluster. . 24
3.14 Write time for 1 kB-100 kB data packages with hashing, grouped by cluster. 24
3.15 Write time for 1 kB-100 kB data packages with hashing using corrected data, grouped by cluster. 25
3.16 Write time for 100 kB-10000 kB data packages with hashing, grouped by cluster. . . 25
3.17 Write time for 100 kB-10000 kB data packages with hashing using corrected data, grouped by cluster. 26
3.18 Write time for data packages without hashing, grouped by cluster. 26
3.19 Write time for data packages without hashing using corrected data, grouped by cluster. 27
3.20 Write time for data packages with hashing, grouped by cluster. 27
3.21 Write time for data packages with hashing using corrected data, grouped by cluster. . 28
3.22 Write time versus the number of tolerated failures for data packages without hashing, grouped by data package size. 29
3.23 Write time versus the number of tolerated failures for data packages without hashing using corrected data, grouped by data package size. 29
3.24 Write time versus the number of tolerated failures for data packages with hashing, grouped by data package size. 30
3.25 Write time versus the number of tolerated failures for data packages with hashing using corrected data, grouped by data package size. 30
3.26 Write time versus the number of tolerated failures without hashing, grouped by data package size. 31
3.27 Write time versus the number of tolerated failures without hashing using corrected data, grouped by data package size. 31
3.28 Write time versus the number of tolerated failures with hashing, grouped by data package size. 32

3.29 Write time versus the number of tolerated failures with hashing using corrected data, grouped by data package size. 32

4.1 Client writing package p to faulty primary server, which then distributes it to the secondary servers. 38
4.2 Servers exchanging data received from faulty primary. 39
4.3 Servers responding to client acknowledging receipt of p from faulty primary. 39
4.4 Client telling servers to commit. 40
4.5 Client writing package p to the primary server, which then distributes it to the secondary servers, of which one is faulty. 41
4.6 Servers exchanging data received from a non-faulty primary. 41
4.7 Servers responding to client acknowledging receipt of p from faulty primary. 42
4.8 Client telling all servers, except for the faulty secondary server, to commit. 42
4.9 Client writing to a cluster with a faulty primary and one faulty secondary server. . . . 43
4.10 Client reading from a cluster with a faulty primary. 44
4.11 Servers responding to client's read request or requesting package from other servers. 44

List of Tables

3.1 Package Write time per storage entity in seconds without hashing. 19
3.2 Package Write time per storage entity in seconds with hashing. 20

List of Listings

3.1 AbstractPackageHandler interface . . . 7
3.2 AbstractFTProtocolHandler constructors . . . 11
3.3 AbstractCodingScheme abstract methods . . . 11
3.4 CodingSchemeFactory get() method . . . 11

A.1 client.StorageClient . . . 54
A.2 client.StorageClientPackageHandler . . . 61
A.3 server.StorageServer . . . 62
A.4 server.StorageServerConfigurationParser . . . 70
A.5 server.StorageServerFactory . . . 74
A.6 server.StorageServerPackageHandler . . . 76
A.7 shared.AbstractPackageHandler . . . 80
A.8 shared.AbstractStorageEntity . . . 91
A.9 shared.codingScheme.AbstractCodingScheme . . . 98
A.10 shared.codingScheme.AbstractCodingSchemeMetadata . . . 99
A.11 shared.codingScheme.CodingSchemeFactory . . . 99
A.12 shared.codingScheme.CodingSchemeType . . . 100
A.13 shared.codingScheme.DefaultCodingScheme . . . 100
A.14 shared.codingScheme.ReedSolomon . . . 101
A.15 shared.exceptions.ClientConfigurationValidationException . . . 101
A.16 shared.exceptions.ConfigurationException . . . 102
A.17 shared.exceptions.FTProtocolException . . . 102
A.18 shared.exceptions.ProtocolException . . . 103
A.19 shared.exceptions.QuitException . . . 103
A.20 shared.exceptions.StorageClusterConfigurationException . . . 104
A.21 shared.exceptions.StorageNodeConfigurationException . . . 104
A.22 shared.exceptions.StorageServerConfigurationException . . . 105
A.23 shared.ftProtocol.AbstractFTProtocolHandler . . . 105
A.24 shared.ftProtocol.DefaultFTProtocol . . . 109
A.25 shared.ftProtocol.DefaultFTProtocolMetadata . . . 128
A.26 shared.ftProtocol.FTProtocolHandlerFactory . . . 128
A.27 shared.ftProtocol.FTProtocolType . . . 131
A.28 shared.storageCluster.StorageCluster . . . 131
A.29 shared.storageCluster.StorageClusterConfigurationParser . . . 137
A.30 shared.storageCluster.StorageClusterFactory . . . 142
A.31 shared.storageCluster.StorageClusterPackageHandler . . . 143
A.32 shared.storageNode.DefaultStorageNode . . . 148
A.33 shared.storageNode.LocalStorageNode . . . 148
A.34 shared.storageNode.StorageNode . . . 150
A.35 shared.storageNode.StorageNodeFactory . . . 152
A.36 shared.storageNode.StorageNodePackageHandler . . . 155
A.37 shared.storageNode.StorageNodeType . . . 156
A.38 shared.storagePackage.AbstractStoragePackageMetadata . . . 156
A.39 shared.storagePackage.HashingSchemeType . . . 156

A.40 shared.storagePackage.ICodable 156
A.41 shared.storagePackage.StoragePackage 158
A.42 shared.storagePackage.StoragePackageFactory 166
A.43 shared.storagePackage.StoragePackageType 166
A.44 shared.util.ConfigUtil 167
A.45 shared.util.CreateData 171
A.46 shared.util.FtStorageUtil 171
A.47 shared.util.HashingUtil 173
A.48 shared.util.LoremIpsum 175

List of Algorithms

4.1 DefaultFTProtocol client-side write 34
4.2 DefaultFTProtocol server-side write 35
4.3 DefaultFTProtocol client-side read 36
4.4 DefaultFTProtocol server-side read 37

1 Introduction

1.1 Problem Description

Consider a distributed system which is a labeled directed tree[1] consisting of one *client* at the root and *storage nodes*, which may be nested at different levels. For two given adjacent nodes, the outgoing label of one is not necessarily equal to the incoming label from the same adjacent node. Subsets of the children of any given node may be formed into *storage clusters*. Newly introduced (storage) nodes may be dynamically added to a storage cluster and, by definition, be connected to the same parent as the other nodes in the same group.

Any node in a given storage cluster may send messages to and receive messages from the nodes in the same storage cluster. Outgoing and incoming edge labels of a parent represent, respectively, the response time for a parent to receive acknowledgement from a child that a message has been received, and the response time for a parent to receive a message from a child after a parent has requested a specific message. Assume that the system is synchronous in which edge labels are finite positive numbers and are bounded by some positive constant. This means that is this constant is reached, a node will assume that a message has been lost or that a faulty node exists.

A parent may request a message it has sent in the past from a specific child. For any given storage cluster, a parent may code[13] a message into smaller ones and send these fragments to a subset of nodes in the storage cluster. In order for a message to be reconstructed under a given fault model and coding scheme, the parent must know and decide how many children in the storage cluster receive which fragment. This means that under a given fault model and configuration, it is possible to reconstruct the original message from any given storage cluster, that is, by reading from a subset of non-faulty storage nodes in a given cluster.

Faults can be categorized into stopping and Byzantine failure models. Stopping systems can refuse to pass on information, but cannot relay false information[31]. That is, only valid data is output by a failed stopping system, or none at all. On the other hand, systems that exhibit Byzantine faults may deliver arbitrary data as output, which could be the result of a software or hardware bug, or an adversary trying to sabotage data or the calculations of these. Although it is impossible to attain a completely fault-free computing system, it is often sufficient to tolerate a predefined number of failures within a given time interval or provide that certain types of failure do not occur[34].

In order for a distributed system to reach consensus, that is, agree upon and eventually output a value, it is imperative that the following conditions[2] hold:

- **Agreement** No non-faulty processes[3] decide on values different from $v \in V$, where V is the value set from which all non-faulty processes are required to produce outputs.
- **Validity** If all non-faulty processes start with some initial value $v \in V$, then v is the only possible decision value for a non-faulty process.

[1] not an arborescence, i.e. for any vertex u connected to another vertex v, there also exists a directed path from v to u
[2] we slightly modify the original definition from [30] to fit our model
[3] each node runs an instance of the distributed algorithm in a process

- **Termination** All non-faulty processes eventually decide.
- **Integrity** All non-faulty processes decide on at most one value $v \in V$. If a non-faulty process has decided on some value v, then v must have been proposed by some process.

Note that an algorithm providing these conditions guarantees correctness for the agreement problem of a distributed system under a stopping, as well as under a Byzantine failure model.

Acknowledging the existence of many Byzantine fault-tolerant protocols and consensus algorithms [9][24][22][27][32] and data coding schemes[11][12][16][18][33], we propose a system that allows a user to evaluate the performance of various combinations of these. More specifically, we want to determine and optimize the response time of reading and writing messages to storage clusters under a given fault model, and evaluate and elaborate on the benefits and drawbacks under certain conditions, i.e. under different fault-tolerant protocols.

We limit ourselves to evaluating the described model above with one client and a 1-level tree. Different storage clusters represent combinations of storage schemes and consensus algorithms, or the lack thereof, which serve as reference points in terms of performance evaluation.

1.2 Motivation

An increasing reliance on digital data is unavoidable as more information is stored in this form and is growing exponentially. Replication, which is easily realized with digital data, is crucial in order to prevent data loss and also increases availability. Entities, such as consumers or businesses, must make data globally available, e.g. over the Internet, in order to be able to share them. At the same time, this may pose risks for sensitive data and incentivizes the use of security mechanisms. *Cloud* services are becoming increasingly popular as more entities connect to the Internet, share data, and want to abstract from the issues of dealing with availability and security. While many consumers and businesses assume a fault-free service, i.e. constant availability and security preventing unauthorized access or tampering, Cloud services have been known to fail under these circumstances[29][36][35][37].

As the sizes of computer network systems surges in order to keep up with more data and consumer demands, it becomes increasingly difficult to maintain the robustness of these networked systems. Future expectations are that arbitrary faults will become more common as the physical limits of circuits are pushed to the extremes and as the integrity of trending ubiquitous and complex distributed systems becomes more vulnerable to compromise[17]. Continuous discoveries of new technologies and increasing reliance on these make it imperative that fault tolerance is addressed properly in mission-critical systems. Handling faults is becoming increasingly challenging in complex systems and as such should be dealt with properly. Our shift to a digital reality poses a problem with the increasing reliance on security, integrity and availability of data.

1.3 Aim of Work

In order to evaluate the performance of different Byzantine fault-tolerant protocols, a system has to be designed and built with an architecture general enough to support these different protocols. This system has to be easily extensible to support new protocols, flexible enough to test under different hardware systems and topologies, and provide simple mechanisms for configuring a deployed system. The system will be designed as a modular framework to maximize its flexibility in terms of development, and must be easy to use in a deployed environment. That is to say, the system should be a *practical* framework for testing various Byzantine fault-tolerant protocols.

Since the extensibility of the framework is one of its key components, an ample amount of information has to be gathered about different types of existing Byzantine fault tolerant protocols in order to determine the interfaces needed to integrate these into the system. Furthermore, it is utterly important to show that a fault-tolerant system is indeed fault tolerant. For this purpose, test cases have to show that using the framework under a default configuration does indeed provide the expected fault tolerance. This *default configuration* for testing purposes will include an implementation of one Byzantine fault-tolerant protocol which allows the use of coding.

1.4 Structure of Work

This document is structured into seven chapters. The first and current chapter introduces some concepts required to understand the underlying problem of fault-tolerance and performance, as well as the motivation and goal of this work. Chapter two describes Byzantine fault-tolerance and a potential improvement to support fault-tolerance in more detail. Current state-of-the-art solutions are also brought forth, along with suggestions to improve on these.

The subsequent chapters, three and four, respectively describe the higher-level design of the framework and default fault-tolerant protocol implementation. The remaining chapters bring forth the results of having implemented the framework, including some advantages over existing solutions and drawbacks. Furthermore, open issues are also pointed out.

2 Fundamentals

2.1 Byzantine Fault-Tolerance

A reliable computer system must be able to cope with the failure of one or more of its components. However, the resilience of such a system often overlooks the possibility of failure by receiving conflicting or incorrect information. The problem of handling such a failure is expressed as the Byzantine Generals Problem[25].

A distributed system containing nodes which could handle arbitrary faults can be modeled by the Byzantine Generals Problem in which independent processes[1] reach agreement on a common value. Unreliable communication media or a faulty process may change the content of a proposed value arbitrarily and could thus prevent a subset of the processes from reaching consensus.

In order for a system to tolerate Byzantine faults, it must be able to handle arbitrary faults that failed components may exhibit. The processes in this system must agree upon a value, i.e. reach consensus on a proposed value, which is proposed by a known *proposer*, or primary node. It is imperative to know that in a system of n processes, no more than $\frac{n-1}{3}$ faulty processes can exist in the system. From this, we define the minimum number of processes in a distributed system that can handle up to f Byzantine faults to be $n \geq 3f + 1$.

We show a sketch proof in figures 2.1-2.3. Figures 2.1 and 2.2 assume that a configuration of $3f$ nodes can handle f faults, where $f = 1$. Node S_0 proposes values p and p' to nodes S_1 and S_2, respectively. It can be seen that in either case, it is impossible to determine which node is faulty, and that it is impossible to tell whether the proposing node or secondary node is faulty. We thus need one more node to be able to distinguish these scenarios.

We generalize this proof to $f > 1$. Assume in figure 2.3 that S_0 is the primary node and faulty, as is node S_1. S_0 proposes several values p, p', and p'', to the secondary nodes $S_1 - S_5$. Again, it is impossible to reach consensus on a value, because it is impossible to tell whether the primary

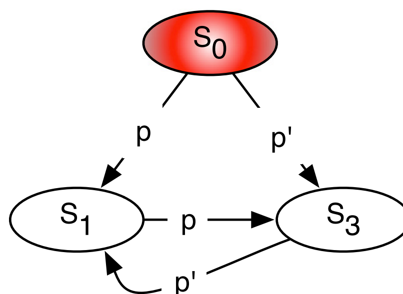


Figure 2.1: Cluster with three nodes and one faulty primary node.

[1] in a distributed system, processes run on individual nodes, and these terms will be used interchangeably in the following text

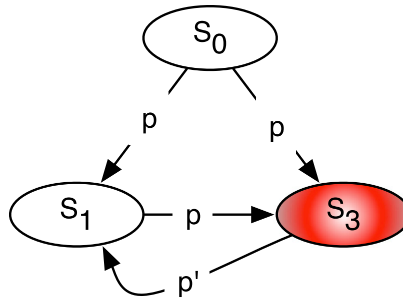


Figure 2.2: Cluster with three nodes and one faulty secondary node.

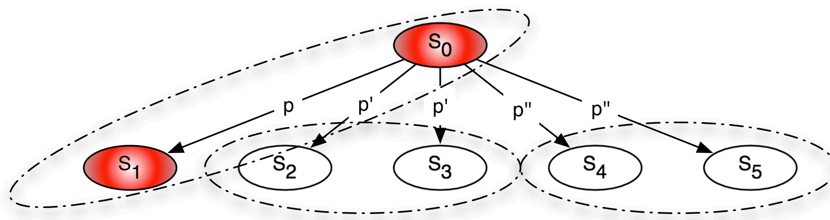


Figure 2.3: Cluster with six nodes and one faulty primary and secondary node.

node, along with $f-1$ other servers[2] are faulty, or whether one of the other $2f$ groups of servers are faulty, that is, S_2 and S_3, or S_4 and S_5. In this generalized case it can also be seen that one extra node to the existing $3f$ will guarantee that the non-faulty servers can distinguish whether or not the proposing server is faulty.

2.2 Coding

Coding is the discipline of transforming information into a sequence of symbols from a finite alphabet. Coding has been studied extensively to provide efficient algorithms for compression[23], security[3], and error-correction[42].

Lately, network coding has been explored for use in distributed storage systems[6], in order to improve the reliability, efficiency, and robustness of such systems[14]. Network coding allows a more efficient transmission of data in a network of systems and is a generalization of the conventional network routing method. In contrast to a *store and forward* procedure, intermediate nodes performing network coding generate output based on previous input. Network coding has been shown to minimize bandwidth between networked storage nodes through the use of regenerating codes[13].

[2] S_1 in this case

2.3 State-of-the-Art

Since even *practical* Byzantine fault-tolerance[7] is considered expensive[8], research has tried to optimize Byzantine fault-tolerance in various ways. At first, research has focused on increasing performance by reducing the number of nodes and communication between the nodes[22] [24] [15]. By decreasing the number of nodes in a system, other properties have to be relaxed. These may be, for example, properties about the assumptions of which nodes may *always* be trusted. Communication costs between nodes can be reduced by making *speculative* assumptions about whether or not data is valid. Communication cost can be further reduced by employing hashing functions in order to verify data that another node has sent. However, this technique is only as resistant as the safety and security of the hash function itself.

Other research shows that *increasing* the total number of nodes can drastically reduce the performance impact introduced by the increased number of nodes due to increased fault-tolerance[1][10]. That is, these protocols are aimed at reducing the performance impact by focusing on fault-scalability. However, these systems seem to be impractical from a commercial point-of-view.

The idea of using coding techniques in Byzantine fault-tolerant (BFT) systems, more specifically in conjunction with network coding techniques, has sprung up in research[28] more recently.

Network codes have been investigated for use in distributed storage systems [13][12], including *exact* regenerating codes for Byzantine fault-tolerant systems[18]. On top of reducing the amount of stored or transmitted data between nodes, network coding can be more secure than a hash function. The performance of a BFT algorithm which employs coding has been evaluated against a “classical” algorithm[27]. However, this has been done on a small scale with few nodes, and there is currently no known work of benchmarks with larger scale systems.

Current existing libraries that provide fault-tolerance for distributed systems[4][39] do not employ or enable the use of coding in their infrastructures, and are thus not suited for achieving our goal. Also, apart from the fact that these systems are not being actively maintained[5][40], changing a fundamental requirement in an existing complex system could introduce further bugs. Furthermore, introducing new functionality seems increasingly challenging in a system providing Byzantine fault-tolerance.

In order to gain impartial results from benchmarking various BFT algorithms, tests under different configurations need to be run on the same platform. Given the more recent developments of BFT systems and distributed storage systems, a truly practical and BFT system could come into fruition with the right combination of features, which may include coding, speculation, and modern data center technologies, including virtualization. Therefore, it is necessary to provide a framework that can provide or be easily implemented with such functionality.

3 Methodological Approach

3.1 Design

We first define our domain as a class diagram in figures 3.1-3.3.

The concept of an `AbstractStorageEntity` denotes any node[1] that can read and write data. Data is transmitted between nodes in a wrapper class `StoragePackage`, which are distinguished from one another by a universally unique identifier (UUID)[26].

An `AbstractPackageHandler` manages a `StoragePackage` for a given `AbstractStorageEntity` in a separate thread. The `AbstractPackageHandler` can, for example, forward a `StoragePackage` to another `AbstractStorageEntity` or decode a queue of encoded `StoragePackages`. The public constructors and methods of `AbstractPackageHandler` which implement functional requirements[2] are shown in listing 3.1.

```
1 //Constructor to prepare the AbstractPackageHandler for reading or writing.
2 public AbstractPackageHandler(AbstractStorageEntity requester,
  ↪ AbstractStorageEntity requestee, StoragePackage storagePackage);

3 public void addPendingPackage(StoragePackage storagePackage);

4 public void closeServerSocket();

5 public void run();
```

Listing 3.1: AbstractPackageHandler interface

The purpose of the constructor in listing 3.1 is to prepare an instance of the class for either reading or writing. The executed command is determined by the `StoragePackageType`. That is, if the passed `StoragePackage` is of type `StoragePackageType.READ`, then the package handler is to request a read from the given requestee. The `run()` method enables the instance to run in a separate thread, whereas the `addPendingPackage()` method adds a given `StoragePackage` to the package handler's queue of packages pending to be processed.

After an `AbstractPackageHandler` has been created by an `AbstractStorageEntity` for a given `StoragePackage`, corresponding `StoragePackages`, i.e. those with the same UUID as the original `StoragePackage` passed to the constructor, may be added to the package handler's processing queue. This processing queue could be used to reconstruct the original data of a given `StoragePackage`, in case the original data is coded, or to determine if a certain number of acknowledgments or corresponding packages have been received from other nodes in the same cluster, which is useful under a Byzantine fault model. The `closeServerSocket()` method is used to clean up a package handler, that is, close all open sockets, after a `StoragePackage` has been successfully written or read.

`AbstractFTProtocolHandler` is a subclass of `AbstractPackageHandler` which coheres to a stricter processing sequence for `StoragePackages`. That is, a `StoragePackage` is processed by

[1] as described in section 1.1

[2] Although the specification of this framework defines mostly non-functional requirements, any specific implementation or extension of it could employ functional requirements.

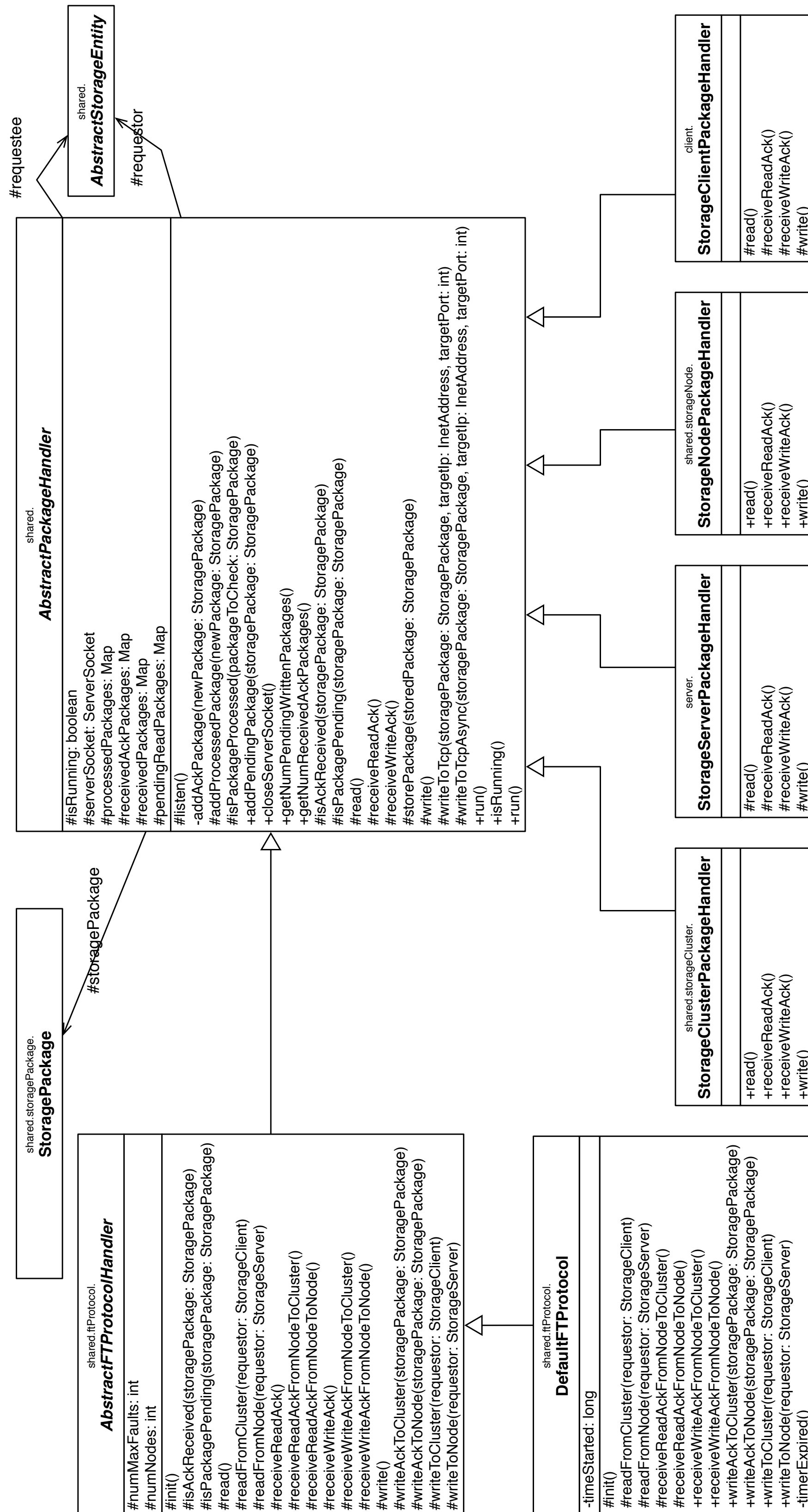


Figure 3.1: Framework class diagram — AbstractPackageHandler

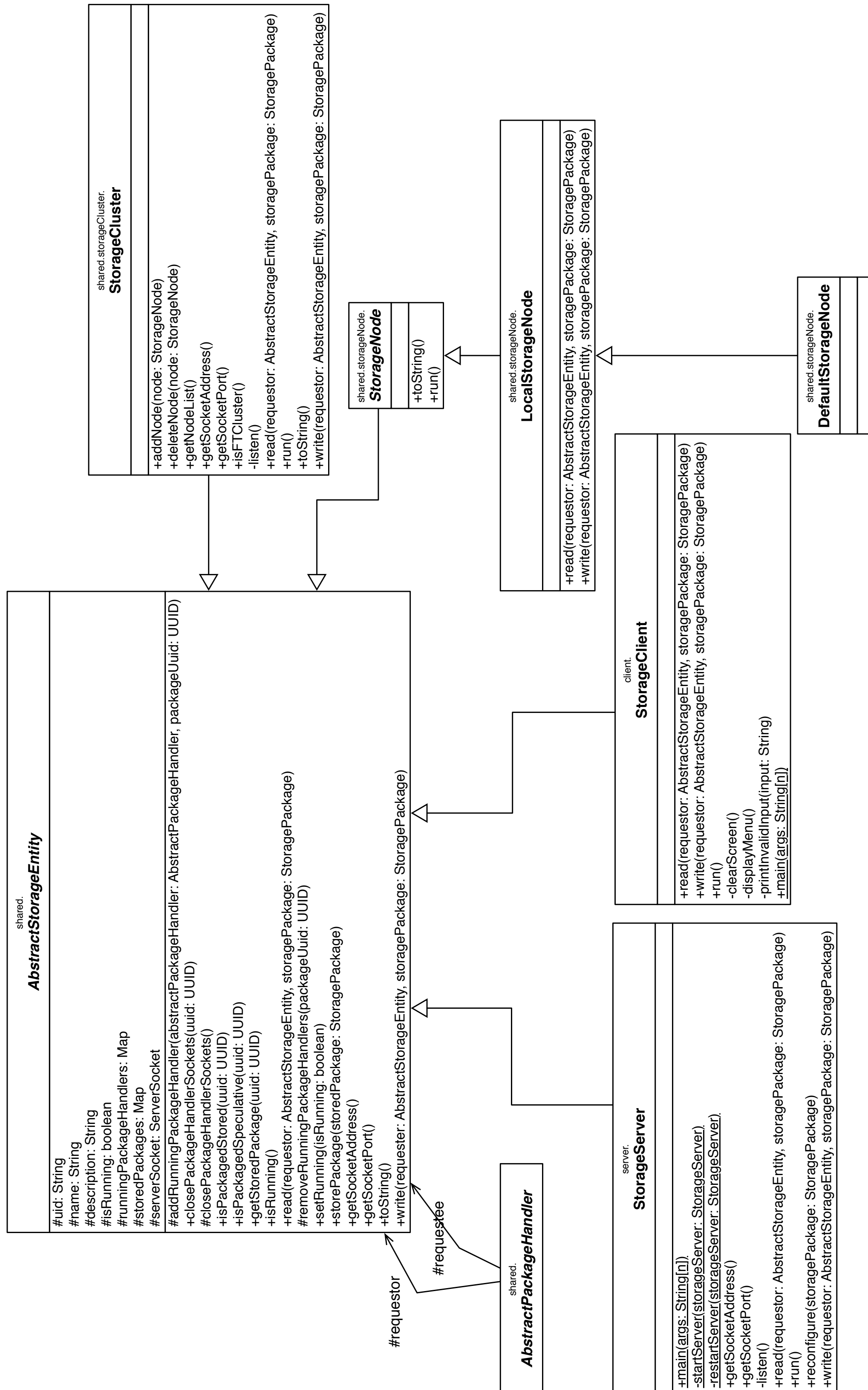


Figure 3.2: Framework class diagram — AbstractStorageEntity

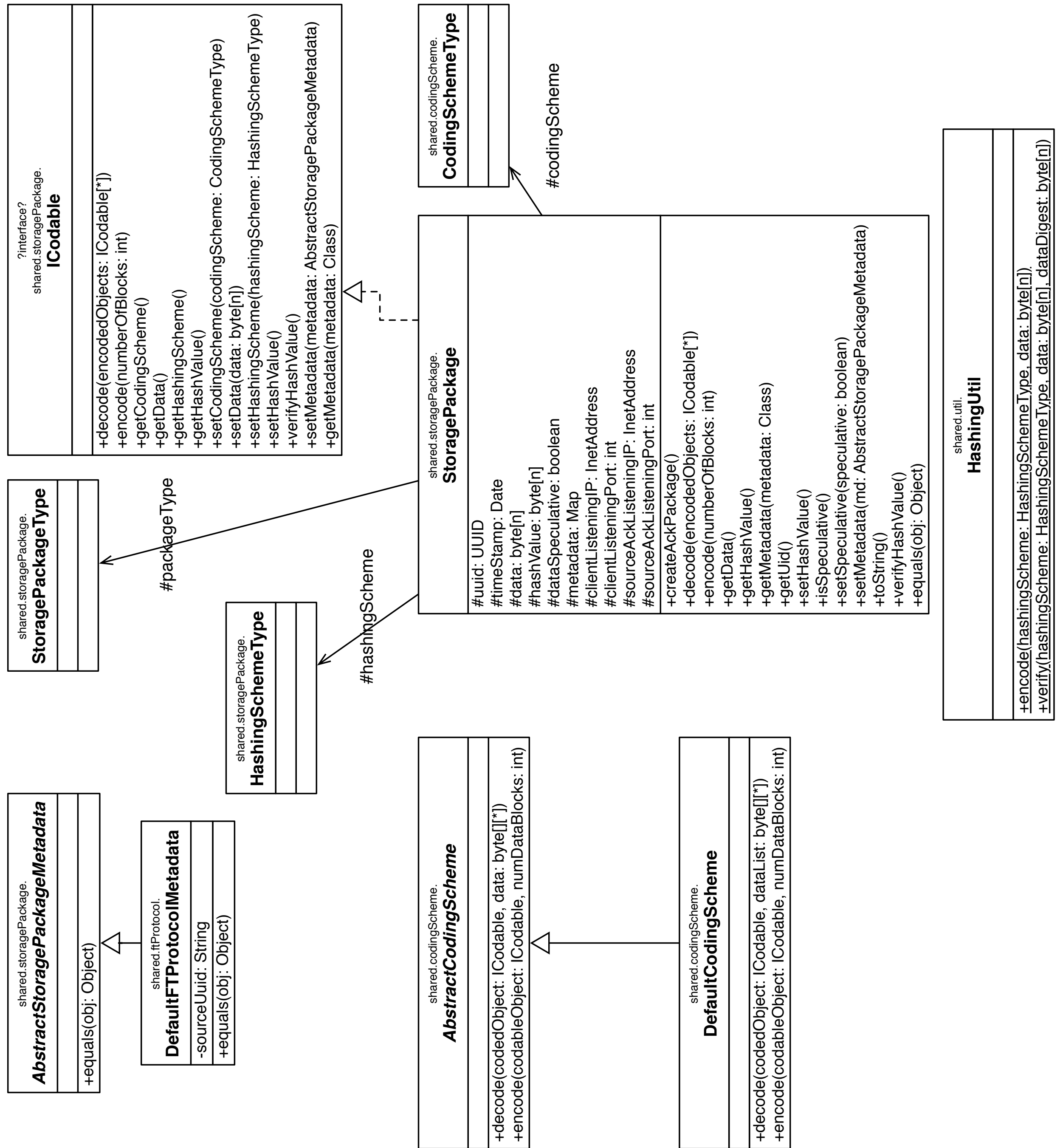


Figure 3.3: Framework class diagram — StoragePackage

instances of `AbstractStorageEntity` in a defined order, which is determined by the constructors of `AbstractFTProtocolHandler`. Also, the notion of a client- and server-side is further underscored in `AbstractFTProtocolHandler`. The constructors for `AbstractFTProtocolHandler` are shown in listing 3.2.

```
1 //Constructor called on the client side to prepare the
  ↪ AbstractFTProtocolHandler for reading or writing.
2 public AbstractFTProtocolHandler(StorageClient requestor, StorageCluster
  ↪ requestee, StoragePackage storagePackage);

3 //Constructor called on the server side to prepare the
  ↪ AbstractFTProtocolHandler for reading or writing.
4 public AbstractFTProtocolHandler(StorageServer requestor, StorageCluster
  ↪ requestee, StoragePackage storagePackage);
```

Listing 3.2: AbstractFTProtocolHandler constructors

The goal of implementing a subclass of `AbstractFTProtocolHandler` is to define a fault-tolerant protocol which writes to and reads from a `StorageCluster`. Since the subclass contains the logic for both the client- and server-side, the protocol is inherently encapsulated inside one entity. In addition to providing a certain level of fault-tolerance, a protocol might employ coding or use hashing as a form of data verification. Throughout this text, the use of *protocol* could denote one that is fault-tolerant, Byzantine fault-tolerant, or not fault tolerant. The fault-tolerance level should be clear from the context.

`AbstractCodingScheme` is an abstract class which requires its subclasses to implement the following two methods, which together define a coding scheme:

```
1 public byte[] decode(ICodable codedObject, List<byte[]> data);

2 public List<byte[]> encode(ICodable codableObject, int numDataBlocks);
```

Listing 3.3: AbstractCodingScheme abstract methods

As with the `AbstractFTProtocolHandler`, defining all processes in one class provides better encapsulation. The goal of the `encode()` and `decode()` methods is to provide functionality to break up the data of a codable object[3] into a list of bytes, as well as to reconstruct the original data from a list of `ICodable` objects. That is, the methods `encode()` and `decode()` are inverse functions of each other.

Even though `StoragePackage` is the only class implementing `ICodable`, the latter is used as a parameter type for the methods in listing 3.3 in order to abstract and emphasize the functionality of `AbstractCodingScheme`. Furthermore, `AbstractCodingScheme` is implemented as an abstract class, rather than an interface, because of the use of reflection in the class `CodingSchemeFactory`. Constructor arguments of a generic *class* are required in order to instantiate objects of a specific class, which is not possible with an interface. In this case, `AbstractCodingScheme` is used to retrieve a class's constructors and their respective arguments in order to instantiate subclasses. An example of reflection for this purpose can be found in listing 3.4.

```
1 //initialize codingSchemes with all values of CodingSchemeType and map to
  ↪ classes that implement AbstractCodingScheme and have the same name as
  ↪ the type value
2 private Map<CodingSchemeType, Class> codingSchemes;

3 public static AbstractCodingScheme get(CodingSchemeType codingScheme) {
4     Class[] constructorArgs = new Class[] {};
```

[3] an object whose class implements `ICodable`

```
5      Constructor<AbstractCodingScheme> ctor;
6      try {
7        ctor = getInstance().codingSchemes.get(codingScheme)
8            .getConstructor(constructorArgs);
9        return ctor.newInstance();
10     } catch (Exception e) {
11       // log error
12     }
13     return new DefaultCodingScheme();
14   }
```

Listing 3.4: CodingSchemeFactory get() method

While the implementation of the `get()` method shown in listing 3.4 is specifically for `AbstractCodingScheme`, the factory pattern is used for any class that is intended to be subclassed or created at runtime by means of a configuration entity, such as a configuration file. The following factories are to be implemented for their respective classes:

- `CodingSchemeFactory`
- `FTProtocolHandlerFactory`
- `StorageClusterFactory`
- `StorageNodeFactory`
- `StorageServerFactory`
- `StoragePackageFactory`

The factories for all subclasses of `AbstractStorageEntity`, as well as the one for `StoragePackage`, create instances of their respective classes by reading a configuration entity, while `CodingSchemeFactory` and `FTProtocolHandlerFactory` use reflection to allow easy subclassing of their respective classes. By employing the strategy of reflection, creating a new subclass of `AbstractCodingScheme` or `AbstractFTProtocolHandler` becomes trivial. The only adaptation required is to create a new class that extends the desired superclass, and to create a new entry inside of the respective `enum` type, i.e. add a new `enum` value with the class name inside of `CodingSchemeType` or `FTProtocolType`.

Given that the framework needs to be configurable, data, as well as configuration parameters, are to be sent in a `StoragePackage`. Communication between `AbstractStorageEntities` is performed solely via the `StoragePackage` wrapper class. For this reason, various flags need to be set in order to differentiate types of `StoragePackages`.

For example, `StoragePackageType` marks a `StoragePackage` as a *data* or *configuration* entity. Further configuration flags inherent to the framework are contained in `HashingSchemeType` and `CodingSchemeType`. Also, `AbstractStoragePackageMetaData` may be subclassed in order to tag a `StoragePackage` with arbitrary data, which could be used by concrete implementations of `AbstractPackageHandler`.

`StorageClient` and `StorageServer` denote two starting points of execution on the client- and server-side, respectively. A client may write a `StoragePackage`, which is ultimately stored on the server-side, and it may also read a given `StoragePackage`, based on its UUID, which is retrieved from one or more instances of `StorageServer`.

In order to benchmark the system described in section 1.1, we describe a sequence diagram in figures 3.4 and 3.5 for the client- and server-side, respectively. These diagrams show the flow of a `StoragePackage` that originates from a `StorageClient` and is distributed and stored among

the `StorageNodes`. An acknowledgment is ultimately sent back to the `StorageClient` from each `StorageNode` after a valid `StoragePackage` has been decided upon. Benchmarks are performed by evaluating the cost of the following metrics:

network bandwidth
: The total number of bytes transmitted over the network. In the case of *writing*, this is measured from the time at which a `StorageClient` first submits a `StoragePackage` for writing, until it is acknowledged that the `StoragePackage` has been stored. In the case of *reading*, this is measured from the time at which a `StorageClient` first requests a specific `StoragePackage`, until the original data of the `StoragePackage` has been received or reconstructed by the `StorageClient`.

storage
: The total number of bytes stored on each `StorageNode` after a `StorageClient` requests a specific `StoragePackage` to be written.

time
: This metric is defined by the time it takes for a `StoragePackage` to be written or read. The interval markers for these actions are the same as those for *network bandwidth*.

The storage logic is defined by the `StorageNode` entity. Although `StorageServer` is coupled with `StorageNode`, the latter may be used on the client side too. Depending on the implementation, a `StorageNode` may store data locally on a server or use a third-party API for remote data storage, e.g. Amazon S3 or Microsoft Azure. For scope of this work, `StorageNodes` will only be instantiated on the server side, and thus the terms *node* and *server* will be used interchangeably. Furthermore, only local storage space will be used for benchmarking purposes, since implementing the use of remote data storage is not in the scope of this work.

3.2 Models

We define a Byzantine fault to be one that renders a `StorageServer` unreliable. An unreliable `StorageServer` may withhold or tamper with the data of a `StoragePackage` as it propagates through the system, which could happen because a `StorageServer` goes offline or maliciously alters the data. As shown in figures 3.4 and 3.5, a `StoragePackage` is handled differently, depending on whether it is written to or read from a `StorageCluster` which is fault-tolerant or not. If it is, then the respective subclass of `AbstractFTProtocolHandler` is instantiated, otherwise the `StoragePackage` is processed by the default sequence of `AbstractProtocolHandlers`.

In a fault-tolerant cluster, the protocol, i.e. the implementing instance of `AbstractFTProtocolHandler`, must be able to recognize a faulty `StoragePackage`. A faulty `StoragePackage` might be one that is received out of sequence, e.g. if an acknowledgment for a package has been received before a corresponding data or configuration `StoragePackage` has been received, or if the data does not correspond with the majority of the received `StoragePackages` for a given UUID. Since there is a minimum threshold for the number of `StoragePackages` received, one that is withheld for a given amount of time is considered faulty. In general, anything that deviates from the intended flow of a `StoragePackage`'s lifecycle is considered a fault.

Taking into account the system architecture described in section 3.1, we present `DefaultFTProtocol`, an algorithm to handle fault-tolerant reads and writes. For this protocol we use a multi-reader multi-writer model, meaning that many clients may write to and read from the storage system.

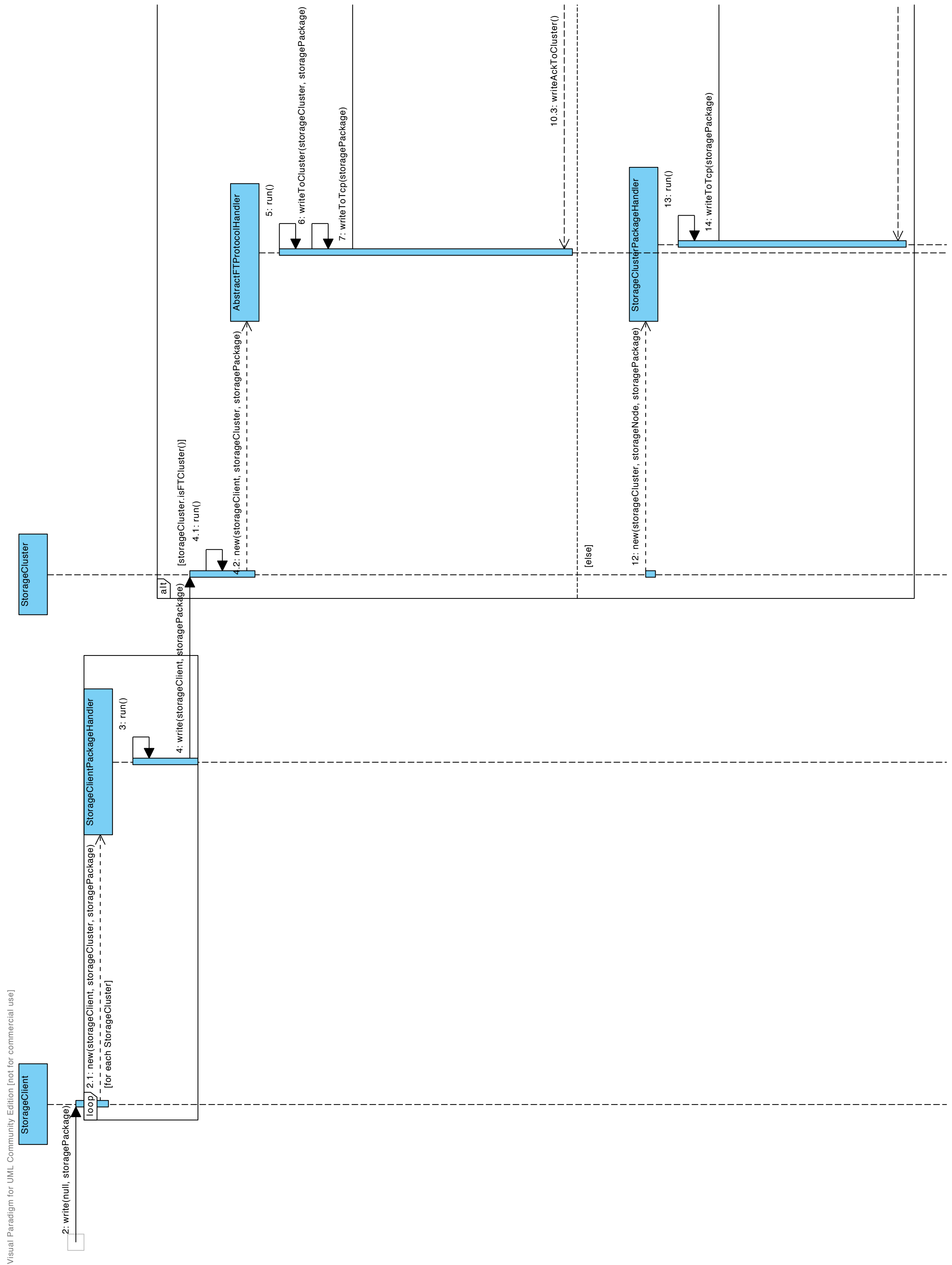


Figure 3.4: Framework sequence diagram — Client-side

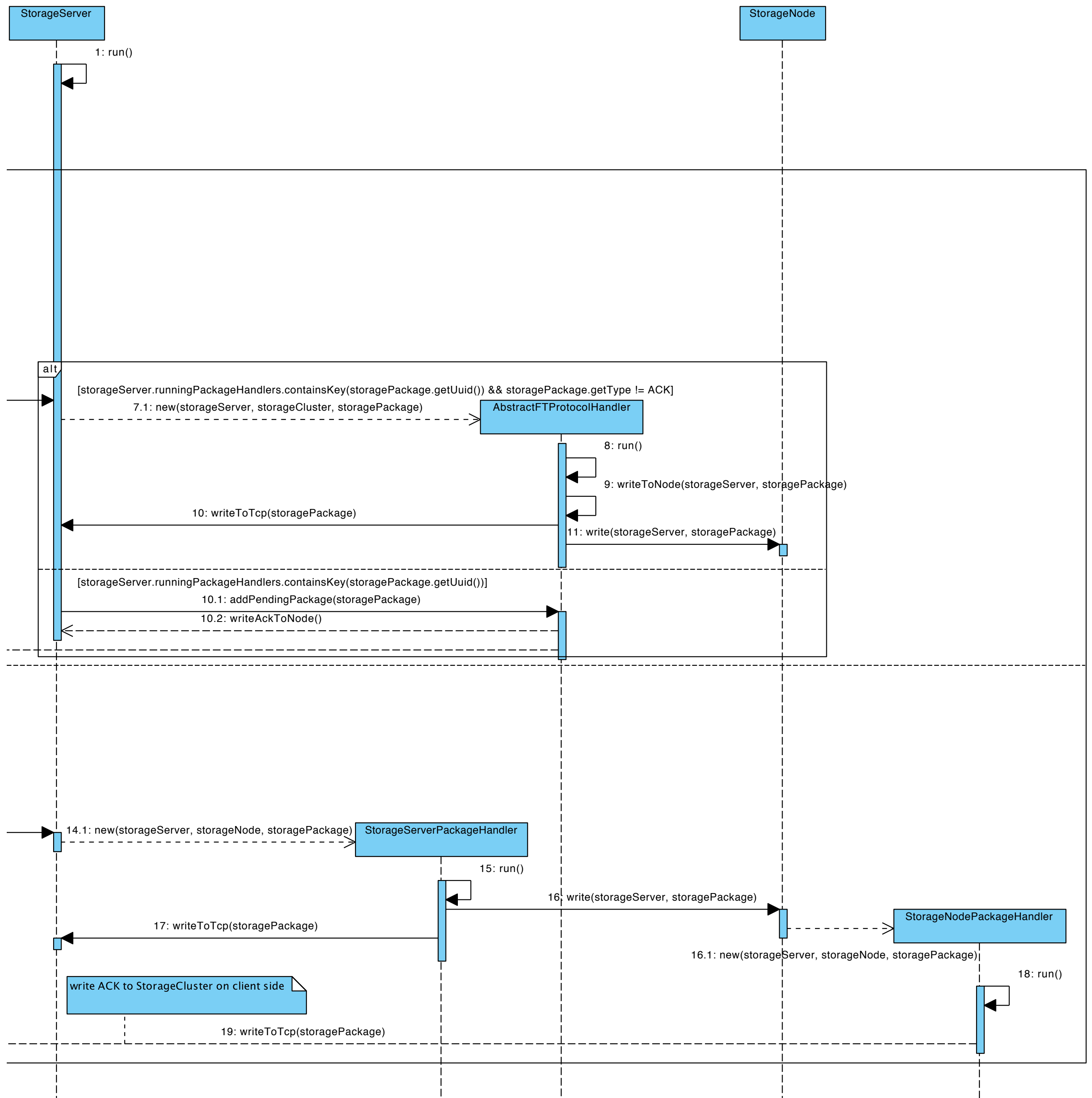


Figure 3.5: Framework sequence diagram — Server-side

Hashing is used for fast verification of data equivalence. For example, when a server node sends a confirmation to the client that a `StoragePackage` has been stored, it may send the data's digest instead of the full data. Coding is used to save space on each server node, since the full data may be recovered by reading from multiple server nodes and combining/decoding it. The pseudocode of the algorithm for client- and server-side reading and writing can be found in listings 4.1-4.4. A high-level description of the protocol follows in section 3.3.

3.3 Implementation

When a `StoragePackage` p is to be stored remotely, it is written by a client to every locally defined `StorageCluster`. Every `StorageCluster` then executes a package handler respective to its defined protocol, and the client waits for responses from each cluster in these individual threads. In the case of a `StorageCluster` employing `DefaultFTProtocol`, the respective subclass of `AbstractFTProtocolHandler` bearing the same name as the protocol, is instantiated. The full data is then written to the primary server S_0 of each cluster using `DefaultFTProtocol`. The protocol proceeds as follows:

If the client receives more than $f + 1$ faulty responses from the servers in a given cluster, or if a timeout is reached before $f+1$ non-faulty packages have been received, the client may reconfigure the cluster by choosing a new S_0, and then writes p to the newly chosen S_0. From the client's perspective, a non-faulty package is one that matches the data of the original package p.

After S_0 has received p, it starts the first round of communication among the servers by sending p_i to every backup server S_i, where p_i denotes the first package received by S_i from S_0. p_i is then sent by S_i to all other $S_j, j \neq i$.

Once the client receives $f + 1$ non-faulty packages, it is marked as non-speculative and sends an acknowledgment to all S_i to commit the `StoragePackage`, and likewise, mark it as non-speculative on the server-side. At this point, it can be assured that the data has been received by at least $2f + 1$ nodes. This is because a non-faulty server node will only send an acknowledgment to the client once $2f$ packages have been received and at least f of those are non-faulty. From a server's perspective, a non-faulty package corresponds with p_i, the first package received from S_0, i.e. the package initially received starting the first round of communication.

In the case when reading, a similar execution occurs as when writing. The client sends a `StoragePackage` of `StoragePackageType READ` and containing the identifying `UUID` of p to S_0. S_0 then forwards the same `StoragePackage` to all $S_i, i \neq 0$. If S_i has previously committed a package with the given `UUID`, S_i returns p_i to the client.

Otherwise, if S_i has not committed a package with the corresponding `UUID`, S_i requests p_j from all $S_j, j \neq i$, as well as p_i from S_0. p_i will be committed if $f + 1$ non-faulty responses have been received within a given time limit, and p_i will be sent to the client. If more than f faulty packages have been received, or if the read times out on the server, a non-faulty server will not respond to the client's request.

Note that the case when more than f faulty packages have been received is only possible when S_0 is faulty. Once the client has received $f + 1$ matching and non-speculative packages, it can be assured that the data matches what the client sent originally, since a non-faulty server will never mark a package as non-speculative if it has not been committed by the client.

Since many clients may read and write to the system, it is possible that two different clients write conflicting `StoragePackage`s with the same `UUID` at the same time. In this case, a server node will prefer the `StoragePackage` with a lower timestamp, i.e. the one that has been received first from a

client C_0. If a server receives a write request from a different client C_1, and the `StoragePackage` from C_0 has not yet been committed, then server will send a notification to C_1 with C_0's original request. C_1 may then choose to accept the data written by C_0 or attempt to overwrite it at a later point.

This same logic for writing data is used when reconfiguring a `StorageCluster`. For example, let us assume that a client C_0 chooses to reconfigure a `StorageCluster`. Before the request has been committed, a different client, C_1, sends a reconfiguration request to the primary S_0. If C_1 sends the request to any other server, the request will be dropped, because any $S_i, i \neq 0$ will only accept requests from S_0. Likewise, S_0 accepts any write requests that are not from S_i. S_0 will recognize that a different client has made a request with the same `UUID` and will respond to C_1 with the current *speculative* data. C_0's request will continue as expected, and C_1 may choose to send its request again, use the speculative data, or send a read request. In the best case, the read request will result in at least $f + 1$ matching responses. If this is not the case, then the write request from C_0 has not yet been committed.

3.4 Testing

3.4.1 Environment

The test cases are to be performed on a Dell PowerEdge T420 with the following hardware specs:

- 2×Intel® Xeon® CPU E5-2420 @ 1.90GHz
- 24 GB RAM @ 1333MHz
- 2×Western Digital WD1003FBYX-1 HDD

The host will run CentOS 6.4 and a QEMU/KVM hypervisor, in which virtual machines run the same operating system as the host. The virtual machines are each allocated with one (virtual) CPU and 512 MB RAM.

3.4.2 Plan

The tests will compare the performance of clusters under different configurations ranging from $f = 0$ until $f = 4$. Each cluster is designated by the names C00-C04, where the number in the cluster's name represents the total number of tolerated faults. Each cluster consists of `StorageNodes`, and the entire test configuration will consist of a total of $4 + 4 + 7 + 10 + 13 = 38$ `StorageNodes`, which are named N00-N37. N00-N03 are grouped into C00, C04-C07 into C01, and so forth. A class `ConfigUtil` has been created for the purposes of generating the configuration files for the clusters and `StorageNodes`. `ConfigUtil` takes as one of its parameters f and generates $f + 1$ clusters, where one of the clusters does not assume any faults. The remaining clusters are defined with $3f + 1$ nodes accordingly, whereby tolerating f failures.

The tests will not include any explicitly faulty nodes. However, given the correctness of the algorithm 3.3, we can assume that at most f nodes are faulty, and that the tests work correctly under these circumstances. We will test the write speed of the different clusters by writing data packages of various sizes incrementing by factors of 10, starting from 1 kB until 10000 kB.

In order to determine the write speed for a `StoragePackage` of a given size in a cluster, we subtract the time it takes from starting the respective `StoragePackageHandler` on the client side until $f+1$

corresponding matching packages are received from the servers. We will use Apache Log4j 2[2] to log the *start* and *end* times for writing a given `StoragePackage`.

3.4.3 Expected Results

We expect that the that the write times increase linearly with increased fault-tolerance and data, i.e. the write time changes linearly with f, as well as with the amount of data written. The baseline for measurements will be writing a 1kB StoragePackage to *C00*, the cluster not supporting any faults. The expected write-time factors, relative to the baseline benchmark, take into account the number of servers in a given cluster and amount of data written. For example, when writing the same amount of data, 1kB, to C04, the cluster supporting up to four faults, we expect that it will take approximately thirteen times longer than writing to C00, since there are thirteen[4] nodes in C04. We do expect that there is a minimum threshold for writing data to a given cluster, i.e. a constant factor caused by overhead that makes writing 1kB of data take as long as 10kB, for example. This constant factor, as well as the constant linear factor, are most likely affected when the system uses hashing.

Given the difficulty of estimating these factors and possibilities of fluctuations in the system, we will have five runs for each data package with a size of 1, 10, 100, 1000, and 10000 kB, using both MD5 hashing and no hashing to verify the data packages. With a total of five clusters, C00-C04, supporting zero to four faults, this amounts to 5 runs $\times$ 5 data packages $\times$ 2 hashing schemes $\times$ 5 clusters $= 250$ data points.

In order to minimize the chances of cached data, which could skew the results, we will generate random data for each run when writing a data package. For this purpose, we will make use of a random *lorem ipsum* text generator[21].

3.4.4 Execution and Analysis

In tables 3.1 and 3.2 we see the results of the required time, in seconds, to write a storage package of a particular size, with and without the use of hashing, to a given storage entity. It should be pointed out that the results for C00 is the average of the write times of the individual storage nodes, N00-N03, in this cluster. The write executions were performed in the same order as they are in the table, from top to bottom.

One can notice that the first two executions, when writing 1kB to the clusters, are significantly higher than the following executions, when writing a package of the same size. For this reason, we performed two additional executions with a 1 kB data package, for a total of seven results for 1 kB of data in each case, with and without hashing. Since this phenomenon is noticeable in both of these cases, and since these cases were performed with a new execution of the program, i.e. stopping and restarting it, we believe that this anomaly is caused by some caching mechanism in the Java Virtual Machine[38], or by the fact that some classes are only loaded when they are first called[20]. Because of this, we believe that these first data points do not represent the true performance of the system, and therefore create a set of *corrected* data points, in which the first two executions are removed.

For comparison reasons, we show the graphical representation of the corrected and uncorrected data grouped by clusters in figures 3.6-3.17. Furthermore, we present a log-log plot of the same data in figures 3.18-3.21.

[4] $3f + 1$

package size (kB)	N00	N01	N02	N03	C00	C01	C02	C03	C04
1	0.291	0.291	0.297	0.297	0.295	0.86	1.182	1.385	1.386
1	0.245	0.245	0.245	0.245	0.245	1.064	1.073	1.333	1.458
1	0.038	0.038	0.038	0.038	0.038	0.201	0.557	0.376	0.652
1	0.054	0.055	0.055	0.055	0.055	0.194	0.477	0.421	0.541
1	0.029	0.029	0.029	0.029	0.029	0.261	0.345	0.417	0.628
1	0.031	0.031	0.031	0.034	0.032	0.148	0.261	0.352	0.459
1	0.067	0.069	0.069	0.071	0.0696667	0.124	0.228	0.383	0.393
10	0.047	0.047	0.055	0.055	0.0523333	0.186	0.328	0.495	0.547
10	0.045	0.045	0.045	0.045	0.045	0.174	0.439	0.507	0.519
10	0.025	0.028	0.028	0.029	0.0283333	0.245	0.31	0.418	0.504
10	0.033	0.033	0.033	0.033	0.033	0.153	0.279	0.467	0.587
10	0.042	0.042	0.042	0.042	0.042	0.179	0.265	0.392	0.461
100	0.046	0.046	0.046	0.046	0.046	0.367	0.466	0.791	1.015
100	0.049	0.049	0.049	0.049	0.049	0.365	0.6	0.784	1.008
100	0.045	0.046	0.046	0.046	0.046	0.306	0.57	0.799	0.989
100	0.047	0.048	0.047	0.048	0.0476667	0.262	0.596	0.706	1.009
100	0.058	0.058	0.058	0.058	0.058	0.247	0.495	0.791	1.127
1000	0.28	0.281	0.28	0.281	0.280667	1.74	2.768	3.444	4.79
1000	0.29	0.353	0.292	0.292	0.312333	1.396	3.344	3.66	4.963
1000	0.285	0.302	0.285	0.287	0.291333	1.828	2.497	3.688	4.733
1000	0.271	0.271	0.271	0.273	0.271667	1.289	2.691	3.251	5.183
1000	0.283	0.283	0.283	0.284	0.283333	1.595	2.631	3.88	4.08
10000	1.948	1.948	1.953	1.953	1.95133	8.497	16.645	26.762	32.432
10000	1.658	1.658	1.658	1.662	1.65933	10.548	17.347	28.161	32.676
10000	1.43	1.431	1.423	1.43	1.428	9.07	15.262	26.975	31.924
10000	1.773	1.773	1.768	1.772	1.771	8.029	17.161	23.664	32.636
10000	1.665	1.671	1.671	1.671	1.671	10.653	13.518	28.129	35.204

Table 3.1: Package Write time per storage entity in seconds without hashing.

We observe in figures 3.6-3.11 and 3.12-3.17 that the shape of the line, i.e. the slope, is affected minimally by removing the invalid data points. Nevertheless, the outliers are still apparent for both cases when using hashing and no hashing.

In figures 3.22-3.25 we present the write times with respect to the number of tolerated failures. For better clarity of this relation between all clusters, we present the same data in log-log plots, which are shown in figures 3.26-3.29.

The line for C00 in figures 3.8 and 3.9 is nearly horizontal, meaning that it takes nearly the same amount of time to write 1 kB of data as it does to write 100 kB of data, which is most likely due to overhead of the system. This phenomenon is also observable in figures 3.14 and 3.15, in the case when hashing is used.

One noticeable difference between the cases of hashing versus no hashing is that hashing greatly reduces the slope of the line. That is, when using hashing, the rate at which the expected time to write increases more slowly with respect to the size of the data package, compared to the case without the use of hashing. However, the points in the hashing case have a higher error margin.

This phenomenon could be explained by the possibility that it takes a different amount of time to compute the hash values of different data packages, even if they are the same size. Moreover, in the

package size (kB)	N00	N01	N02	N03	C00	C01	C02	C03	C04
1	0.228	0.229	0.227	0.229	0.228333	0.845	1.21	1.177	1.247
1	0.034	0.034	0.034	0.034	0.034	0.136	0.368	0.722	0.659
1	0.054	0.054	0.054	0.053	0.0536667	0.136	0.49	0.216	0.571
1	0.022	0.022	0.022	0.022	0.022	0.26	0.288	0.544	0.32
1	0.025	0.025	0.025	0.025	0.025	0.156	0.159	0.398	0.468
1	0.024	0.024	0.024	0.024	0.024	0.08	0.173	0.415	0.403
1	0.033	0.033	0.03	0.043	0.0353333	0.098	0.357	0.178	0.368
10	0.024	0.021	0.024	0.021	0.022	0.101	0.131	0.387	0.42
10	0.025	0.02	0.02	0.02	0.02	0.335	0.496	0.348	0.427
10	0.054	0.057	0.054	0.054	0.055	0.157	0.17	0.307	0.509
10	0.035	0.036	0.039	0.035	0.0366667	0.093	0.139	0.25	0.487
10	0.023	0.023	0.025	0.024	0.024	0.139	0.157	0.473	0.326
100	0.034	0.034	0.034	0.034	0.034	0.564	0.451	0.504	1.15
100	0.047	0.047	0.047	0.047	0.047	0.182	0.907	0.759	1.019
100	0.04	0.04	0.04	0.04	0.04	0.169	0.556	0.455	1.199
100	0.05	0.05	0.05	0.05	0.05	0.361	0.609	0.697	0.848
100	0.047	0.047	0.047	0.047	0.047	0.565	0.458	0.729	1.144
1000	0.146	0.147	0.146	0.144	0.145667	0.777	1.857	3.726	4.594
1000	0.163	0.163	0.161	0.162	0.162	1.48	2.442	3.836	4.306
1000	0.137	0.137	0.135	0.135	0.135667	1.681	2.029	4.183	4.26
1000	0.157	0.163	0.157	0.154	0.158	1.165	2.804	2.504	4.65
1000	0.271	0.271	0.27	0.27	0.270333	1.598	2.611	3.96	4.767
10000	1.815	1.789	1.819	1.812	1.80667	8.034	17.697	27.043	36.997
10000	1.879	1.898	1.898	1.898	1.898	7.873	16.385	23.113	32.184
10000	1.693	1.697	1.693	1.693	1.69433	9.157	16.02	28.839	38.673
10000	1.847	1.854	1.849	1.854	1.85233	7.903	16.051	24.792	35.087
10000	1.864	1.744	1.743	1.743	1.74333	13.533	15.72	24.527	40.789

Table 3.2: Package Write time per storage entity in seconds with hashing.

case when no hashing is used, it could be that the memory shared between the virtual machines is used more efficiently[19], meaning that less CPU cycles are needed to compute and access data. Since all virtual machines, that is, the individual nodes N00-N37 which represent the remote storage systems, are sharing the CPUs of the host, the CPU scheduler could be another source for the margin of error, if it schedules the individual virtual machines in an “unfair” manner.

Another observation we can make in figures 3.9 and 3.15 is that, for each cluster, it does indeed take approximately the same amount of time to write 1 kB as it does to write 10 kB of data. For C00 and C01, this is even the case when writing data packages of 100 kB in size. This observation is better illustrated in figures 3.18-3.21, which show the relation of writing a package of all sizes within a given cluster. It is also interesting to note that the rate at which the write time increases is very marginal when writing data packages of sizes 1 kb-100 kB.

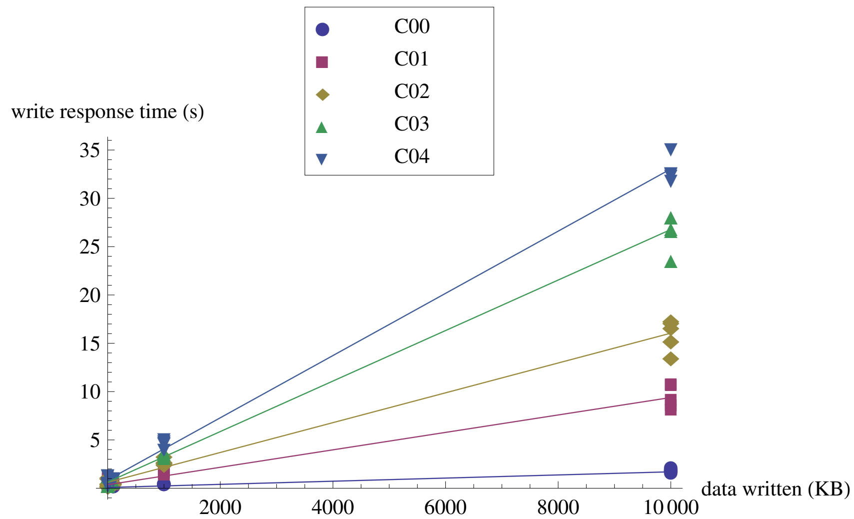


Figure 3.6: Write time for data packages without hashing, grouped by cluster.

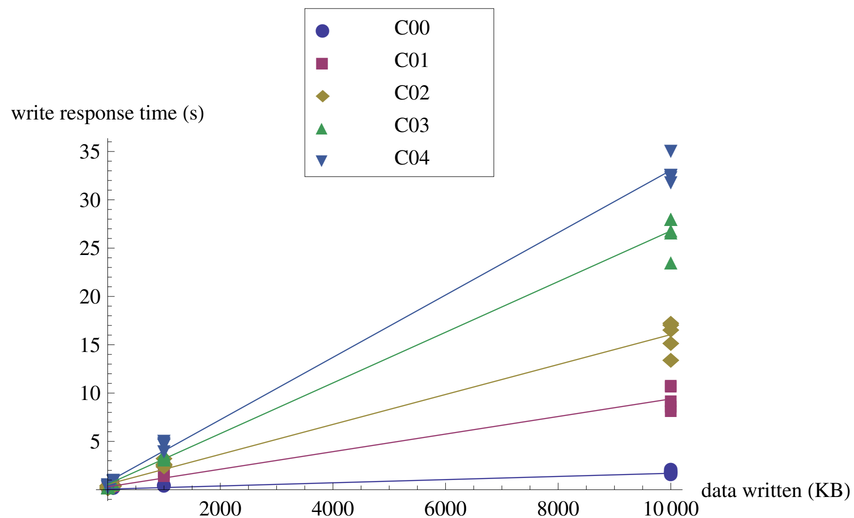


Figure 3.7: Write time for data packages without hashing using corrected data, grouped by cluster.

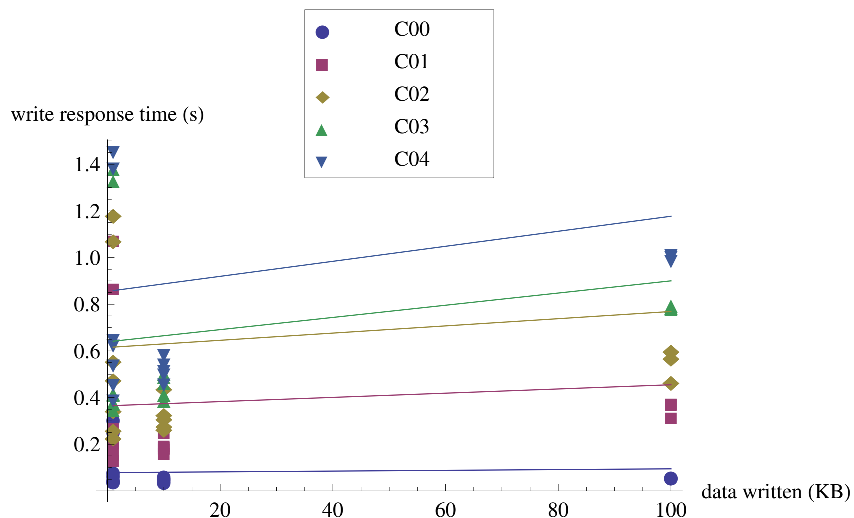


Figure 3.8: Write time for 1 kB-100 kB data packages without hashing, grouped by cluster.

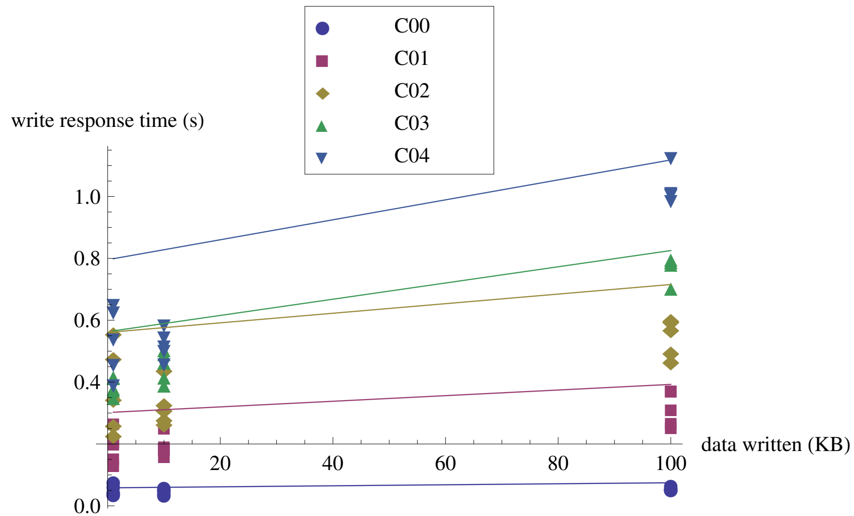


Figure 3.9: Write time for 1 kB-100 kB data packages without hashing using corrected data, grouped by cluster.

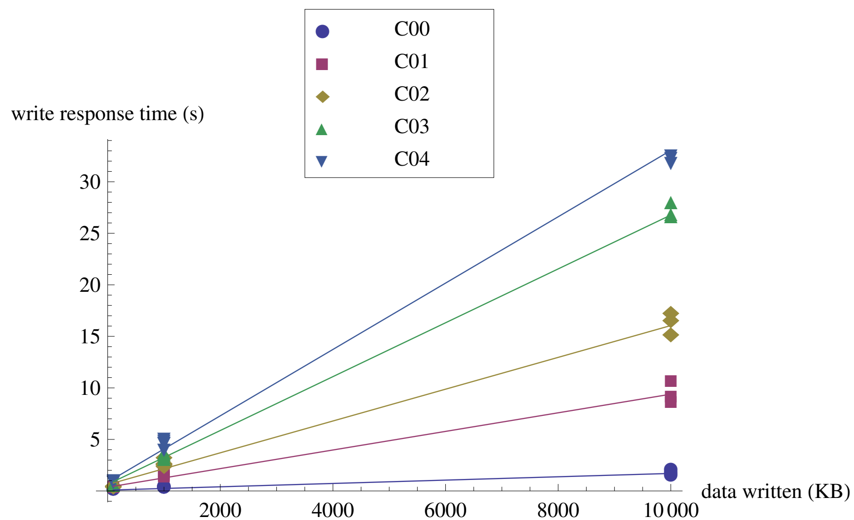


Figure 3.10: Write time for 100 kB-10000 kB data packages without hashing, grouped by cluster.

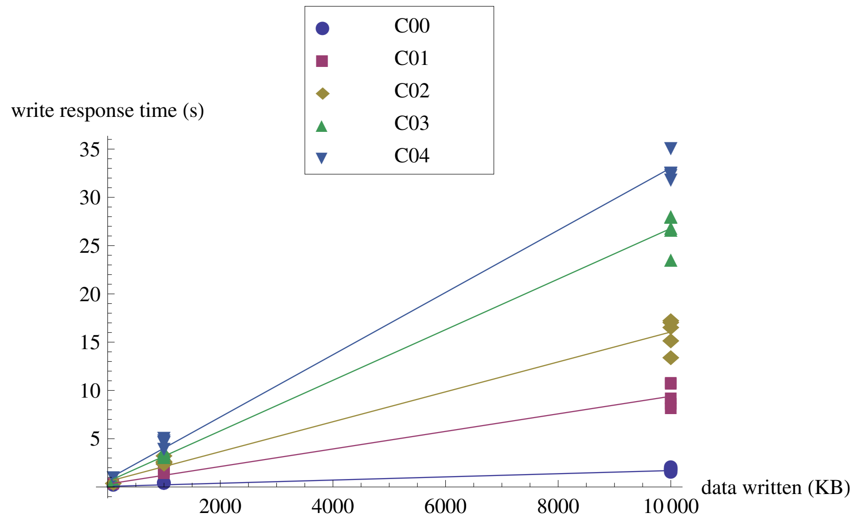


Figure 3.11: Write time for 100 kB-10000 kB data packages without hashing using corrected data, grouped by cluster.

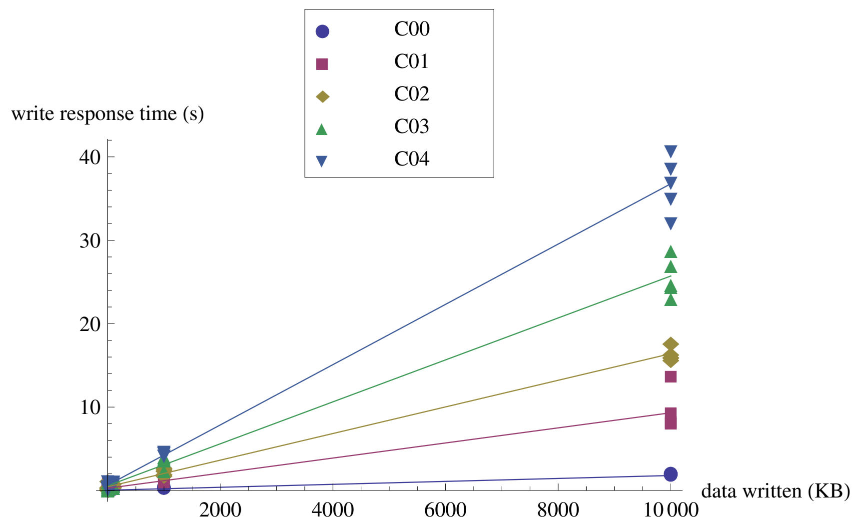


Figure 3.12: Write time for data packages with hashing, grouped by cluster.

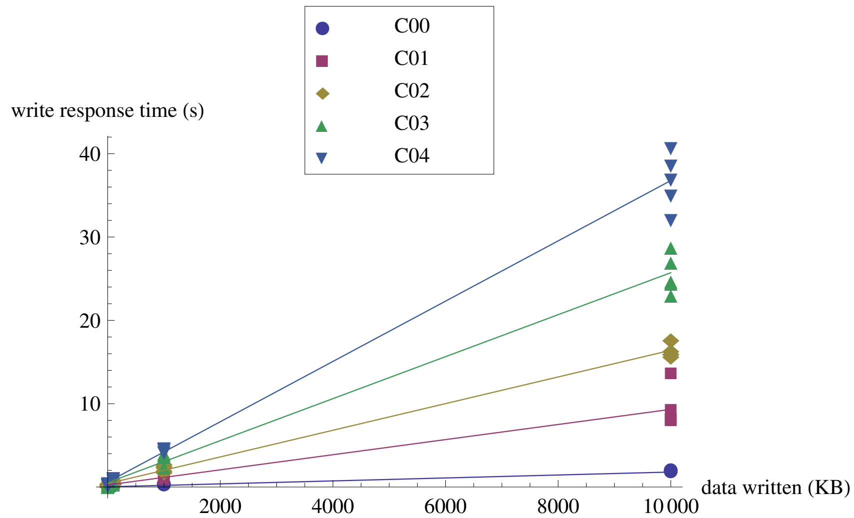


Figure 3.13: Write time for data packages with hashing using corrected data, grouped by cluster.

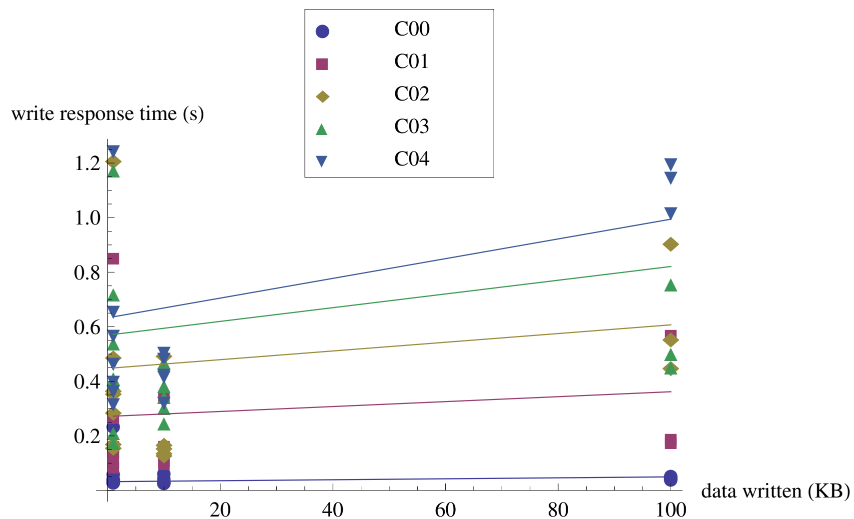


Figure 3.14: Write time for 1 kB-100 kB data packages with hashing, grouped by cluster.

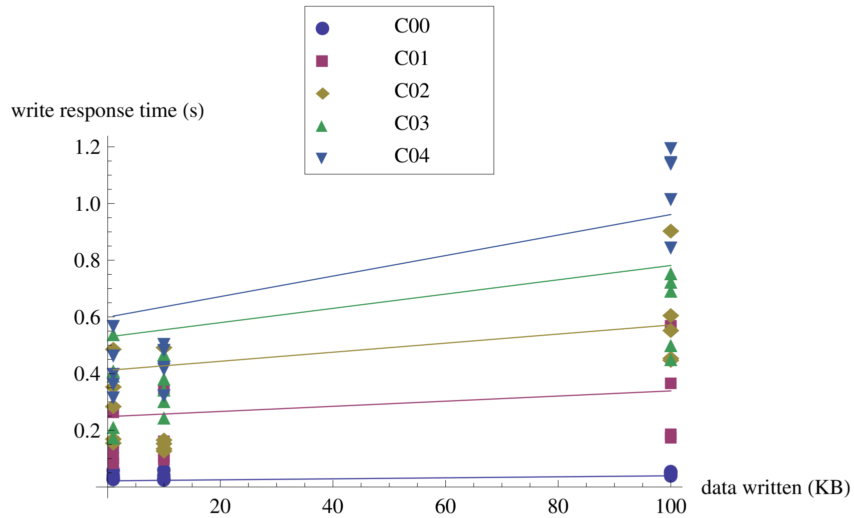


Figure 3.15: Write time for 1 kB-100 kB data packages with hashing using corrected data, grouped by cluster.

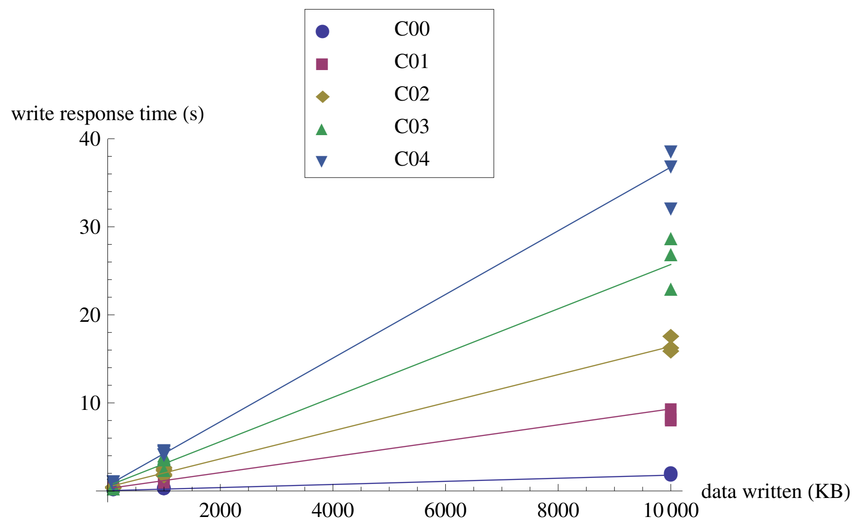


Figure 3.16: Write time for 100 kB-10000 kB data packages with hashing, grouped by cluster.

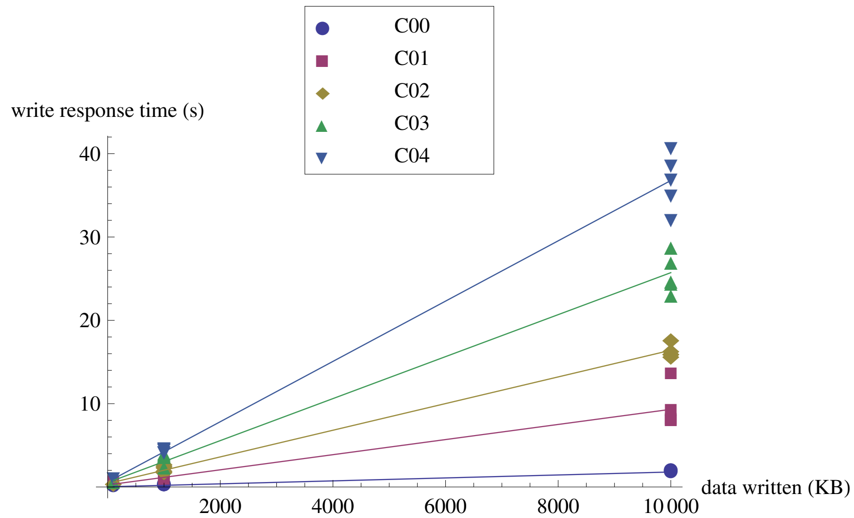


Figure 3.17: Write time for 100 kB-10000 kB data packages with hashing using corrected data, grouped by cluster.

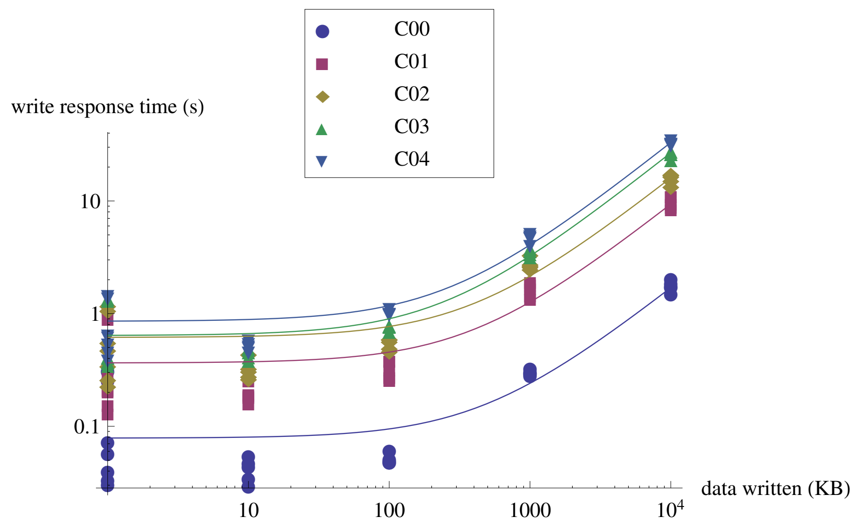


Figure 3.18: Write time for data packages without hashing, grouped by cluster.

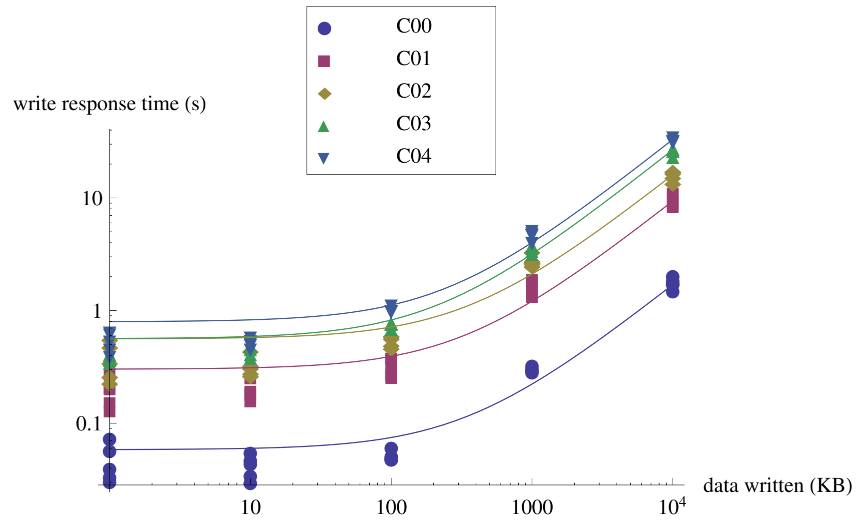


Figure 3.19: Write time for data packages without hashing using corrected data, grouped by cluster.

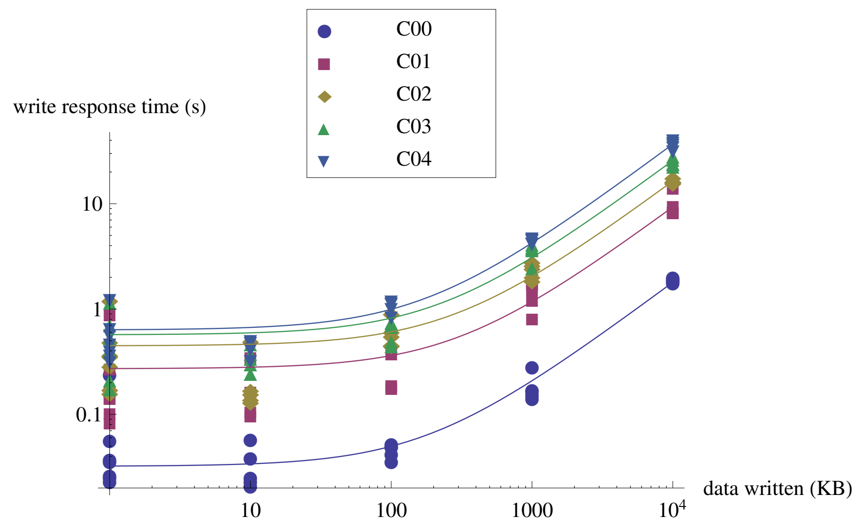


Figure 3.20: Write time for data packages with hashing, grouped by cluster.

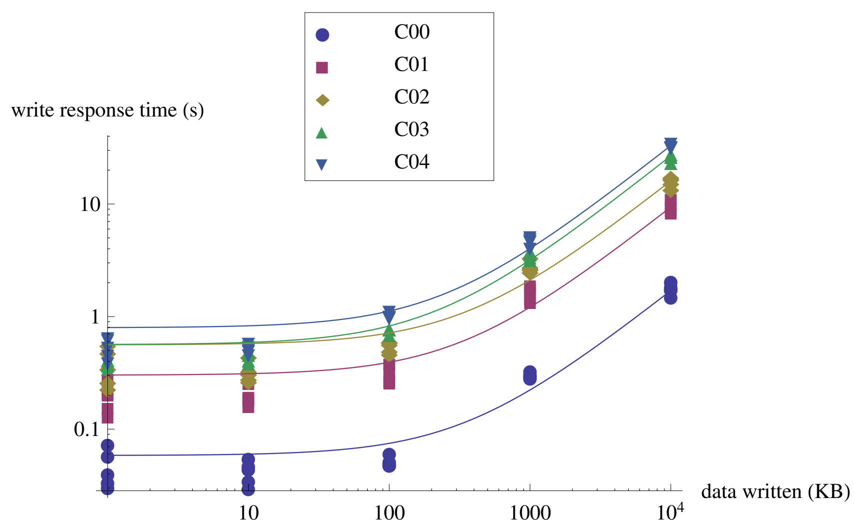


Figure 3.21: Write time for data packages with hashing using corrected data, grouped by cluster.

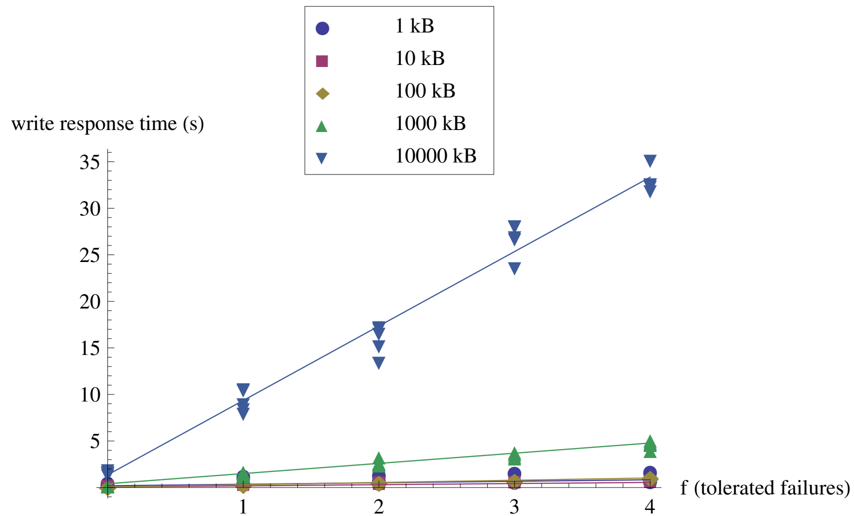


Figure 3.22: Write time versus the number of tolerated failures for data packages without hashing, grouped by data package size.

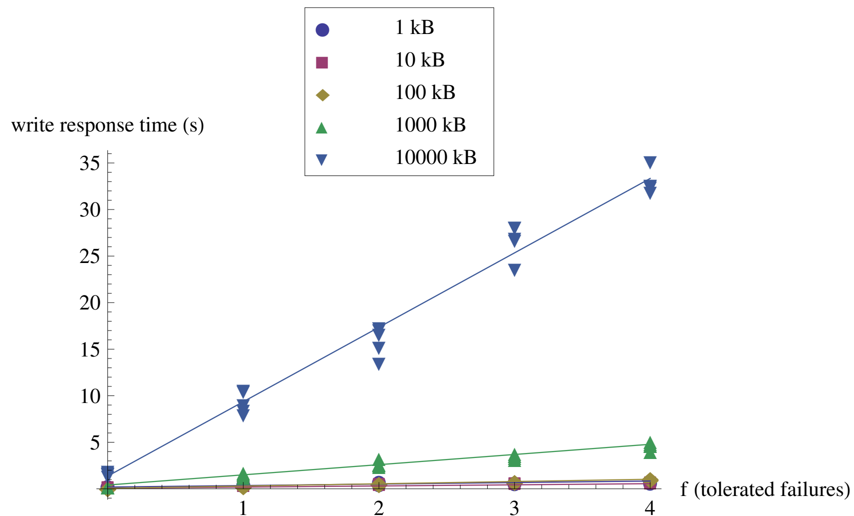


Figure 3.23: Write time versus the number of tolerated failures for data packages without hashing using corrected data, grouped by data package size.

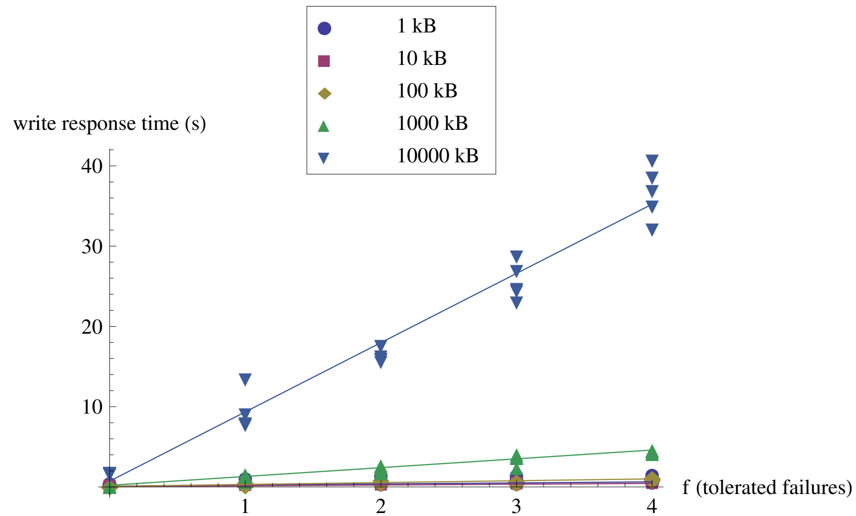


Figure 3.24: Write time versus the number of tolerated failures for data packages with hashing, grouped by data package size.

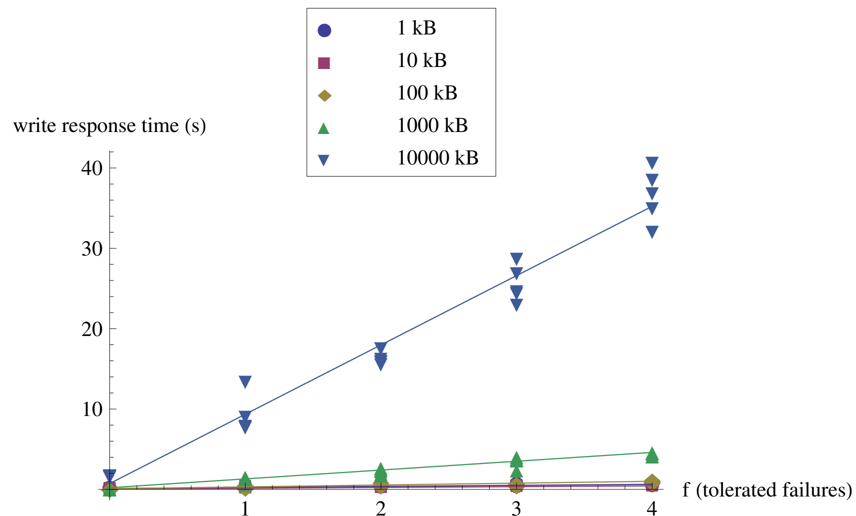


Figure 3.25: Write time versus the number of tolerated failures for data packages with hashing using corrected data, grouped by data package size.

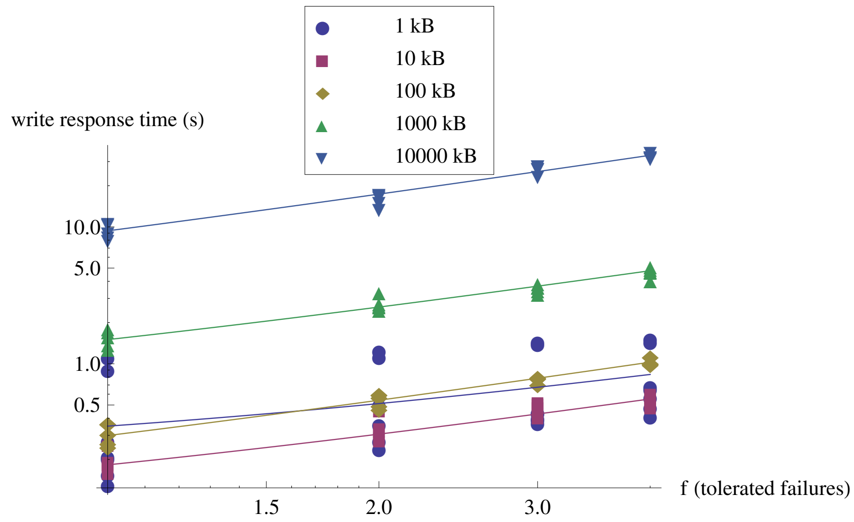


Figure 3.26: Write time versus the number of tolerated failures without hashing, grouped by data package size.

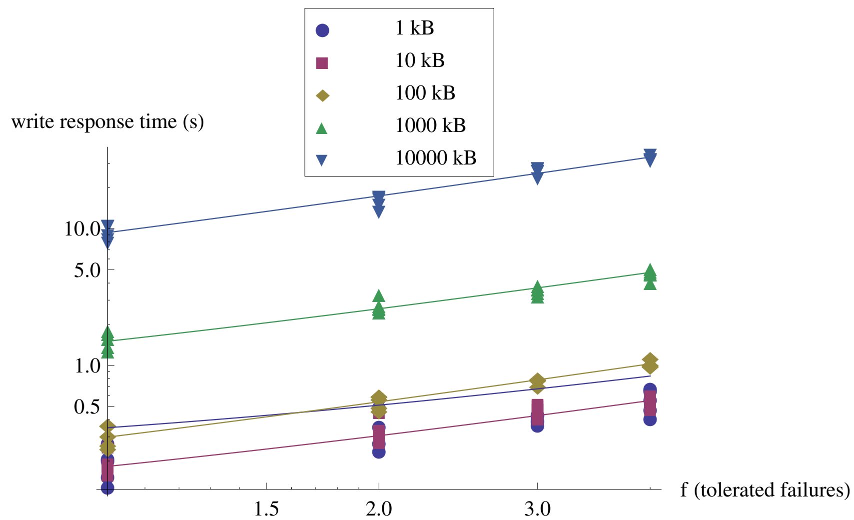


Figure 3.27: Write time versus the number of tolerated failures without hashing using corrected data, grouped by data package size.

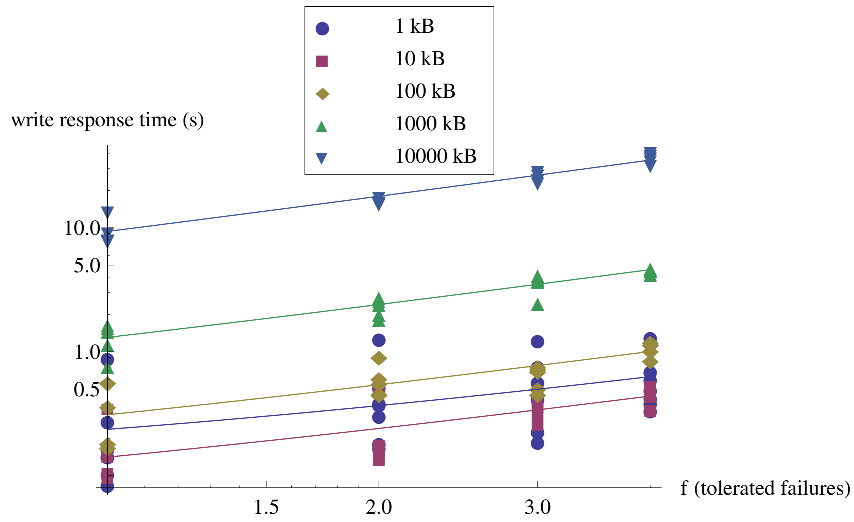


Figure 3.28: Write time versus the number of tolerated failures with hashing, grouped by data package size.

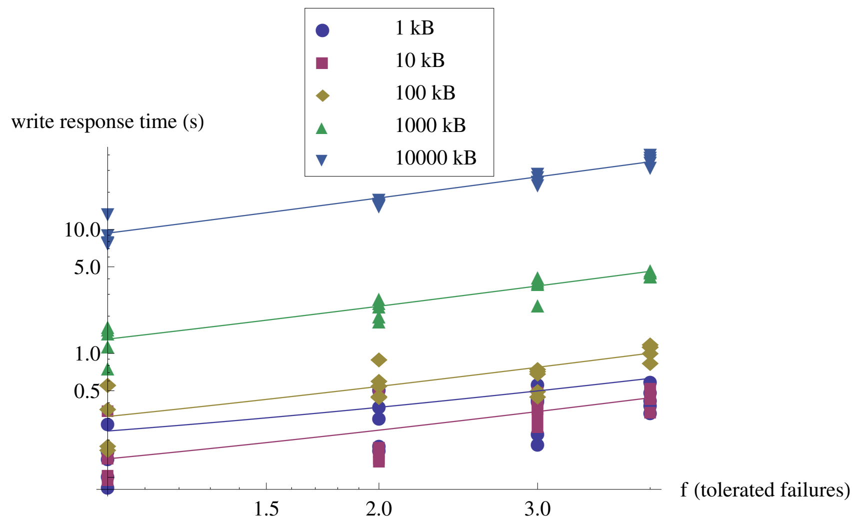


Figure 3.29: Write time versus the number of tolerated failures with hashing using corrected data, grouped by data package size.

4 Algorithms

4.1 DefaultFTProtocol

4.1.1 Pseudocode

Following is the default fault-tolerant algorithm implementation which supports coding, as well as fast data verification, i.e. using hash values to quickly compute the equivalence of data packets. The algorithm is divided into four sections, a client- and server-side for both writing and reading a `StoragePackage`. We use the following definitions within the algorithm:

node An instance of `StorageServer`.

f Maximum number of failed nodes in a cluster that the algorithm can handle.

N $\geq 3f + 1 = \|$`StorageCluster`$\|$, the number of nodes in a cluster.

S_i Reference to a particular node, where S_0 is the primary node of any given cluster.

p The `StoragePackage` containing the original data sent from the client, which is to be stored remotely.

p_i Reference to the particular `StoragePackage` stored on node S_i. Note that when no coding is used, p_i's encapsulated data equals p in the non-faulty case.

4.1.2 Graphical Example

In this section we provide a graphical representation of our default fault-tolerant protocol. We show a total of three scenarios for $f = 1$, and one for $f = 2$:

- Writing in the case when the primary server is faulty,
- writing in the case when one of the secondary servers is faulty,
- writing in the case when the primary server and one of the secondary servers is faulty, and
- reading in the case when the primary server is faulty.

For each scenario we provide an explanation in each step, argue the correctness of the algorithm.

Because of its triviality, we do not show the reading case with a faulty secondary server.

```
input  : The StoragePackage p to be stored.
output : Confirmation that p has been stored.
1  foreach StorageCluster do
2      write p to S_0;
3  end
   // for each StorageCluster, wait for ACKs in separate
      threads...
4  packages_verified ← 0;
5  packages_processed ← 0;
6  verified ← false;
7  while verified = false do
8      p_i ←poll(StoragePackage queue);
9      if p_i is request already made by other client then
10         break current loop;
11     if not coding flag set then
12         if hashing flag set then
13             if hash(p_i)=hash(p) then
14                 packages_verified ←packages_verified +1;
15         else
16             if p_i = p then
17                 packages_verified ←packages_verified +1;
18         packages_processed ←packages_processed +1;
19         if packages_verified ≥ f + 1 then
20             verified ← true;
21     else
22         if hashing flag set then
23             if hash(p_i)=hash(p) then
24                 packages_verified ←packages_verified +1;
25             packages_processed ←packages_processed +1;
26             if packages_verified ≥ f + 1 then
27                 verified ← true;
28         else decoding happens on client side
29             if ||StoragePackage queue|| ≥ f + 1 and packages_processed ≤ f then
30                 packages_processed ←packages_processed +1;
31                 decode data from queue;
32                 if decoding successful then
33                     verified ← true;
34     if (packages_processed − packages_verified) ≥ f + 1 and verified = false or
       timer_expired() then
35         choose new S_0 and write new StorageCluster configuration;
36         if configuration request preceded by another client's then
37             update local configuration;
38         packages_verified ← 0;
39         packages_processed ← 0;
40         write p to S_0;
41 end
```

Algorithm 4.1: DefaultFTProtocol client-side write

```
input  : The StoragePackage p to be stored
output: ACK from at least every non-faulty server to client containing the full data or
        digest of p_i
   // primary node (S_0):
 1 if new write request received from client then
 2     code p into p_0 ... p_{N-1} so that the original data can be recovered by reading f + 1
       different error-free p_i;
 3     foreach S_i, i ≠ 0 do
 4         write p_i to S_i;
 5     end
 6 else write request has already been received from another client, but not yet committed
 7     respond to new client with previous client's request;
   // each S_i:
 8 foreach S_j, j ≠ i do
 9     write p_i to S_j;
10 end
11 packages_verified ← 0;
12 packages_processed ← 0;
13 verified ← false;
14 while verified = false do
15     p_j ←poll(StoragePackage queue);
16     if not coding flag set then
17         if hashing flag set then
18             if hash(p_i)=hash(p_j) then
19                 packages_verified ←packages_verified +1;
20         else
21             if p_i = p_j then
22                 packages_verified ←packages_verified +1;
23         packages_processed ←packages_processed +1;
24         if packages_verified ≥ f + 1 then
25             verified ← true;
26         if packages_processed ≥ f + 1 and verified = false then
27             break current loop;
28     else
29         if ||StoragePackage queue|| ≥ f + 1 and packages_processed ≤ f then
30             packages_processed ←packages_processed +1;
31             decode data from queue;
32             if decoding successful then
33                 verified ← true;
34     if (packages_processed − packages_verified) ≥ f + 1 and verified = false or
       timer_expired() then
35         break current loop;
36 end
37 send ACK to client containing p_i with the full or hash of the (decoded) data;
38 receive ACK from client;
39 if ACK from client matches p_i then
40     remove speculative flag from p_i
41 commit p_i;
```

Algorithm 4.2: DefaultFTProtocol server-side write

```
input  : UUID of a stored StoragePackage
output: The requested StoragePackage p
1  foreach StorageCluster do
2  |  write UUID to S_0;
3  end
   // for each StorageCluster, wait for ACKs in separate
      threads...
4  packages_processed ← 0;
5  verified ← false;
6  H ←new hash table;          // contains received data (at most f+1
   different keys possible)
7  while verified = false do
8  |  p ←poll(StoragePackage queue);
9  |  if not coding flag set then
10 |  |  if hashing flag set then
11 |  |  |  if contains(H, hash(p)) then
12 |  |  |  |  H[hash(p)] ← H[hash(p)] + 1;
13 |  |  |  else
14 |  |  |  |  H[hash(p)] ← 1;
15 |  |  |  if H[hash(p)] ≥ f + 1 then
16 |  |  |  |  verified ← true;
17 |  |  else
18 |  |  |  if contains(H, p) then
19 |  |  |  |  H[p] ← H[p] + 1;
20 |  |  |  else
21 |  |  |  |  H[p] ← 1;
22 |  |  |  if H[p] ≥ f + 1 then
23 |  |  |  |  verified ← true;
24 |  |  packages_processed ←packages_processed +1;
25 |  else
26 |  |  if ||StoragePackage queue|| ≥ f + 1 and packages_processed ≤ f then
27 |  |  |  packages_processed ←packages_processed +1;
28 |  |  |  decode data from queue;
29 |  |  |  if decoding successful then
30 |  |  |  |  verified ← true;
31 |  if packages_processed ≥ f + 1 and verified = false or timer_expired() then
32 |  |  choose new S_0 and write new StorageCluster configuration;
33 |  |  if configuration request preceded by another client's then
34 |  |  |  update local configuration;
35 |  |  packages_processed ← 0;
36 |  |  write UUID to S_0;
37 end
```

Algorithm 4.3: DefaultFTProtocol client-side read

```
   input  : UUID of a stored StoragePackage
   output: The requested StoragePackage p_i
   // primary node (S_0):
 1 foreach S_i, i ≠ 0 do
 2 │ write UUID to S_i;
 3 end
   // each S_i:
 4 if not UUID stored or coding flag set then
 5 │ p_i ← null;
 6 │ foreach S_j, j ≠ i do
 7 │ │ request p_j from S_j;
 8 │ end
 9 │ packages_processed ← 0;
10 │ verified ← false;
11 │ H ←new hash table;       // contains received data (at most f + 1
   │ different keys possible)
12 │ while verified = false do
13 │ │ p ←poll(StoragePackage queue);
14 │ │ if not coding flag set then
15 │ │ │ if hashing flag set then
16 │ │ │ │ if contains(H, hash(p)) then
17 │ │ │ │ │ H[hash(p)] ← H[hash(p)] + 1;
18 │ │ │ │ else
19 │ │ │ │ │ H[hash(p)] ← 1;
20 │ │ │ │ if H[hash(p)] ≥ f + 1 then
21 │ │ │ │ │ verified ← true;
22 │ │ │ else
23 │ │ │ │ if contains(H, p) then
24 │ │ │ │ │ H[p] ← H[p] + 1;
25 │ │ │ │ else
26 │ │ │ │ │ H[p] ← 1;
27 │ │ │ │ if H[p] ≥ f + 1 then
28 │ │ │ │ │ verified ← true;
29 │ │ │ packages_processed ←packages_processed +1;
30 │ │ else
31 │ │ │ if ||StoragePackage queue|| ≥ f + 1 and packages_processed ≤ f then
32 │ │ │ │ packages_processed ←packages_processed +1;
33 │ │ │ │ decode data from queue;
34 │ │ │ │ if decoding successful then
35 │ │ │ │ │ verified ← true;
36 │ │ if packages_processed ≥ f + 1 and verified = false or timer_expired()
   │ │ then
37 │ │ │ break current loop;
38 │ end
39 write p_i to client;
```

Algorithm 4.4: DefaultFTProtocol server-side read

Writing with a Faulty Primary Server

For this scenario, we will assume that the primary server, S_0, is faulty. In figure 4.1, the client writes a package p to S_0.

In order for the servers to reach consensus, S_0 has to send p to at least $2f$ servers. Assume that S_0 does not do this, by sending p', p", and so on, to all secondary servers.[1] In this case none of the secondary servers will ever reach consensus, meaning that the client will time out, and elect a new primary server. If S_0 sends package p to $2f$ servers, then these servers will reach consensus on p.

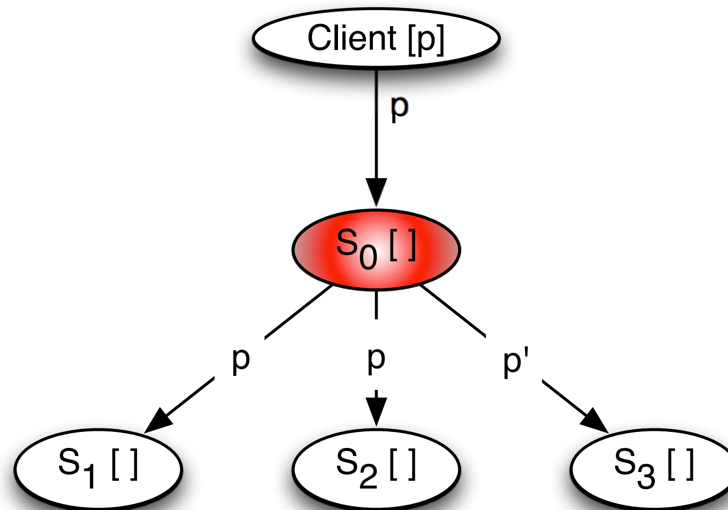


Figure 4.1: Client writing package p to faulty primary server, which then distributes it to the secondary servers.

Figure 4.2 shows the round of servers exchanging the data received from the primary server. After the round of exchanges, S_1 and S_2 have received $2f$ copies of p from $2f$ other servers and store it, whereby S_3 does not store anything, since it has not received $2f$ matching packages.

In figure 4.3, we see that once a non-faulty server has received $2f$ matching packages from $2f$ different servers, it sends an acknowledgment to the client with either the full value of p or the hash value.

The client tells all non-faulty servers in figure 4.4 to commit the original package p, i.e. mark it as non-speculative. Even if S_0 is faulty and stores p' as a committed package, we can still retrieve the original data p when reading it from the servers in a majority vote. Moreover, if S_0 would have sent p' to $2f$ different servers, the client would not respond to those to commit the data. In this case, the client has received more than f faulty packages, and again, a new primary would be chosen and the write phase would be restarted.

If $f + 1$ or more faulty packages p' have been received by the client, i.e. ones that do not correspond to the original package p, it elects a new primary server, since receiving more than f faulty

[1] Note that p', p'', etc, represent faulty packages.

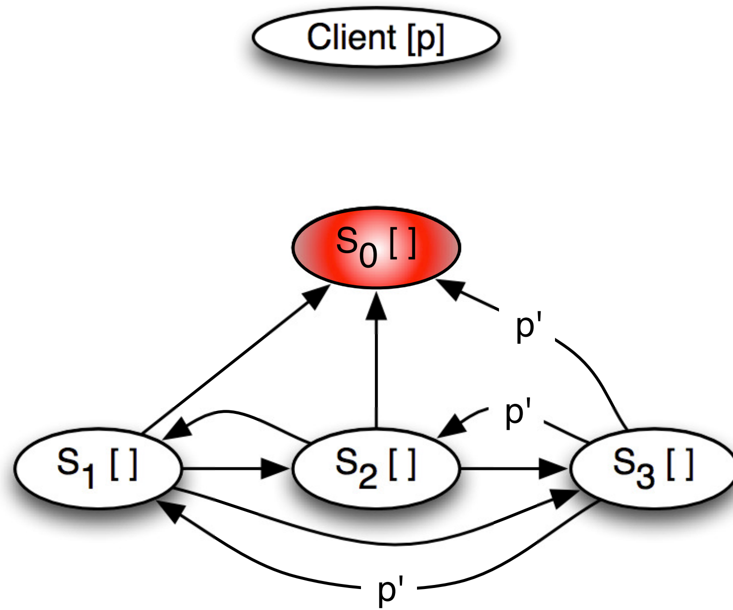


Figure 4.2: Servers exchanging data received from faulty primary.

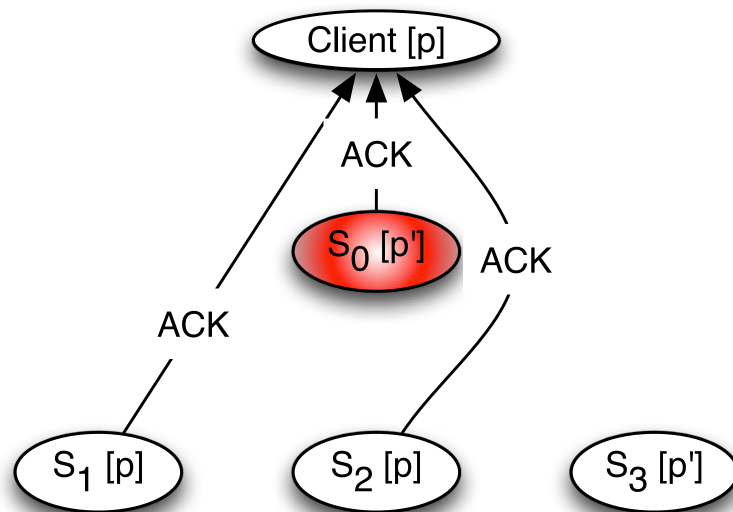


Figure 4.3: Servers responding to client acknowledging receipt of p from faulty primary.

packages is only possible if the primary server is faulty. Also note that the client remembers p until it has been acknowledged by $f + 1$ servers. After the client sends the commit message to the respective servers, we can be assured that at least $2f$ servers have the original data p. This means that when we read, f of those can be faulty, and the majority $f + 1$ of the servers will have the original package p.

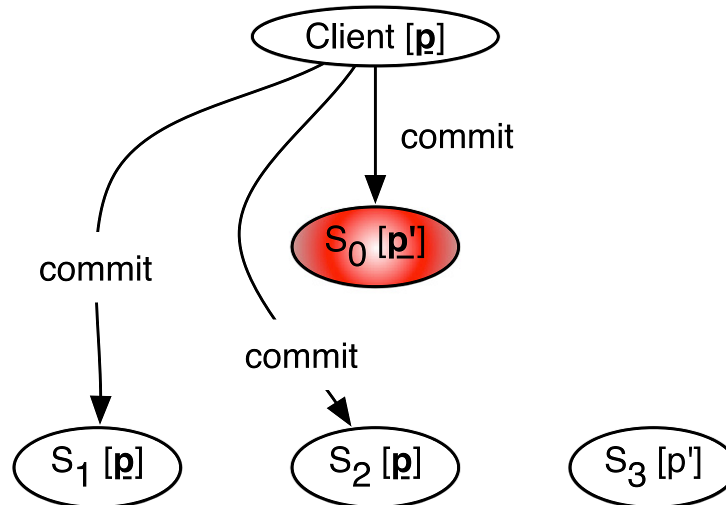


Figure 4.4: Client telling servers to commit.

Writing with a Faulty Secondary Server

In the scenario in figure 4.5, one of the secondary servers is faulty. In this case, it does not matter what data the primary sends the faulty server, since it can be assumed that it is tampered with by the faulty server.

Like in figure 4.2, after the round of exchanges, S_1 and S_2 have received $2f$ copies of p from $2f$ other servers in figure 4.6. In this scenario, S_0 has also received $2f$ copies of p from $2f$ other servers for a total of $2f + 1$ non-faulty servers that have stored p.

The client receives at least $f + 1$ matching packages corresponding to p in figure 4.7, and then replies to all non-faulty servers with a commit message in figure 4.8. It does not matter if S_3 sends an ACK that it has stored a package or not.

Upon the next read, it is clear that we will get at least $f + 1$ matching and committed packages containing the original data p. It also does not affect the read if S_3 marks its stored package p' as committed, since there will never be more than f servers that mark a package p' as such.

Writing with a Faulty Primary Server and one Faulty Secondary Server

We will now look at the case when p is received by the secondary servers, but with $f = 2$, and the faulty servers are the primary and one of the secondary servers. Just like in figures 4.1 and 4.5,

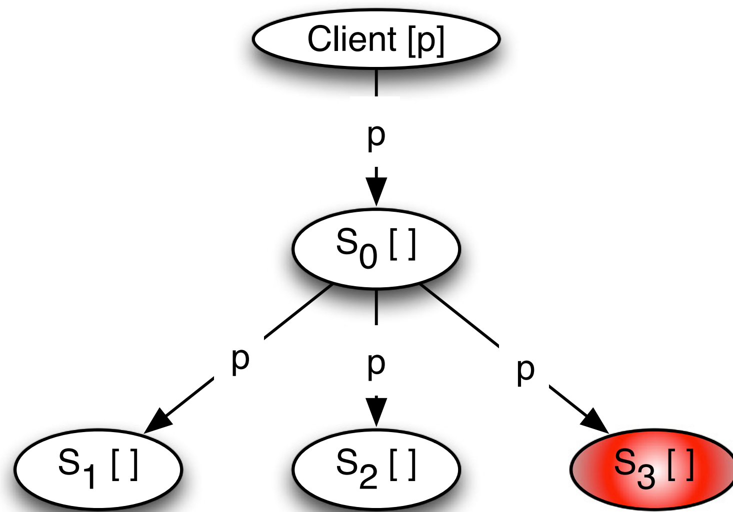


Figure 4.5: Client writing package *p* to the primary server, which then distributes it to the secondary servers, of which one is faulty.

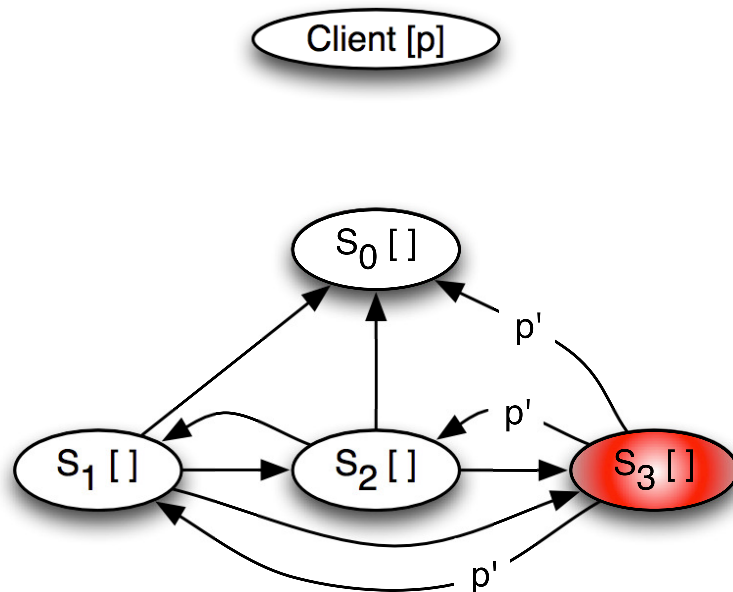


Figure 4.6: Servers exchanging data received from a non-faulty primary.

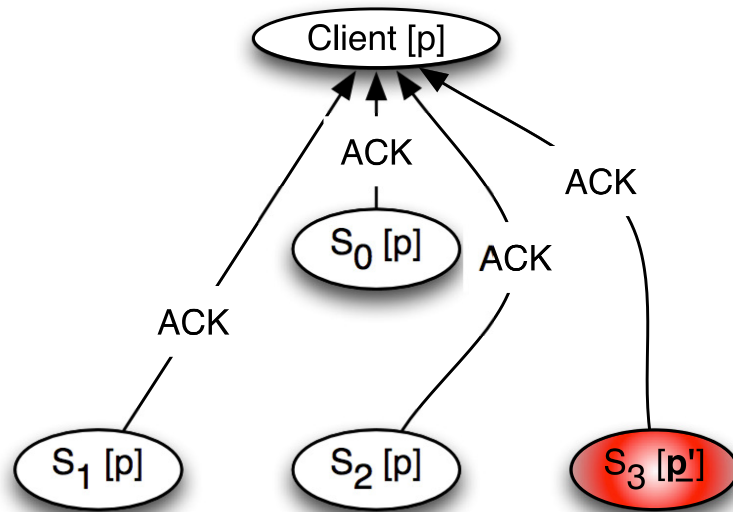


Figure 4.7: Servers responding to client acknowledging receipt of *p* from faulty primary.

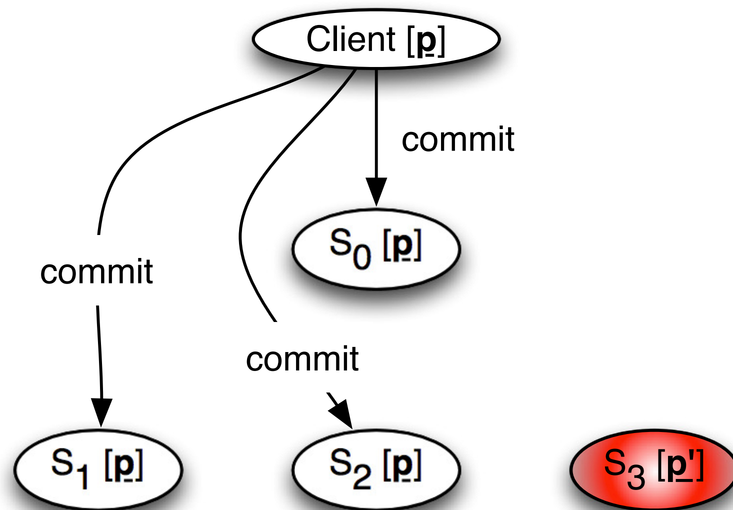


Figure 4.8: Client telling all servers, except for the faulty secondary server, to commit.

the client writes p to the primary server. Like in the previous cases, with $f = 1$, we need at least $2f + 1$ servers with the same package in order to reach consensus.

In the case when $f = 2$, we show that even if only $f + 1$ of the *non-faulty* servers receive p, the majority still reaches consensus and stores p. We can see that there is no way for S_5 and S_6 to store a faulty package p', or at least a package different from $S_2 - S_4$, since we do not meet the requirement that $2f$ other servers have received p'. This means that all non-faulty servers receive and store the same package p, or none at all.

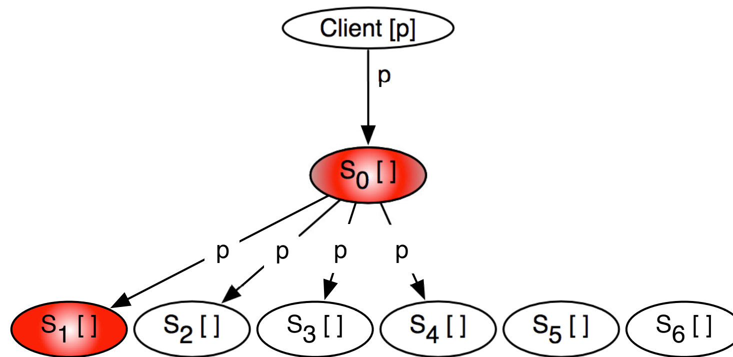


Figure 4.9: Client writing to a cluster with a faulty primary and one faulty secondary server.

Reading with a Faulty Primary Server

In this section we will describe the read scenario. After the client has sent a commit message to at least $f + 1$ servers, it is possible to retrieve the original data. In the scenario in figure 4.10, we will assume that the primary server is faulty and that the package p has been stored correctly as described in section 4.1.2.

If S_0 forwards the read request to all servers, the client waits until it has received $f + 1$ matching *and* non-speculative packages, which will have been received from S_1 and S_2. In all other cases, the client will reach a timeout and elect a new primary:

1. If S_0 forwards the read request to either S_1 or S_2, less than $f + 1$ matching packages will have been received.

2. If S_0 forwards the read request to S_3, S_3 will first send a read request to all servers, since it has not committed its package p'. Once S_3 has received $f + 1$ matching and committed packages from the other servers, it will respond to the client. The client will again get into a timeout, because it is impossible for it to receive $f + 1$ matching and non-speculative packages.

In all cases, as soon as S_3 receives a read request, it will first request the data from the other servers before responding to the read request, meaning that the client will never get more than f matching and non-speculative packages that are invalid.

Since the system supports multiple clients, we need to cover the case when a different client requests p before the first client writing it has committed p. This is easily done with the $speculative$ flag, which is unset after a client sends a commit message and received by a server. With this flag, a different client can easily verify if a received set of packages is speculative or not. In order to be

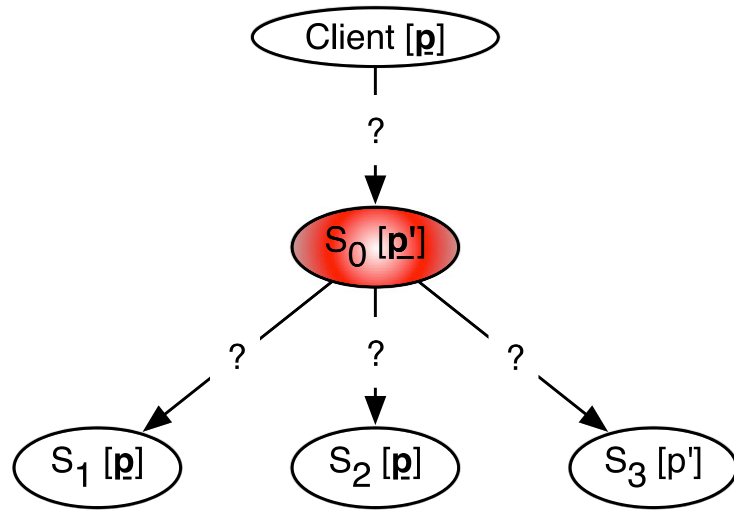


Figure 4.10: Client reading from a cluster with a faulty primary.

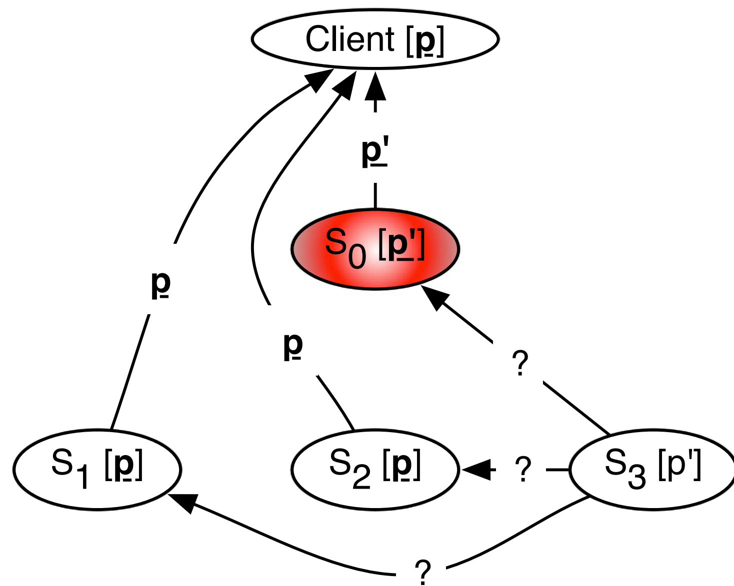


Figure 4.11: Servers responding to client's read request or requesting package from other servers.

certain that a received package is non-speculative, we need to receive at least $f + 1$ matching and non-speculative packages from different servers.

This argument holds true even for $f > 1$. For example, when $f = 2$, even if all of the servers in figure 4.9 receive p' instead of p, a clients will never receive $f + 1$ matching *and* non-speculative packages.

5 Implementation Results

The implementation of the framework in Java allows it to be run on multiple platforms. This can be beneficial in an environment where Byzantine faults are anticipated. Due to the fact that the framework can easily be executed on distinct platforms, a failure on one particular platform would not necessarily occur on a different one. For example, if a particular execution of bits results in an error on one type of CPU due to a hardware bug, one or more CPUs performing the same execution could counterbalance that bug.

Another example where running different platforms could be beneficial is if one particular operating system bears a security bug. Also in this case, if an underlying operating system is subverted by an adversary and is potentially brought to a halt or spews out false information, the uninterrupted systems could compensate for that matter.

The multi-threaded characteristic of the framework allows CPU- and time-intensive tasks to be performed in parallel without inhibiting the main thread of execution. Given the potentially erratic behavior of a system under a Byzantine fault model, different threads could be used to detect or even correct errors. Speculative executions in a BFT system has already been investigated[24], and it is this speculative nature that the framework takes advantage of in the default BFT-protocol implementation. That is, once a server node receives a package, it is immediately stored on its associated storage node. Note that the framework's current implementation status is that only the local storage system can be used.

The package that has been stored at this point is only marked as non-speculative once an acknowledgment from the originating client has been received. Of course, the acknowledgment from the client must correspond with the previously stored package in order to mark it as non-speculative. A corresponding package is one whose data or hash value matches that of the stored package. In the case of a coded package, depending on the coding scheme, one or more corresponding packages may be needed to for a decoding operation in order to detect an error or to retrieve the original data. In fact, under a Byzantine fault model, we would need a minimum of $f+1$ different packages in order to at least *detect* a fault. This is because we cannot assume that one single given package contains valid data.

In order to fully take advantage of the framework and to be able to acquire meaningful benchmarking results, some of the framework's sought features still remain to be implemented and/or debugged: server-side reconfiguration, more storage node types, and coding schemes. Even though the calls to an abstract coding scheme have been integrated into the code, the actual functionality of encoding and decoding by a concrete coding scheme, such as Reed-Solomon[27], still needs to be done.

6 Discussion

Even though we have introduced yet[4] another[39] storage framework for a distributed system, the modular design of the presented framework will hopefully allow future changes and extensions to be implemented more easily. Furthermore, the generality of the framework should be able to incorporate various types of distributed algorithms. Added functionality such as hashing or coding, as well as the ability to tag a `StoragePackage` with abstract metadata, should be flexible and versatile enough to accommodate distributed algorithms of different nature.

When benchmarking the performance of several algorithms, it is utterly important that the same platform is used as a baseline, otherwise the results could be rendered incomparable; seemingly minor differences on one platform could draw an accidental advantage over another platform. For example, if a client reads data from two different remote systems, both of which perform different consensus algorithms before returning a response to the client, one of the remote systems could gain an unfair advantage if the data is kept in memory at all times.

This framework provides a common ground for the purpose of benchmarking the relative performance of distributed algorithms. The algorithms should essentially differ in the operation of distributing data between nodes and in the assumptions made about the validity of data. Of course, it would also be possible to compare the read and write performance of different cloud storage providers if these were to be implemented, though this is not the primary goal of the framework.

Nevertheless, providing an abstract storage facility in the framework allows for more diverse testing environments. For example, many cloud storage providers provide varying degrees of reliability for their storage solutions. Also, different remote storage systems serve varying degrees of response time, which could be another aspect of interest when designing a distributed fault-tolerant protocol.

Although it is difficult and perhaps even impossible to test remote cloud storage solutions in the faulty case, one could make assumptions about the infrastructure and compare the performance in the non-faulty case to that of a local storage solution. That is, one would write to one single node which writes to the cloud storage system. The comparison system would be a cluster of nodes that store to the local file system. Of course the cluster of nodes writing to a local file system would have to provide a similar level of fault tolerance as the cloud storage system. The default fault-tolerant protocol might be appropriate for a comparison of this sort.

One assumption that the framework's default fault-tolerant protocol makes is that clients are non-faulty. Although this assumption may seem bold, it does not impact the stability of the system. A faulty client is considered one if it writes different data values for a given `UUID` of a `StoragePackage` to the server nodes. Since a server node will only accept a given `StoragePackage` as valid if it if it received $2f$ corresponding packages from other server nodes, there is no way for a faulty client to confuse the servers in this sense. That is, a server will only accept a proposed value if there are at least $2f+1$ nodes that have accepted that same value. This means that a client cannot force a non-faulty server to accept invalid data. Furthermore, if f of these $2f+1$ nodes are faulty, a future retrieval of the information will result in the majority always returning the original data.

Another assumption that was made when designing the protocol is that a client's upload rate to servers is limited and that the bandwidth between the servers is significantly faster. For this reason

the client writes the full data only to the primary server of any given cluster and further interaction, e.g. acknowledging receipt of some data, happens directly with the servers.

One disadvantage of the protocol is that as the number of tolerated failures f increases, the amount of data that has to be stored in memory increases. This does not pose to be a problem under a small server load or small packets of data, but as the number of potential failures increases, so does the number of matching packages required to verify their validity. Until a given package has not been verified, it remains in memory.

7 Conclusion

The relation between write time and data size, as well as write time and f, is indeed linear for the implemented fault-tolerant protocol. We managed to boost performance slightly by employing hashing in order to quickly verify data, but the use of storage space is still inefficient. Perhaps future extensions to the program could introduce coding. Not only could this improve the efficiency of the use of storage space, but the performance when employing coding could be compared to the current performance and mechanisms provided by the framework. Such benchmarks could be made even more interesting by comparing various coding techniques, as well as different hashing schemes.

Since the framework was built keeping in mind that it should be able to run on different platforms, a comparison of such configurations could be interesting too. Of course, if the framework were to be run on different operating systems or system architectures, one has to keep in mind that certain calculations used for coding or hashing may perform very differently on each system. Other possibilities for test scenarios would be to benchmark the framework on different physical computers, which was not done for the purpose of the tests performed in this paper. Benchmarking the framework on systems across networks over an Internet link, or laptops creating a mesh Wi-Fi network could prove to be such a setup with more fascinating results.

Furthermore, a completely different distributed protocol could be implemented and tested against the default implementations provided by this framework. The default fault-tolerant protocol implementation was designed to be efficient for use with modern infrastructures; clients have a relatively slow upload rate, whereas servers, which are most likely grouped in a data center, are networked together by extraordinarily fast connections. The default non-fault-tolerant protocol was designed keeping in mind that the read and write processes should be performed as quickly as possible.

Nevertheless, there could be more effective algorithms which cover the original scenarios that the protocols were designed to support. It is important to propose new algorithms in order to reach new breakthroughs, but often it is difficult to test them under different scenarios. This framework should provide the mechanisms to easily do that, namely implement new protocols, as well as test them in an easily deployable environment.

Apart from the issue concerning shared CPUs and memory discussed in section 3.4.4, another component lessening the “realism” of these tests, was the fact that the network connection between the client and servers was the same speed as between the servers themselves. An early idea was to make use of a program which simulates real-world network connections, causing connection drops, limiting bandwidth, and delaying packages [41], but this was not implemented in the test scenarios after all.

During the course of implementing the framework, further ideas came up to integrate into it. For example, the use of time stamps, or even the “speculative” flag, which was inspired by one particular paper[24]. These were relatively simple to integrate, but many dependent components were affected too, which was very often difficult to locate, especially since these relations were not depicted in the simplified diagrams in section 3.1. One big overhaul of the program was when the fault-tolerant protocol and the non-fault-tolerant protocol were separated for the purpose of increasing performance of the latter. The possibility to separate these was not part of the initial design, but would have been beneficial in the overhaul.

Bibliography

References

[1] Michael Abd-El-Malek et al. “Fault-scalable Byzantine Fault-tolerant Services”. In: *SIGOPS Oper. Syst. Rev.* 39.5 (Oct. 2005), pp. 59–74. ISSN: 0163-5980. DOI: 10.1145/1095809.1095817. URL: http://doi.acm.org/10.1145/1095809.1095817.

[3] Erman Ayday, Farshid Delgosha, and Faramarz Fekri. “Data authenticity and availability in multihop wireless sensor networks”. In: *ACM Trans. Sen. Netw.* 8.2 (Mar. 2012), 10:1–10:26. ISSN: 1550-4859. DOI: 10.1145/2140522.2140523. URL: http://doi.acm.org/10.1145/2140522.2140523.

[6] Yuval Cassuto. “What can coding theory do for storage systems?” In: *SIGACT News* 44.1 (Mar. 2013), pp. 80–88. ISSN: 0163-5700. DOI: 10.1145/2447712.2447734. URL: http://doi.acm.org/10.1145/2447712.2447734.

[7] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296824.

[8] Nikos Chondros, Konstantinos Kokordelis, and Mema Roussopoulos. “On the Practicality of Practical Byzantine Fault Tolerance”. In: *Proceedings of the 13th International Middleware Conference*. Middleware ’12. ontreal, Quebec, Canada: Springer-Verlag New York, Inc., 2012, pp. 436–455. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442654.

[9] Allen Clement et al. “Making Byzantine fault tolerant systems tolerate Byzantine faults”. In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. NSDI’09. Boston, Massachusetts: USENIX Association, 2009, pp. 153–168. URL: http://dl.acm.org/citation.cfm?id=1558977.1558988.

[10] James Cowling et al. “HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI ’06. Seattle, Washington: USENIX Association, 2006, pp. 177–190. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298473.

[11] T.K. Dikaliotis, A.G. Dimakis, and T. Ho. “Security in distributed storage systems by communicating a logarithmic number of bits”. In: *Information Theory Proceedings (ISIT), 2010 IEEE International Symposium on*. 2010, pp. 1948 –1952. DOI: 10.1109/ISIT.2010.5513354.

[12] A.G. Dimakis et al. “A Survey on Network Codes for Distributed Storage”. In: *Proceedings of the IEEE* 99.3 (2011), pp. 476 –489. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2096170.

[13] A.G. Dimakis et al. “Network Coding for Distributed Storage Systems”. In: *Information Theory, IEEE Transactions on* 56.9 (2010), pp. 4539 –4551. ISSN: 0018-9448. DOI: 10.1109/TIT.2010.2054295.

[14] Alexandros G. Dimakis, Vinod Prabhakaran, and Kannan Ramchandran. "Decentralized erasure codes for distributed networked storage". In: *IEEE/ACM Trans. Netw.* 14.SI (June 2006), pp. 2809–2816. ISSN: 1063-6692. DOI: 10.1109/TIT.2006.874535. URL: http://dx.doi.org/10.1109/TIT.2006.874535.

[15] Tobias Distler and Rüdiger Kapitza. "Increasing Performance in Byzantine Fault-tolerant Systems with On-demand Replica Consistency". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 91–106. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966455. URL: http://doi.acm.org/10.1145/1966445.1966455.

[16] Dan Dobre et al. "Proofs of Writing for Efficient and Robust Storage". In: *CoRR* abs/1212.3555 (2012).

[17] K. Driscoll et al. "Byzantine Fault Tolerance, from Theory to Reality." In: *SAFECOMP*. Ed. by Stuart Anderson, Massimo Felici, and Bev Littlewood. Vol. 2788. Lecture Notes in Computer Science. Springer, Mar. 22, 2004, pp. 235–248. ISBN: 3-540-20126-2. URL: http://dblp.uni-trier.de/db/conf/safecomp/safecomp2003.html#DriscollHSZ03.

[18] Y.S. Han, Rong Zheng, and Wai Ho Mow. "Exact regenerating codes for Byzantine fault tolerance in distributed storage". In: *INFOCOM, 2012 Proceedings IEEE*. 2012, pp. 2498 –2506. DOI: 10.1109/INFCOM.2012.6195641.

[22] Rüdiger Kapitza et al. "CheapBFT: resource-efficient byzantine fault tolerance". In: *Proceedings of the 7th ACM european conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 295–308. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168866. URL: http://doi.acm.org/10.1145/2168836.2168866.

[23] Donald E. Knuth. "Dynamic Huffman coding". In: *J. Algorithms* 6.2 (June 1985), pp. 163–180. ISSN: 0196-6774. DOI: 10.1016/0196-6774(85)90036-7. URL: http://dx.doi.org/10.1016/0196-6774(85)90036-7.

[24] Ramakrishna Kotla et al. "Zyzzyva: speculative byzantine fault tolerance". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 45–58. ISSN: 0163-5980. DOI: 10.1145/1323293.1294267. URL: http://doi.acm.org/10.1145/1323293.1294267.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[26] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122 (Proposed Standard). Internet Engineering Task Force, July 2005. URL: http://www.ietf.org/rfc/rfc4122.txt.

[27] Guanfeng Liang, B. Sommer, and N. Vaidya. "Experimental performance comparison of Byzantine Fault-Tolerant protocols for data centers". In: *INFOCOM, 2012 Proceedings IEEE*. 2012, pp. 1422 –1430. DOI: 10.1109/INFCOM.2012.6195507.

[28] Guanfeng Liang and Nitin H. Vaidya. "Byzantine broadcast in point-to-point networks using local linear coding". In: *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. PODC '12. Madeira, Portugal: ACM, 2012, pp. 319–328. ISBN: 978-1-4503-1450-3. DOI: 10.1145/2332432.2332492. URL: http://doi.acm.org/10.1145/2332432.2332492.

[30] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 1558603484.

[31] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (Apr. 1980), pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: http://doi.acm.org/10.1145/322186.322188.

[32] Xiang Pei, Yongjian Wang, and Zhongzhi Luan. "Nova: A Robustness-oriented Byzantine Fault Tolerance Protocol". In: *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. 2010, pp. 151 –156. DOI: 10.1109/GCC.2010.40.

[33] R. Rodrigues et al. "Automatic Reconfiguration for Large-Scale Reliable Storage Systems". In: *Dependable and Secure Computing, IEEE Transactions on* 9.2 (2012), pp. 145 –158. ISSN: 1545-5971. DOI: 10.1109/TDSC.2010.52.

[34] Richard D. Schlichting and Fred B. Schneider. "Fail-stop processors: an approach to designing fault-tolerant computing systems". In: *ACM Trans. Comput. Syst.* 1.3 (Aug. 1983), pp. 222–238. ISSN: 0734-2071. DOI: 10.1145/357369.357371. URL: http://doi.acm.org/10.1145/357369.357371.

[42] Hao Wang and Bill Lin. "Designing efficient codes for synchronization error channels". In: *Proceedings of the Nineteenth International Workshop on Quality of Service*. IWQoS '11. San Jose, California: IEEE Press, 2011, 43:1–43:9. URL: http://dl.acm.org/citation.cfm?id=1996039.1996090.

Online References

[2] *Apache Log4j 2*. Dec. 2013. URL: http://logging.apache.org/log4j/2.x/.

[4] *BFT-SMaRT; High-performance Byzantine Fault-Tolerant State Machine Replication*. Jan. 2013. URL: http://code.google.com/p/bft-smart/.

[5] *BFT-SMaRT open issues list*. Jan. 2013. URL: http://code.google.com/p/bft-smart/issues/list.

[19] *How to use the Kernel Samepage Merging feature*. Dec. 2013. URL: https://www.kernel.org/doc/Documentation/vm/ksm.txt.

[20] *Java: Loading, Linking, and Initializing*. Dec. 2013. URL: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html.

[21] *Java lorem ipsum generator*. Dec. 2013. URL: https://github.com/oliverdodd/jlorem.

[29] Bob Lord. *Keeping our users secure*. 2013. URL: http://blog.twitter.com/2013/02/keeping-our-users-secure.html.

[35] Amazon AWT Team. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. 2011. URL: http://aws.amazon.com/message/65648/.

[36] Amazon S3 Team. *Amazon S3 Availability Event: July 20, 2008*. 2008. URL: http://status.aws.amazon.com/s3-20080720.html.

[37] Ben Treynor. *Gmail back soon for everyone*. 2011. URL: http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html.

[38] *Understanding JIT Compilation and Optimizations*. Dec. 2013. URL: http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html.

[39] *UpRight; infrastructure and library for building fault tolerant distributed systems*. Jan. 2013. URL: http://code.google.com/p/upright/.

[40] *UpRight open issues list*. Jan. 2013. URL: http://code.google.com/p/upright/wiki/UpRightIssue.

[41] *WANem, Wide Area Network Emulator*. June 2013. URL: http://wanem.sourceforge.net/.

A Appendix

A.1 Source Code

```
1 package edu.illinois.ece.ftstorage.client;
2
3 import java.io.BufferedReader;
4 import java.io.File;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.util.List;
8 import java.util.UUID;
9
10 import org.apache.commons.cli.CommandLine;
11 import org.apache.commons.cli.CommandLineParser;
12 import org.apache.commons.cli.HelpFormatter;
13 import org.apache.commons.cli.Options;
14 import org.apache.commons.cli.ParseException;
15 import org.apache.commons.cli.PosixParser;
16 import org.apache.commons.configuration.ConfigurationException;
17 import org.apache.logging.log4j.LogManager;
18 import org.apache.logging.log4j.Logger;
19
20 import edu.illinois.ece.ftstorage.server.StorageServerConfigurationParser;
21 import edu.illinois.ece.ftstorage.shared.AbstractStorageEntity;
22 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
23 import edu.illinois.ece.ftstorage.shared.exceptions.QuitException;
24 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
25 import
   ↪ edu.illinois.ece.ftstorage.shared.storageCluster.StorageClusterConfigurationParser;
26 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
27 import edu.illinois.ece.ftstorage.shared.storagePackage.HashingSchemeType;
28 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
29 import edu.illinois.ece.ftstorage.shared.util.CreateData;
30
31 public class StorageClient extends AbstractStorageEntity {
32
33   /**
34    *
35    */
36   private static final long serialVersionUID = 2016578721963295246L;
37   protected final static Logger logger = LogManager.getLogger(Thread
38       .currentThread().getStackTrace()[1].getClassName());
39   final static Options options = new Options();
40
41   private HashingSchemeType hashingSchemeType = HashingSchemeType.NONE;
42   private CodingSchemeType codingSchemeType = CodingSchemeType.NONE;
43
44   List<StorageCluster> clusters;
45
46   StorageClient() throws IOException, ConfigurationException {
47     this(UUID.randomUUID().toString(), "StorageClient",
48         "StorageClient description");
49     this.clusters = StorageClusterConfigurationParser.getInstance()
50         .getClusters();
```

```
51      this.storedPackages = StorageClusterConfigurationParser.getInstance()
52          .getPackages();
53    }
54
55    StorageClient(String uid, String name, String description) {
56      super(uid, name, description);
57    }
58
59    @Override
60    public void read(AbstractStorageEntity requestor,
61        StoragePackage storagePackage) {
62      for (StorageCluster storageCluster : this.clusters) {
63        Thread t;
64        try {
65          StorageClientPackageHandler storageClientPackageHandler = new
          ↪ StorageClientPackageHandler(
66              this, storageCluster, storagePackage.clone());
67          this.addRunningPackageHandler(storageClientPackageHandler,
68              storagePackage.getUid());
69          t = new Thread(storageClientPackageHandler);
70          logger.debug("Starting new {} for {} in {}",
71              storageClientPackageHandler.getClass().getName(),
72              storagePackage, t.getName());
73          t.start();
74        } catch (IOException e) {
75          logger.fatal("Caught {}. Error was: {}", e.getClass(),
76              e.getMessage());
77        }
78      }
79    }
80
81    @Override
82    public void write(AbstractStorageEntity requestor,
83        StoragePackage storagePackage) {
84      for (StorageCluster storageCluster : this.clusters) {
85        Thread t;
86        try {
87          StorageClientPackageHandler storageClientPackageHandler = new
          ↪ StorageClientPackageHandler(
88              this, storageCluster, storagePackage.clone());
89          this.addRunningPackageHandler(storageClientPackageHandler,
90              storagePackage.getUid());
91          t = new Thread(storageClientPackageHandler);
92          logger.debug("Starting new {} for {} in {}",
93              storageClientPackageHandler.getClass().getName(),
94              storagePackage, t.getName());
95          t.start();
96        } catch (IOException e) {
97          logger.fatal("Caught {}. Error was: {}", e.getClass(),
98              e.getMessage());
99        }
100     }
101   }
102
103   @Override
104   public void run() {
105   }
106
107   private void clearScreen() {
108     for (int i = 0; i < 25; i++) {
109       System.out.println();
110     }
```

```
111     // The following is dependent on the terminal supporting this escape
112     // command. We use the previous command, just in case the terminal does
113     // not support this escape key.
114     System.out.print("\033[2J");
115   }
116
117   private void displayMenu() throws QuitException {
118     String input = "";
119     InputStreamReader isr = new InputStreamReader(System.in);
120     BufferedReader br = new BufferedReader(isr);
121     // clear the screen a few times because we want the menu at the bottom
122     // of the screen, just in case the prompt starts at the top of the
123     // screen.
124     for (int i = 0; i < 5; i++) {
125       this.clearScreen();
126     }
127     do {
128       try {
129         System.out.println("StorageClient");
130         System.out.println("Please enter a command: ");
131         System.out.println("\t change (h)ashing scheme (currently "
132             + this.hashingSchemeType + ")");
133         System.out.println("\t (w)rite a StoragePackage");
134         System.out.println("\t (r)ead a StoragePackage");
135         System.out.println("\t (q)uit");
136         System.out.print("Input: ");
137         input = br.readLine();
138         if (input.toLowerCase().trim().equals("q")) {
139           throw new QuitException();
140         } else if (input.toLowerCase().trim().equals("h")) {
141           this.clearScreen();
142           HashingSchemeType[] hashingSchemes = HashingSchemeType
143               .values();
144           do {
145             int counter = 0;
146             int hashingSchemeIndex = 0;
147             try {
148               System.out.println("Current hashing scheme is: "
149                   + this.hashingSchemeType);
150               for (HashingSchemeType hashingScheme : hashingSchemes) {
151                 System.out.println("(" + counter + "):"
152                     + hashingScheme);
153                 counter++;
154               }
155               System.out
156                   .print("Please enter a hashing scheme type or go (b)ack:
                  ↪ ");
157               input = br.readLine();
158               this.clearScreen();
159               if (!input.isEmpty()) {
160                 hashingSchemeIndex = Integer.parseInt(input);
161                 // throws NumberFormatException
162               }
163               if (hashingSchemeIndex < 0
164                   || hashingSchemeIndex >= hashingSchemes.length) {
165                 System.out
166                     .println("Please enter a number between 0 and "
167                         + (hashingSchemes.length - 1));
168               } else {
169                 this.hashingSchemeType =
                 ↪ hashingSchemes[hashingSchemeIndex];
170               }
```

```
171            } catch (NumberFormatException e) {
172              System.out
173                  .println("Please enter an integer value.");
174            } catch (Exception e) {
175              logger.error(
176                  "Caught unchecked exception {} with message {}.",
177                  e.getClass().getName(), e.getMessage());
178            }
179          } while (!input.toLowerCase().trim().equals("b"));
180          this.clearScreen();
181        } else if (input.toLowerCase().trim().equals("w")) {
182          this.clearScreen();
183          do {
184            try {
185              StoragePackage storagePackage = null;
186              System.out
187                  .println("Write (t)ext to all remote systems.");
188              System.out
189                  .println("Write (l)orem ipsum text to all remote
                   ↪ systems.");
190              System.out
191                  .println("Write the (c)luster configuration to all
                   ↪ remote systems.");
192              System.out
193                  .println("Write the (s)server configuration to all
                   ↪ remote systems.");
194              System.out
195                  .println("Write the (a)pplication to all remote
                   ↪ systems.");
196              System.out
197                  .print("Please enter a StoragePackage type to write or
                   ↪ go (b)ack: ");
198              input = br.readLine();
199              if (input.toLowerCase().trim().equals("b")) {
200                this.clearScreen();
201                break;
202              }
203              if (input.toLowerCase().trim().equals("l")) {
204                input = " ";
205                System.out
206                    .println("Please enter the number of kilobytes to be
                     ↪ written remotely:");
207                int kBytesOfDataToGenerate = 0;
208                do {
209                  input = br.readLine();
210                  if (!input.isEmpty()) {
211                    try {
212                      kBytesOfDataToGenerate = Integer
213                          .parseInt(input);
214                    } catch (NumberFormatException e) {
215                      System.out
216                          .println("Please enter an integer value.");
217                    }
218                  }
219                } while (kBytesOfDataToGenerate == 0);
220                String string = CreateData
221                    .createString(kBytesOfDataToGenerate);
222                storagePackage = new StoragePackage(string,
223                    this.codingSchemeType,
224                    this.hashingSchemeType);
225              } else if (input.toLowerCase().trim().equals("t")) {
226                StringBuilder sb = new StringBuilder();
```

```
227                 input = " ";
228                 System.out
229                     .println("Please enter the text to be stored remotely.
                        ↪ Hit <ENTER> or <RETURN> twice to finish writing:");
230                 do {
231                   input = br.readLine();
232                   sb.append(input);
233                   if (!input.isEmpty()) {
234                     sb.append(System.lineSeparator());
235                   }
236                 } while (!input.isEmpty());
237                 storagePackage = new StoragePackage(
238                     sb.toString(), this.codingSchemeType,
239                     this.hashingSchemeType);
240               } else if (input.toLowerCase().trim().equals("c")) {
241                 storagePackage = new StoragePackage(
242                     StorageClusterConfigurationParser
243                         .getInstance().getConfig(),
244                     this.codingSchemeType,
245                     this.hashingSchemeType);
246               } else if (input.toLowerCase().trim().equals("s")) {
247                 storagePackage = new StoragePackage(
248                     StorageServerConfigurationParser
249                         .getInstance().getConfig(),
250                     this.codingSchemeType,
251                     this.hashingSchemeType);
252               } else if (input.toLowerCase().trim().equals("a")) {
253                 File currentJar = new File(StorageClient.class
254                     .getProtectionDomain()
255                     .getCodeSource()
256                     .getLocation()
257                     .getFile()
258                     .replaceAll("FTStorage-Client.jar",
259                         "FTStorage-Server.jar"));
260                 storagePackage = new StoragePackage(currentJar,
261                     this.codingSchemeType,
262                     this.hashingSchemeType);
263               } else {
264                 this.printInvalidInput(input);
265               }
266               if (storagePackage != null) {
267                 System.out.println("Writing package with UUID "
268                     + storagePackage.getUid());
269                 this.write(this, storagePackage);
270               }
271             } catch (IllegalArgumentException
272                 | ConfigurationException e) {
273               logger.error("Caught {} with message {}.",
274                   e.getClass(), e.getMessage());
275             } catch (Exception e) {
276               logger.error(
277                   "Caught unchecked exception {} with message {}.",
278                   e.getClass().getName(), e.getMessage());
279             }
280           } while (!input.toLowerCase().trim().equals("b"));
281         } else if (input.toLowerCase().trim().equals("r")) {
282           this.clearScreen();
283           do {
284             try {
285               StoragePackage storagePackage = null;
286               System.out
```

```
287                     .print("Please enter the UUID of the StoragePackage to
                        ↪ read or go (b)ack: ");
288                 input = br.readLine();
289                 if (input.toLowerCase().trim().equals("b")) {
290                   this.clearScreen();
291                   break;
292                 }
293                 UUID storedUuid = null;
294                 try {
295                   storedUuid = UUID.fromString(input);
296                 } catch (IllegalArgumentException e) {
297                   System.out
298                       .println("\""
299                           + input
300                           + "\" is not a valid UUID. Please enter a valid
                          ↪ character sequence.");
301                 }
302                 if (storedUuid != null) {
303                   StoragePackage storedPackage = new StoragePackage(
304                       storedUuid, this.codingSchemeType,
305                       this.hashingSchemeType);
306                   if (this.isPackagedStored(storedPackage)) {
307                     System.out
308                         .println("Reading StoragePackage with UUID "
309                             + input
310                             + " from remote systems...");
311                     storedPackage = this
312                         .getStoredPackage(storedPackage);
313                     storagePackage = new StoragePackage(
314                         storedUuid, this.codingSchemeType,
315                         this.hashingSchemeType);
316                     // use storedPackage.getCodingScheme() and
317                     // storedPackage.getHashingScheme() because
318                     // the StoragePackage might not have been
319                     // stored with the
320                     // codingScheme/hashingScheme information
321                     synchronized (this.storedPackages) {
322                       this.removeStoragePackage(storagePackage);
323                       for (StorageCluster storageCluster : this.clusters) {
324                         storageCluster
325                             .removeStoragePackage(storagePackage);
326                         if (!storageCluster.isFTCluster()) {
327                           // if we have an FT cluster,
328                           // then the requestor is the
329                           // client, whereas for a non-FT
330                           // cluster, the requestor is the
331                           // cluster itself and the
332                           // requestee a StorageNode
333                           for (StorageNode storageNode : storageCluster
334                               .getNodeList()) {
335                             storageNode
336                                 .removeStoragePackage(storagePackage);
337                           }
338                         }
339                       }
340                     }
341                     this.read(this, storagePackage);
342                     while (!this
343                         .isPackagedStored(storagePackage)) {
344                       // wait for the response to come back
345                     }
346                     System.out
```

```
347                         .println("Received package from remote systems.
                            ↪ Value of StoragePackage is:");
348                     System.out.println(new String(this
349                         .getStoredPackage(storagePackage)
350                         .getData()).substring(0, 100));
351                     System.out
352                         .print("Please hit enter to go back to the previous
                            ↪ menu.");
353                     input = br.readLine();
354                     break;
355                   } else {
356                     System.out
357                         .println("The StoragePackage with UUID "
358                             + input
359                             + " is not stored. Attemtping to read with
                                ↪ default values of hashing and coding scheme
                                ↪ type.");
360                     System.out
361                         .println("Reading StoragePackage with UUID "
362                             + input
363                             + " from remote systems...");
364                     storagePackage = new StoragePackage(
365                         storedUuid, CodingSchemeType.NONE,
366                         HashingSchemeType.NONE);
367                     this.read(this, storagePackage);
368                     while (!this
369                         .isPackagedStored(storagePackage)) {
370                       // wait for the response to come back
371                     }
372                     System.out
373                         .println("Received package from remote systems.
                            ↪ Value of StoragePackage is:");
374                     System.out.println(new String(
375                         storagePackage.getData()));
376                     System.out
377                         .print("Please hit enter to go back to the previous
                            ↪ menu.");
378                     input = br.readLine();
379                     break;
380                   }
381                 }
382               } catch (IllegalArgumentException e) {
383                 logger.error("Caught {} with message {}.",
384                     e.getClass(), e.getMessage());
385               } catch (Exception e) {
386                 logger.error(
387                     "Caught unchecked exception {} with message {}.",
388                     e.getClass().getName(), e.getMessage());
389               }
390             } while (!input.toLowerCase().trim().equals("b"));
391           } else {
392             this.printInvalidInput(input);
393           }
394         } catch (IOException e) {
395           logger.error("IOException while reading input.");
396         }
397       } while (!input.toLowerCase().trim().equals("q"));
398     }
399
400     private void printInvalidInput(String input) {
401       this.clearScreen();
402       System.out
```

```
403         .println("Please enter a valid option. \""
404             + input
405             + "\" is not valid input. Please choose a letter from the
                ↪ options in the parentheses.");
406   }
407
408   /**
409    * @param args
410    */
411   public static void main(String[] args) {
412     try {
413       StorageClient client = new StorageClient();
414       CommandLineParser parser = new PosixParser();
415       HelpFormatter formatter = new HelpFormatter();
416       CommandLine line = null;
417       line = parser.parse(options, args);
418       if (line.hasOption("help")) {
419         formatter.printHelp("StorageClient", options);
420       } else {
421         client.displayMenu();
422       }
423     } catch (ConfigurationException | IOException | ParseException e) {
424       logger.error("Caught {} with message {}.", e.getClass(),
425           e.getMessage());
426     } catch (QuitException e) {
427       System.out.println("Exiting StorageClient...");
428       System.exit(0);
429     }
430   }
431 }
```

Listing A.1: client.StorageClient

```
 1 package edu.illinois.ece.ftstorage.client;
 2
 3 import java.io.IOException;
 4 import java.net.InetSocketAddress;
 5
 6 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
 7 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
 8 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
 9 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
10
11 public class StorageClientPackageHandler extends AbstractPackageHandler {
12
13   final StorageClient requestor;
14   final StorageCluster requestee;
15
16   public StorageClientPackageHandler(StorageClient requestor,
17       StorageCluster requestee, StoragePackage storagePackage)
18       throws IOException {
19     super(requestor, requestee, storagePackage);
20     this.requestor = requestor;
21     this.requestee = requestee;
22   }
23
24   @Override
25   protected void read() throws ProtocolException {
26     logger.trace("{} reading from {}", this.requestor.getUid(),
27         this.requestee.getUid());
28     this.requestee.startStorageEntity();
29     InetSocketAddress isa = (InetSocketAddress) this.serverSocket
```

```
30         .getLocalSocketAddress();
31     this.storagePackage.setSourceAckListeningIP(isa.getAddress());
32     this.storagePackage.setSourceAckListeningPort(this.requestee
33         .getSocketPort());
34     this.requestee.read(this.requestor, this.storagePackage);
35   }
36 
37   @Override
38   protected void receiveReadAck() {
39   }
40 
41   @Override
42   protected void receiveWriteAck() {
43   }
44 
45   @Override
46   protected void write() throws ProtocolException {
47     logger.trace("{} writing to {}", this.requestor.getUid(),
48         this.requestee.getUid());
49     logger.info("{} writing to {}", this.requestor.getUid(),
50         this.requestee.getUid());
51     this.requestee.startStorageEntity();
52     InetSocketAddress isa = (InetSocketAddress) this.serverSocket
53         .getLocalSocketAddress();
54     this.storagePackage.setSourceAckListeningIP(isa.getAddress());
55     this.storagePackage.setSourceAckListeningPort(this.requestee
56         .getSocketPort());
57     this.requestee.write(this.requestor, this.storagePackage);
58   }
59 }
```

Listing A.2: client.StorageClientPackageHandler

```
1 package edu.illinois.ece.ftstorage.server;
2 
3 import java.io.ByteArrayInputStream;
4 import java.io.EOFException;
5 import java.io.File;
6 import java.io.FileOutputStream;
7 import java.io.IOException;
8 import java.io.ObjectInput;
9 import java.io.ObjectInputStream;
10 import java.io.OutputStream;
11 import java.net.BindException;
12 import java.net.ServerSocket;
13 import java.net.Socket;
14 import java.net.URISyntaxException;
15 import java.util.ArrayList;
16 import java.util.List;
17 import java.util.UUID;
18 
19 import javax.xml.transform.TransformerException;
20 
21 import org.apache.commons.configuration.ConfigurationException;
22 import org.apache.logging.log4j.LogManager;
23 import org.apache.logging.log4j.Logger;
24 import org.w3c.dom.Document;
25 
26 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
27 import edu.illinois.ece.ftstorage.shared.AbstractStorageEntity;
28 import
   ↪ edu.illinois.ece.ftstorage.shared.exceptions.StorageClusterConfigurationException;
```

```
29 import
   ↪ edu.illinois.ece.ftstorage.shared.ftProtocol.AbstractFTProtocolHandler;
30 import
   ↪ edu.illinois.ece.ftstorage.shared.ftProtocol.FTProtocolHandlerFactory;
31 import
   ↪ edu.illinois.ece.ftstorage.shared.storageCluster.StorageClusterConfigurationParser;
32 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
33 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
34 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
35 import edu.illinois.ece.ftstorage.shared.util.FtStorageUtil;
36
37 public class StorageServer extends AbstractStorageEntity {
38
39   /**
40    *
41    */
42   private static final long serialVersionUID = -5856889596622157349L;
43
44   protected final static Logger logger = LogManager.getLogger(Thread
45       .currentThread().getStackTrace()[1].getClassName());
46
47   /**
48    * @param args
49    */
50   public static void main(String[] args) {
51     List<StorageServer> ssl = null;
52     try {
53       ssl = StorageServerConfigurationParser.getInstance().getServers();
54     } catch (ConfigurationException | IOException e) {
55       logger.fatal(
56           "Could not parse server configuration. Caught {}. Error was: {}",
57           e.getClass(), e.getMessage());
58       System.exit(0);
59     }
60     for (StorageServer storageServer : ssl) {
61       if (args.length == 0) {
62         startServer(storageServer);
63       } else {
64         for (String arg : args) {
65           if (storageServer.getUid().equals(arg)) {
66             startServer(storageServer);
67           }
68         }
69       }
70     }
71   }
72
73   private static void startServer(StorageServer storageServer) {
74     logger.trace("Starting StorageServer listening on {}:{}", storageServer
75         .getStorageNode().getStorageServerAddress(), storageServer
76         .getStorageNode().getStorageServerPort());
77     Thread t;
78     t = new Thread(storageServer);
79     t.start();
80   }
81
82   private static void restartServer(StorageServer storageServer) {
83     // TODO
84     final String javaBin = System.getProperty("java.home") + File.separator
85         + "bin" + File.separator + "java";
86     File currentJar;
87     try {
```

```
88      currentJar = new File(storageServer.getClass()
89          .getProtectionDomain().getCodeSource().getLocation()
90          .toURI());
91
92      /* is it a jar file? */
93      if (!currentJar.getName().endsWith(".jar")) {
94        return;
95      }
96
97      /* Build command: java -jar application.jar */
98      final ArrayList<String> command = new ArrayList<String>();
99      command.add(javaBin);
100     command.add("-jar");
101     command.add(currentJar.getPath());
102     command.add(storageServer.getUid());
103     final ProcessBuilder builder = new ProcessBuilder(command);
104     storageServer.closePackageHandlerSockets();
105     if (storageServer.serverSocket.isBound()) {
106       storageServer.serverSocket.close();
107     }
108     logger.trace("Restarting with command {}.", builder.command());
109     logger.info("Attempting to restart server with command {}.",
110         builder.command());
111     builder.start();
112     System.exit(0);
113   } catch (URISyntaxException | IOException e) {
114     logger.error(
115         "Error while trying to restart {}. Message was \"{}\".",
116         storageServer.getClass(), e.getMessage());
117   }
118 }
119
120 ServerSocket serverSocket;
121
122 StorageNode storageNode;
123
124 public StorageServer(String uid, String name, String description,
125     StorageNode storageNode) throws IOException, ConfigurationException {
126   super(uid, name, description);
127   this.storageNode = storageNode;
128   try {
129     this.serverSocket = new ServerSocket(
130         this.storageNode.getStorageServerPort(), 50,
131         this.storageNode.getSocketAddress());
132   } catch (BindException e) {
133     logger.error("Could not bind on port {}.",
134         this.storageNode.getStorageServerPort());
135     throw e;
136   }
137   StorageClusterConfigurationParser sccp =
    ↪ StorageClusterConfigurationParser
138       .getInstance(uid);
139   sccp.setConfigFilePath(FtStorageUtil.getPathWithNewParentFolderName(
140       sccp.getConfig().getFile(), this.getUid()));
141   this.storedPackages = StorageClusterConfigurationParser.getInstance(
142       this.getUid()).getPackages();
143 }
144
145 public StorageNode getStorageNode() {
146   return this.storageNode;
147 }
148
```

```
149   private Object listen() throws IOException, ClassNotFoundException {
150     Socket socket = this.serverSocket.accept();
151     ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
152     Object receivedObject = in.readObject();
153     logger.trace("Received Object at {} from {}.", this.getUid(),
154         socket.getRemoteSocketAddress());
155     in.close();
156     socket.close();
157     return receivedObject;
158   }
159
160   @Override
161   public void read(AbstractStorageEntity requestor,
162       StoragePackage storagePackage) {
163     logger.trace("Reading {} from {} with FTProtocol {}.", storagePackage,
164         this.getStorageNode().getStorageCluster().getUid(), this
165             .getStorageNode().getStorageCluster().getFtProtocol());
166     synchronized (this.runningPackageHandlers) {
167       if (this.isPackagedStored(storagePackage)) {
168         logger.trace("{} is stored on {}. isSpeculative? {}",
169             storagePackage, this.getUid(),
170             storagePackage.isSpeculative());
171       }
172       if (this.runningPackageHandlers
173           .containsKey(storagePackage.getUid())) {
174         logger.trace(
175             "{} package handler(s) already running on {} for {} .",
176             this.runningPackageHandlers
177                 .get(storagePackage.getUid()).size(), this
178                 .getUid(), storagePackage.getUid());
179         for (AbstractPackageHandler packageHandler :
            ↪ this.runningPackageHandlers
180             .get(storagePackage.getUid())) {
181           packageHandler.addPendingPackage(storagePackage);
182           // attempt to restart package handler if it has stopped
183           // packageHandler.startPackageHandler();
184         }
185       } else {
186         Thread t;
187         if (this.storageNode.getStorageCluster().isFTCluster()) {
188           try {
189             AbstractFTProtocolHandler ftProtocolHandler =
                ↪ FTProtocolHandlerFactory
190                 .get(this, this.getStorageNode()
191                     .getStorageCluster(), storagePackage,
192                     this.storageNode.getStorageCluster()
193                         .getFtProtocol());
194             this.addRunningPackageHandler(ftProtocolHandler,
195                 storagePackage.getUid());
196             t = new Thread(ftProtocolHandler);
197             logger.debug("Starting new {} for {} on {} in {}",
198                 ftProtocolHandler.getClass().getName(),
199                 storagePackage, requestor.getUid(), t.getName());
200             t.start();
201           } catch (IOException e) {
202             logger.error("Caught {}. Error was: {}", e.getClass(),
203                 e.getMessage());
204           }
205         } else {
206           try {
207             StorageServerPackageHandler storageServerPackageHandler = new
                ↪ StorageServerPackageHandler(
```

```
208                 this, this.getStorageNode(), storagePackage);
209             this.addRunningPackageHandler(
210                 storageServerPackageHandler,
211                 storagePackage.getUid());
212             t = new Thread(storageServerPackageHandler);
213             logger.debug("Starting new {} for {} on {} in {}",
214                 storageServerPackageHandler.getClass()
215                     .getName(), storagePackage, requestor
216                     .getUid(), t.getName());
217             t.start();
218           } catch (IOException e) {
219             logger.error("Caught {}. Error was: {}", e.getClass(),
220                 e.getMessage());
221           }
222         }
223       }
224     }
225   }
226 
227   @Override
228   public void run() {
229     boolean run = false;
230     synchronized (this.syncObject) {
231       if (!this.isRunning()) {
232         this.isRunning = true;
233         run = true;
234         logger.trace("Starting {} {}.", this.getClass().getName(),
235             this.getUid());
236       } else {
237         run = false;
238         logger.trace("{} {} already running.", this.getClass()
239             .getName(), this.getUid());
240       }
241     }
242     if (run) {
243       while (true) {
244         try {
245           if (logger.isDebugEnabled()) {
246             int numPendingPackages = 0;
247             synchronized (this.runningPackageHandlers) {
248               for (UUID uuid : this.runningPackageHandlers
249                   .keySet()) {
250                 for (AbstractPackageHandler aph :
                    ↪ this.runningPackageHandlers
251                     .get(uuid)) {
252                   numPendingPackages += aph
253                       .getNumPendingWrittenPackages();
254                 }
255               }
256             }
257             logger.trace(
258                 "isPrimary? {} StorageServer {}: Number of stored/pending
                    ↪ StoragePackages: {}/{}",
259                 this.storageNode.equals(this.getStorageNode()
260                     .getStorageCluster().getPrimaryNode()),
261                 this.getUid(), this.storedPackages.size(),
262                 numPendingPackages);
263           }
264           Object receivedObject = this.listen();
265           if (receivedObject instanceof StoragePackage) {
266             StoragePackage storagePackage = (StoragePackage)
                ↪ receivedObject;
```

```
267             storagePackage.setSpeculative(true);
268             logger.debug("Received {} on {}.", storagePackage,
269                 this.getUid());
270             if (storagePackage.getPackageType() !=
                ↪ StoragePackageType.READ) {
271               this.write(this, storagePackage);
272             } else {
273               this.read(this, storagePackage);
274             }
275           } else {
276             logger.warn(
277                 "Handling of class {} has not been implemented yet.",
278                 receivedObject.getClass());
279           }
280         } catch (ClassNotFoundException | IOException e) {
281           logger.error("Caught {}. Error was: {}", e.getClass(),
282               e.getMessage());
283         }
284       }
285     }
286   }
287
288   public void reconfigure(StoragePackage storagePackage) {
289     // TODO test
290     if (this.isPackagedStored(storagePackage)) {
291       try {
292         // all non-faulty servers reached consensus and the
293         // package has not been tampered with, so we are now
294         // ready to reconfigure the server
295         // also assume that the package handler has verified the
296         // package and added it to the stored package queue
297         // before we reach this point
298         if (storagePackage.getPackageType().equals(
299             StoragePackageType.CONFIGURATION)) {
300           logger.info("Reconfiguring {}.", this.getUid());
301           ByteArrayInputStream bis = null;
302           ObjectInputStream in = null;
303           try {
304             bis = new ByteArrayInputStream(storagePackage.getData());
305             in = new ObjectInputStream(bis);
306             Document newDocument = null;
307             Object storagePackageData = in.readObject();
308             if (storagePackageData instanceof Document) {
309               newDocument = (Document) storagePackageData;
310               if (newDocument.getFirstChild().getAttributes()
311                   .getNamedItem("type") != null) {
312                 if (newDocument.getFirstChild().getAttributes()
313                     .getNamedItem("type").getNodeValue()
314                     .equals("ClusterConfig")) {
315                   logger.info("Received ClusterConfig.");
316                   this.write(this, storagePackage);
317                   StorageClusterConfigurationParser
318                       .getInstance(this.getUid())
319                       .setDocument(newDocument,
320                           this.getUid());
321                 } else if (newDocument.getFirstChild()
322                     .getAttributes().getNamedItem("type")
323                     .getNodeValue().equals("ServerConfig")) {
324                   logger.info("Received ServerConfig.");
325                   this.write(this, storagePackage);
326                   StorageServerConfigurationParser
327                       .getInstance(this.getUid())
```

```
328                     .setDocument(newDocument,
329                         this.getUid());
330               } else {
331                 logger.trace(FtStorageUtil.documentToString(
332                     storagePackage.getData().length,
333                     newDocument));
334                 throw new StorageClusterConfigurationException(
335                     "Attribute \"type\" has an unrecognized value "
336                         + newDocument
337                             .getFirstChild()
338                             .getAttributes()
339                             .getNamedItem(
340                                 "type")
341                         + ".");
342               }
343             } else {
344               logger.trace(FtStorageUtil.documentToString(
345                   storagePackage.getData().length,
346                   newDocument));
347               throw new StorageClusterConfigurationException(
348                   "Attribute 'type' is not set in XML configuration.");
349             }
350           } else {
351             logger.warn(
352                 "Handling of {} of type {} has not been implemented
                 ↪ yet.",
353                 storagePackageData.getClass(),
354                 storagePackage.getPackageType());
355           }
356         } catch (TransformerException | ConfigurationException
357             | IOException e) {
358           logger.error("Caught {}. Error was: {}", e.getClass(),
359               e.getMessage());
360         } finally {
361           FtStorageUtil.closeSilently(bis, in);
362         }
363       } else if (storagePackage.getPackageType().equals(
364           StoragePackageType.APPLICATION)) {
365         logger.info("Reconfiguring {}.", this.getUid());
366         ByteArrayInputStream bais = null;
367         ObjectInput in = null;
368         OutputStream outputStream = null;
369         try {
370           File currentJar = new File(this.getClass()
371               .getProtectionDomain().getCodeSource()
372               .getLocation().toURI());
373           File newJar = new File(
374               FtStorageUtil.getPathWithNewParentFolderName(
375                   currentJar, this.getUid()));
376           outputStream = new FileOutputStream(newJar);
377           outputStream.write(storagePackage.getData());
378           logger.debug("Wrote new {} to {}.",
379               storagePackage.getPackageType(),
380               newJar.toPath());
381           this.write(this, storagePackage);
382           restartServer(this);
383         } catch (EOFException | URISyntaxException e) {
384           logger.error("Caught {}. Error was: {}", e.getClass(),
385               e.getMessage());
386         } finally {
387           if (bais != null) {
388             bais.close();
```

```
389             }
390             if (in != null) {
391               in.close();
392             }
393             if (outputStream != null) {
394               outputStream.close();
395             }
396           }
397         }
398       } catch (ClassNotFoundException | IOException e) {
399         logger.error("Caught {}. Error was: {}", e.getClass(),
400             e.getMessage());
401       }
402     }
403   }
404 
405   public void setStorageNode(StorageNode storageNode) {
406     this.storageNode = storageNode;
407   }
408 
409   @Override
410   public void write(AbstractStorageEntity requestor,
411       StoragePackage storagePackage) {
412     logger.trace("Writing {} to {} with FTProtocol {}.", storagePackage,
413         this.getStorageNode().getStorageCluster().getUid(), this
414             .getStorageNode().getStorageCluster().getFtProtocol());
415     if (this.isPackagedStored(storagePackage)) {
416       logger.trace("{} is already stored on {}. isSpeculative? {}",
417           storagePackage, this.getUid(),
418           this.isPackageSpeculative(storagePackage.getUid()));
419     }
420     synchronized (this.runningPackageHandlers) {
421       if (this.runningPackageHandlers
422           .containsKey(storagePackage.getUid())) {
423         for (AbstractPackageHandler packageHandler :
        ↪ this.runningPackageHandlers
424             .get(storagePackage.getUid())) {
425           logger.debug("Adding {} to {} as pending package.",
426               storagePackage, packageHandler.getClass().getName());
427           packageHandler.addPendingPackage(storagePackage);
428           if (!packageHandler.isRunning()) {
429             Thread t = new Thread(packageHandler);
430             logger.debug("Restarting {} for {} on {} in {}",
431                 packageHandler.getClass().getName(),
432                 storagePackage, requestor.getUid(), t.getName());
433             t.start();
434           }
435         }
436       } else {
437         Thread t;
438         if (this.storageNode.getStorageCluster().isFTCluster()) {
439           try {
440             AbstractFTProtocolHandler ftProtocolHandler =
            ↪ FTProtocolHandlerFactory
441                 .get(this, this.getStorageNode()
442                     .getStorageCluster(), storagePackage,
443                     this.storageNode.getStorageCluster()
444                         .getFtProtocol());
445             this.addRunningPackageHandler(ftProtocolHandler,
446                 storagePackage.getUid());
447             t = new Thread(ftProtocolHandler);
448             logger.debug("Starting new {} for {} on {} in {}",
```

```
449                 ftProtocolHandler.getClass().getName(),
450                 storagePackage, requestor.getUid(), t.getName());
451             t.start();
452           } catch (IOException e) {
453             logger.error("Caught {}. Error was: {}", e.getClass(),
454                 e.getMessage());
455           }
456         } else {
457           try {
458             StorageServerPackageHandler storageServerPackageHandler = new
                ↪ StorageServerPackageHandler(
459                 this, this.getStorageNode(), storagePackage);
460             this.addRunningPackageHandler(
461                 storageServerPackageHandler,
462                 storagePackage.getUid());
463             t = new Thread(storageServerPackageHandler);
464             logger.debug("Starting new {} for {} on {} in {}",
465                 storageServerPackageHandler.getClass()
466                     .getName(), storagePackage, requestor
467                     .getUid(), t.getName());
468             t.start();
469           } catch (IOException e) {
470             logger.error("Caught {}. Error was: {}", e.getClass(),
471                 e.getMessage());
472           }
473         }
474       }
475     }
476   }
477 }
```

Listing A.3: server.StorageServer

```
1 package edu.illinois.ece.ftstorage.server;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import java.util.Iterator;
7 import java.util.LinkedList;
8 import java.util.List;
9 import java.util.Map;
10 import java.util.concurrent.ConcurrentHashMap;
11
12 import javax.xml.parsers.DocumentBuilderFactory;
13 import javax.xml.parsers.ParserConfigurationException;
14
15 import org.apache.commons.configuration.ConfigurationException;
16 import org.apache.commons.configuration.HierarchicalConfiguration;
17 import org.apache.commons.configuration.XMLConfiguration;
18 import org.apache.logging.log4j.LogManager;
19 import org.apache.logging.log4j.Logger;
20 import org.w3c.dom.Document;
21 import org.w3c.dom.Node;
22
23 import
   ↪ edu.illinois.ece.ftstorage.shared.exceptions.ClientConfigurationValidationException;
24 import
   ↪ edu.illinois.ece.ftstorage.shared.exceptions.StorageClusterConfigurationException;
25 import
   ↪ edu.illinois.ece.ftstorage.shared.exceptions.StorageServerConfigurationException;
26 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
```

```
27 import edu.illinois.ece.ftstorage.shared.util.FtStorageUtil;
28
29 public class StorageServerConfigurationParser {
30
31   protected final static Logger logger = LogManager.getLogger(Thread
32       .currentThread().getStackTrace()[1].getClassName());
33
34   private String uid;
35
36   private static Map<String, StorageServerConfigurationParser> instances;
37
38   private static final String defaultConfigFilePath =
   ↪ "META-INF/serverConfig.xml";
39
40   private String configFilePath = "";
41
42   private XMLConfiguration config;
43
44   private XMLConfiguration configure() throws ConfigurationException {
45     File configFile;
46     XMLConfiguration configuration = null;
47     try {
48       if ((configFile = new File(this.configFilePath)).isFile()) {
49         this.configFilePath = configFile.getCanonicalPath();
50         configuration = this.configure(configFile.getCanonicalPath());
51       } else if ((configFile = new File(defaultConfigFilePath)).isFile()) {
52         this.configFilePath = configFile.getCanonicalPath();
53         configuration = this.configure(configFile.getCanonicalPath());
54       } else if ((configFile = new File("src/main/resources/"
55           + defaultConfigFilePath)).isFile()) {
56         this.configFilePath = configFile.getCanonicalPath();
57         // assume we are in the project root directory
58         configuration = this.configure(configFile.getCanonicalPath());
59       } else {
60         this.configFilePath = configFile.getCanonicalPath();
61         throw new ConfigurationException(
62             "ERROR: No server config file found. Please check your
             ↪ classpath and/or verify the location of the file.");
63       }
64     } catch (IOException e) {
65       throw new ConfigurationException(
66           "ERROR: Could not open or read configuration file at "
67               + this.configFilePath, e);
68     }
69     return configuration;
70   }
71
72   private XMLConfiguration configure(String configurationFilePath)
73       throws ConfigurationException {
74     XMLConfiguration configuration = null;
75     configuration = new XMLConfiguration(configurationFilePath);
76     if (this.validate(configuration) == false) {
77       return null;
78     }
79     return configuration;
80   }
81
82   public synchronized static StorageServerConfigurationParser getInstance(
83       String uid) throws ConfigurationException {
84     StorageServerConfigurationParser instance;
85     if (instances == null) {
```

```
86      instances = new ConcurrentHashMap<String,
        ↪ StorageServerConfigurationParser>();
87      instance = new StorageServerConfigurationParser();
88    } else {
89      if (instances.containsKey(uid)) {
90        instance = instances.get(uid);
91      } else {
92        instance = new StorageServerConfigurationParser();
93        instances.put(uid, instance);
94      }
95    }
96    instance.uid = uid;
97    return instance;
98  }
99
100 public static StorageServerConfigurationParser getInstance()
101     throws ConfigurationException {
102   return StorageServerConfigurationParser.getInstance("default");
103 }
104
105 private boolean validate(XMLConfiguration configuration)
106     throws ClientConfigurationValidationException {
107   Iterator<String> iter = configuration.getKeys("Servers.Server[@uid]");
108   ConcurrentHashMap<String, Integer> uidCounter = new
      ↪ ConcurrentHashMap<>();
109   while (iter.hasNext()) {
110     String keyValue = iter.next();
111     for (String uid : configuration.getStringArray(keyValue)) {
112       if (uidCounter.containsKey(uid)) {
113         throw new ClientConfigurationValidationException(
114             "ERROR: Found duplicate StorageServer UID (" + uid
115                 + ") in config file.");
116       } else {
117         uidCounter.put(uid.toString(), 1);
118       }
119     }
120   }
121   return true;
122 }
123
124 private StorageServerConfigurationParser() throws ConfigurationException
    ↪ {
125   this.config = this.configure();
126 }
127
128 public XMLConfiguration getConfig() throws ConfigurationException {
129   if (instances == null) {
130     getInstance(this.uid);
131   }
132   return this.config;
133 }
134
135 public List<StoragePackage> getPackages() throws ConfigurationException {
136   // TODO store record of stored packages which will be read by a storage
137   // entity
138   List<StoragePackage> spl = new ArrayList<StoragePackage>();
139   // String uid;
140   // String name;
141   // String description;
142   // for (HierarchicalConfiguration xmlPackage : config
143   // .configurationsAt("Clusters.Cluster")) {
144   // name = xmlPackage.getString("[@name]");
```

```
145     // uid = xmlPackage.getString("[@uid]");
146     // description = xmlPackage.getString("Description");
147     // StoragePackage storageCluster = new StoragePackage(uid, name,
148     // description);
149     // StoragePackage storagePackage = new StoragePackage(
150     // InetAddress.getLocalHost(), 9100, xmlPackage);
151     // StoragePackage storagePackage = new StoragePackage("hello world!");
152     // (InetAddress senderIP, int senderPort, InetAddress targetIP,
153     // int targetPort, UUID id, byte[] data, byte[] hashValue)
154     // for (HierarchicalConfiguration clusterNode : xmlPackage
155     // .configurationsAt("Nodes.Node")) {
156     // StorageNode node = null;
157     // try {
158     // node = StorageNodeFactory.fromXmlConfigurationAndCluster(
159     // clusterNode, storageCluster);
160     // } catch (StorageNodeConfigurationException e) {
161     // throw new ConfigurationException(
162     // "ERROR: Could not create new storage node.", e);
163     // }
164     // if (node != null) {
165     // storageCluster.addNode(node);
166     // }
167     // }
168     // spl.add(storagePackage);
169     // }
170     return spl;
171   }
172 
173   public List<StorageServer> getServers() throws ConfigurationException,
174       IOException {
175     List<StorageServer> ssl = new LinkedList<StorageServer>();
176     for (HierarchicalConfiguration server : this.config
177         .configurationsAt("Servers.Server")) {
178       StorageServer storageServer = StorageServerFactory
179           .fromXmlConfiguration(server);
180       ssl.add(storageServer);
181     }
182     return ssl;
183   }
184 
185   protected void saveConfigFile() throws ConfigurationException {
186     this.validate(this.config);
187     this.config.save();
188     logger.info("Successfully saved config file to {}.",
189         this.config.getFile());
190   }
191 
192   private void setConfig(XMLConfiguration config) {
193     try {
194       String configFilePath = this.config.getFileName();
195       this.config.setFileName(configFilePath + ".bak");
196       this.saveConfigFile();
197     } catch (ConfigurationException e) {
198       logger.error("Caught {}. Error was: {}", e.getClass(),
199           e.getMessage());
200       return;
201     }
202     logger.info("Successfully created backup of old config file at {}.",
203         this.config.getFile());
204     this.config = config;
205     try {
206       this.saveConfigFile();
```

```
207     } catch (ConfigurationException e) {
208       logger.error("Caught {}. Error was: {}", e.getClass(),
209           e.getMessage());
210       return;
211     }
212     logger.info("Successfully saved new config file.");
213   }
214 
215   public void setConfigFilePath(String configFilePath) {
216     this.configFilePath = configFilePath;
217   }
218 
219   public void setDocument(Document newDocument,
220       String newDocumentSubfolderPath) throws ConfigurationException,
221       StorageClusterConfigurationException {
222     Node nodeType = newDocument.getFirstChild().getAttributes()
223         .getNamedItem("type");
224     if (nodeType == null) {
225       throw new StorageClusterConfigurationException(
226           "Attribute 'type' is not set for root node in XML configuration.
          ↪ Expecting 'ServerConfig' as a value.");
227     }
228     if (nodeType.getNodeValue().equals("ServerConfig")) {
229       logger.debug("Setting new ServerConfig.");
230       XMLConfiguration newConfig = (XMLConfiguration) this.getConfig()
231           .clone();
232       try {
233         newConfig.setDocumentBuilder(DocumentBuilderFactory
234             .newInstance().newDocumentBuilder());
235         newConfig
236             .getDocumentBuilder()
237             .getDOMImplementation()
238             .createDocument(newDocument.getNamespaceURI(), null,
239                 newDocument.getDoctype())
240             .importNode(newDocument.getFirstChild(), false);
241         newConfig.setFileName(FtStorageUtil
242             .getPathWithNewParentFolderName(this.config.getFile(),
243                 newDocumentSubfolderPath));
244         this.setConfig(newConfig);
245       } catch (ParserConfigurationException e) {
246         logger.error("Caught {}. Error was: {}", e.getClass(),
247             e.getMessage());
248       }
249     } else {
250       String message = String
251           .format("ERROR: Could not set new document in %s. Attribute
          ↪ 'type' has an unrecognized value %s. Please verify the
          ↪ structure of the document, in particular the root node
          ↪ element %s.",
252               this.getClass(), nodeType, newDocument
253                   .getFirstChild().getNodeName());
254       throw new StorageServerConfigurationException(message);
255     }
256   }
257 }
```

Listing A.4: server.StorageServerConfigurationParser

```
1 package edu.illinois.ece.ftstorage.server;
2 
3 import java.io.IOException;
4 import java.util.List;
```

```
5
6 import org.apache.commons.configuration.ConfigurationException;
7 import org.apache.commons.configuration.HierarchicalConfiguration;
8 import org.apache.logging.log4j.LogManager;
9 import org.apache.logging.log4j.Logger;
10
11 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
12 import
   ↪ edu.illinois.ece.ftstorage.shared.storageCluster.StorageClusterConfigurationParser;
13 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
14 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNodeFactory;
15
16 public class StorageServerFactory {
17
18   protected final static Logger logger = LogManager.getLogger(Thread
19       .currentThread().getStackTrace()[1].getClassName());
20
21   public static StorageServer fromXmlConfiguration(
22       HierarchicalConfiguration serverElement) throws IOException,
23       ConfigurationException {
24     List<StorageCluster> scl = null;
25     StorageServer storageServer;
26     StorageNode storageNode = null;
27     try {
28       scl = StorageClusterConfigurationParser.getInstance().getClusters();
29     } catch (ConfigurationException e) {
30       logger.fatal(
31           "Caught {}. Error getting cluster configuration. Specific
           ↪ message was: {}",
32           e.getClass(), e.getMessage());
33     }
34     String serverUid = serverElement.getString("[@uid]");
35     String serverName = serverElement.getString("[@name]");
36     String serverType = serverElement.getString("[@type]");
37     String nodeUid = serverElement.getString("[@storageNode]");
38     String serverDescription = serverElement.getString("Description");
39     for (StorageCluster storageCluster : scl) {
40       for (StorageNode node : storageCluster.getNodeList()) {
41         if (node.getUid().equals(nodeUid)) {
42           storageNode = node;
43         }
44       }
45     }
46     if (storageNode == null) {
47       storageNode = StorageNodeFactory.getDefaultStorageNode();
48     }
49     switch (serverType) {
50     case "DefaultServer":
51       storageServer = new StorageServer(serverUid, serverName,
52           serverDescription, storageNode);
53       break;
54     default:
55       logger.warn("Server Type " + serverType
56           + " not recognized. Using default value.");
57       storageServer = new StorageServer(serverUid, serverName,
58           serverDescription, storageNode);
59       break;
60     }
61     storageServer.setStorageNode(storageNode);
62     return storageServer;
63   }
64
```

```
65   public static StorageServer fromXmlConfigurationAndNode(
66       HierarchicalConfiguration serverElement, StorageNode storageNode)
67       throws IOException, ConfigurationException {
68     StorageServer storageServer;
69     String serverUid = serverElement.getString("[@uid]");
70     String serverName = serverElement.getString("[@name]");
71     String serverType = serverElement.getString("[@type]");
72     String serverDescription = serverElement.getString("Description");
73     switch (serverType) {
74     case "DefaultServer":
75       storageServer = new StorageServer(serverUid, serverName,
76           serverDescription, storageNode);
77       break;
78     default:
79       logger.warn("Server Type " + serverType
80           + " not recognized. Using default value.");
81       storageServer = new StorageServer(serverUid, serverName,
82           serverDescription, storageNode);
83       break;
84     }
85     storageServer.setStorageNode(storageNode);
86     return storageServer;
87   }
88
89 }
```

Listing A.5: server.StorageServerFactory

```
1 package edu.illinois.ece.ftstorage.server;
2
3 import java.io.IOException;
4 import java.net.InetAddress;
5 import java.util.HashMap;
6 import java.util.LinkedList;
7 import java.util.List;
8 import java.util.Map;
9 import java.util.concurrent.BlockingQueue;
10
11 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
12 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
13 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
14 import
   ↪ edu.illinois.ece.ftstorage.shared.ftProtocol.DefaultFTProtocolMetadata;
15 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
16 import edu.illinois.ece.ftstorage.shared.storagePackage.HashingSchemeType;
17 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
18 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
19
20 public class StorageServerPackageHandler extends AbstractPackageHandler {
21
22   final StorageServer requestor;
23   final StorageNode requestee;
24   private boolean verified;
25
26   public StorageServerPackageHandler(StorageServer requestor,
27       StorageNode requestee, StoragePackage storagePackage)
28       throws IOException {
29     super(requestor, requestee, storagePackage);
30     this.requestor = requestor;
31     this.requestee = requestee;
32   }
33
```

```
34  @Override
35  protected void read() {
36    logger.debug("Received {} on {} and reading from {}...",
37        this.storagePackage, this.requestor.getUid(), this.requestee);
38    if (!this.requestee.isPackagedStored(this.storagePackage)
39        || (this.storagePackage.getCodingScheme() !=
        ↪ CodingSchemeType.NONE)) {
40      // if we use coding or if the package is not storeds, then we need
41      // to request the other packages from other servers
42      if (!this.requestee.isPackagedStored(this.storagePackage)) {
43        logger.trace(
44            "Package {} is not stored locally. Requesting it from other
            ↪ servers...",
45            this.storagePackage);
46      } else {
47        logger.trace(
48            "We are using {} {} and no hashing. Decoding is happening on
            ↪ server-side. Requesting packages from other servers...",
49            CodingSchemeType.class.getName(), this.storagePackage
50                .getCodingScheme().getClass().getName());
51        if (this.storagePackage.getHashingScheme() !=
        ↪ HashingSchemeType.NONE) {
52
53        }
54      }
55      int clusterSize = this.requestee.getStorageCluster().getNodeList()
56          .size();
57      for (int i = 0; i < clusterSize; i++) {
58        StorageNode storageNode = this.requestee.getStorageCluster()
59            .getNodeList().get(i);
60        if (!storageNode.getUid().equals(
61            this.requestor.getStorageNode().getUid())) {
62          this.storagePackage.setClientListeningIP(this.requestor
63              .getSocketAddress());
64          this.storagePackage.setClientListeningPort(this.requestor
65              .getSocketPort());
66          this.writeToTcpAsync(this.storagePackage,
67              storageNode.getStorageServerAddress(),
68              storageNode.getStorageServerPort());
69        }
70      }
71    } else {
72      // add this.storagePackage so that in receiveReadAckFromNodeToNode
73      // we will return the stored value in this.requestee to the client
74      // this.addPendingPackage(this.storagePackage);
75      if (this.requestee.isPackagedStored(this.storagePackage)) {
76        StoragePackage storedPackage = this.requestee
77            .getStoredPackage(this.storagePackage);
78        if (storedPackage != null) {
79          // just in case another thread decided to remove the stored
80          // package in the meantime, we wann be on the safe side
81          this.storagePackage.setData(storedPackage.getData());
82          this.storagePackage.setHashValue();
83        } else {
84          logger.warn(
85              "Another thread removed the stored package {} from {} in the
              ↪ meantime!",
86              this.storagePackage, this.requestee.getUid());
87        }
88      } else {
89        logger.warn(
```

```
90              "Another thread removed the stored package {} from {} in the
                ↪ meantime!",
91              this.storagePackage, this.requestee.getUid());
92          }
93          this.verified = true;
94        }
95      }
96
97      @Override
98      protected void receiveReadAck() {
99        if (!this.requestee.isPackagedStored(this.storagePackage)
100           || (this.storagePackage.getCodingScheme() !=
              ↪ CodingSchemeType.NONE)) {
101         logger.trace("Waiting for responses from servers...");
102       }
103       int numNodes = this.requestee.getStorageCluster().getNodeList().size();
104       int numMaxFaults = (numNodes - 1) / 3;
105       int numReceivedAcks = 0;
106       Map<InetAddress, Integer> clientList = new HashMap<>();
107       clientList.put(this.storagePackage.getClientListeningIP(),
108           this.storagePackage.getClientListeningPort());
109       List<StoragePackage> codedPackageList = new
          ↪ LinkedList<StoragePackage>();
110       BlockingQueue<StoragePackage> pendingPackageQueue = null;
111       synchronized (this.receivedPackages) {
112         pendingPackageQueue = this.receivedPackages.get(this.storagePackage
113             .getUid());
114       }
115       while (!this.verified && !this.timerExpired()) {
116         StoragePackage pendingPackage = pendingPackageQueue.peek();
117         if (pendingPackage != null) {
118           pendingPackageQueue.poll();
119           logger.trace("Processing {}.", pendingPackage);
120           byte[] key;
121           int numMatchingPackages = 0;
122           if (this.storagePackage.getCodingScheme() ==
              ↪ CodingSchemeType.NONE) {
123             if (pendingPackage.getClientListeningIP().equals(
124                 this.storagePackage.getClientListeningIP())
125                 && pendingPackage.getClientListeningPort() ==
                    ↪ this.storagePackage
126                     .getClientListeningPort()) {
127               if (!this.storagePackage.getHashingScheme().equals(
128                   HashingSchemeType.NONE)) {
129                 key = pendingPackage.getData();
130               } else {
131                 pendingPackage.setHashValue();
132                 key = pendingPackage.getHashValue();
133               }
134               if (this.pendingReadPackages.containsKey(key)) {
135                 numMatchingPackages = this.pendingReadPackages
136                     .get(key);
137                 numMatchingPackages++;
138               }
139               numReceivedAcks++;
140               this.pendingReadPackages.put(key, numMatchingPackages);
141               if (numMatchingPackages > 0) {
142                 logger.trace("Received {} matching read packages.",
143                     numMatchingPackages);
144                 this.verified = true;
145               }
146             } else if (!clientList.containsKey(pendingPackage
```

```
147                 .getClientListeningIP())) {
148               clientList.put(pendingPackage.getClientListeningIP(),
149                   pendingPackage.getClientListeningPort());
150             }
151           } else {
152             if (pendingPackage.getClientListeningIP().equals(
153                 this.storagePackage.getClientListeningIP())
154                 && pendingPackage.getClientListeningPort() ==
                    ↪ this.storagePackage
155                     .getClientListeningPort()) {
156               // only attempt to decode packages designated for
157               // the same client
158               codedPackageList.add(pendingPackage);
159             } else if (!clientList.containsKey(pendingPackage
160                 .getClientListeningIP())) {
161               // another client (or server) requested the same package
162               // in the meantime. No need to decode everything twice,
163               // so we just return the decoded data to this node as
164               // well
165               clientList.put(pendingPackage.getClientListeningIP(),
166                   pendingPackage.getClientListeningPort());
167             }
168             if (codedPackageList.size() > numMaxFaults) {
169               numReceivedAcks++;
170               StoragePackage decodedPackage;
171               if ((decodedPackage = pendingPackage
172                   .decode(codedPackageList)) != null) {
173                 logger.trace("Successfully decoded {}.",
174                     decodedPackage);
175                 this.storagePackage.setData(decodedPackage
176                     .getData());
177                 this.storagePackage.setHashValue();
178                 this.verified = true;
179               }
180             }
181           }
182           if (numReceivedAcks > 0 && this.verified == true) {
183             this.storagePackage.setSpeculative(false);
184             this.closeServerSocket();
185             break;
186           }
187         }
188       }
189       logger.trace("Responding to clients from {}...",
190           this.requestor.getUid());
191       DefaultFTProtocolMetadata metadata = new DefaultFTProtocolMetadata(
192           this.requestor.getUid());
193       this.storagePackage.setMetadata(metadata);
194       StoragePackage ackPackage = this.storagePackage.createAckPackage();
195       for (InetAddress clientAddress : clientList.keySet()) {
196         this.writeToTcpAsync(ackPackage, clientAddress,
197             clientList.get(clientAddress));
198       }
199     }
200
201     @Override
202     protected void receiveWriteAck() {
203
204     }
205
206     @Override
207     protected void write() throws ProtocolException {
```

```
208     logger.debug("Received {} from {} and writing to {}...",
209         this.storagePackage, this.requestor.getUid(), this.requestee);
210     if (this.storagePackage.getPackageType() != StoragePackageType.ACK) {
211       if (!this.requestee.isPackagedStored(this.storagePackage)) {
212         this.storagePackage.setSpeculative(false);
213         this.requestee.write(this.requestor, this.storagePackage);
214         this.storePackage(this.storagePackage);
215         this.writeToTcpAsync(this.storagePackage.createAckPackage(),
216             this.storagePackage.getClientListeningIP(),
217             this.storagePackage.getClientListeningPort());
218       }
219     }
220   }
221 }
```

Listing A.6: server.StorageServerPackageHandler

```
1 package edu.illinois.ece.ftstorage.shared;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.net.BindException;
7 import java.net.InetAddress;
8 import java.net.ServerSocket;
9 import java.net.Socket;
10 import java.util.Date;
11 import java.util.Map;
12 import java.util.UUID;
13 import java.util.concurrent.BlockingQueue;
14 import java.util.concurrent.ConcurrentHashMap;
15 import java.util.concurrent.LinkedBlockingQueue;
16
17 import org.apache.logging.log4j.LogManager;
18 import org.apache.logging.log4j.Logger;
19
20 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
21 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
22 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
23 import edu.illinois.ece.ftstorage.shared.util.FtStorageUtil;
24
25 public abstract class AbstractPackageHandler implements Runnable {
26
27   protected final static Logger logger = LogManager.getLogger(Thread
28       .currentThread().getStackTrace()[1].getClassName());
29
30   protected boolean isRunning = false;
31   private final Object syncObject = new Object();
32
33   protected ServerSocket serverSocket;
34   protected StoragePackage storagePackage;
35   protected final AbstractStorageEntity requestor;
36   protected final AbstractStorageEntity requestee;
37
38   protected final Map<UUID, BlockingQueue<StoragePackage>>
     ↪ processedPackages;
39   protected final Map<UUID, BlockingQueue<StoragePackage>>
     ↪ receivedAckPackages;
40   protected final Map<UUID, BlockingQueue<StoragePackage>>
     ↪ receivedPackages;
41   protected final Map<byte[], Integer> pendingReadPackages;
42
```

```
43  protected final long timeStarted = (new Date()).getTime();
44  protected final static long timeoutValue = 1000 * 300;
45
46  /**
47   * Constructor to prepare the {@code AbstractPackageHandler} for writing.
48   *
49   * @param requestor
50   *            The {@code AbstractStorageEntity} requesting that the given
51   *            {@code StoragePackage} is written.
52   * @param requestee
53   *            The {@code AbstractStorageEntity} that is supposed to
54   *            write/store the given {@code StoragePackage}.
55   * @param storagePackage
56   *            The {@code StoragePackage} to be written.
57   * @throws IOException
58   */
59  public AbstractPackageHandler(AbstractStorageEntity requestor,
60      AbstractStorageEntity requestee, StoragePackage storagePackage)
61      throws IOException {
62    this.requestor = requestor;
63    this.requestee = requestee;
64    this.storagePackage = storagePackage;
65    try {
66      this.serverSocket = new ServerSocket(0, 50,
67          InetAddress.getLocalHost());
68    } catch (BindException e) {
69      if (this.serverSocket != null) {
70        logger.error("Could not bind on port {}.",
71            this.serverSocket.getLocalPort());
72      }
73      throw e;
74    }
75    this.receivedAckPackages = new ConcurrentHashMap<UUID,
    ↪ BlockingQueue<StoragePackage>>();
76    BlockingQueue<StoragePackage> ackPackageQueue = new
    ↪ LinkedBlockingQueue<StoragePackage>();
77    this.receivedAckPackages.put(storagePackage.getUid(), ackPackageQueue);
78
79    this.receivedPackages = new ConcurrentHashMap<UUID,
    ↪ BlockingQueue<StoragePackage>>();
80    BlockingQueue<StoragePackage> receivedPackageQueue = new
    ↪ LinkedBlockingQueue<StoragePackage>();
81    this.receivedPackages
82        .put(storagePackage.getUid(), receivedPackageQueue);
83
84    this.processedPackages = new ConcurrentHashMap<UUID,
    ↪ BlockingQueue<StoragePackage>>();
85    BlockingQueue<StoragePackage> processedPackageQueue = new
    ↪ LinkedBlockingQueue<StoragePackage>();
86    this.receivedPackages.put(storagePackage.getUid(),
87        processedPackageQueue);
88    this.pendingReadPackages = new ConcurrentHashMap<byte[], Integer>();
89    logger.trace("Created {} for {} ({}) on {}.", this.serverSocket
90        .getClass().getName(), this.getClass().getName(),
91        this.storagePackage, this.serverSocket.getLocalSocketAddress());
92  }
93
94  protected Object listen() throws IOException, ClassNotFoundException {
95    Socket socket = this.serverSocket.accept();
96    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
97    Object receivedObject = in.readObject();
98    logger.trace("Received Object at {} from {}.", this.requestor.getUid(),
```

```
99          socket.getRemoteSocketAddress());
100     in.close();
101     socket.close();
102     return receivedObject;
103   }
104
105   private final void addAckPackage(StoragePackage newPackage) {
106     if (newPackage.getPackageType().equals(StoragePackageType.ACK)) {
107       logger.debug("Adding {} to ACK list for {}.", newPackage.getUid(),
108           this.requestor.getUid());
109       BlockingQueue<StoragePackage> ackPackageQueue;
110       synchronized (this.receivedAckPackages) {
111         if (this.receivedAckPackages.containsKey(newPackage.getUid())) {
112           ackPackageQueue = this.receivedAckPackages.get(newPackage
113               .getUid());
114         } else {
115           ackPackageQueue = new LinkedBlockingQueue<StoragePackage>();
116           this.receivedAckPackages.put(newPackage.getUid(),
117               ackPackageQueue);
118         }
119         ackPackageQueue.add(newPackage);
120       }
121     } else {
122       logger.warn(
123           "Attempting to add non-ACK package type ({}) to ACK list for
          ↪ {}.",
124           newPackage, this.requestor.getUid());
125     }
126   }
127
128   protected final void addProcessedPackage(StoragePackage newPackage) {
129     logger.debug("Adding {} to list of processed packages for {}.",
130         newPackage, this.requestor.getUid());
131     BlockingQueue<StoragePackage> processedPackageQueue;
132     synchronized (this.processedPackages) {
133       if (this.processedPackages.containsKey(newPackage.getUid())) {
134         processedPackageQueue = this.processedPackages.get(newPackage
135             .getUid());
136       } else {
137         processedPackageQueue = new LinkedBlockingQueue<StoragePackage>();
138         this.processedPackages.put(newPackage.getUid(),
139             processedPackageQueue);
140       }
141       processedPackageQueue.add(newPackage);
142     }
143   }
144
145   protected final boolean isPackageProcessed(StoragePackage
      ↪ packageToCheck) {
146     BlockingQueue<StoragePackage> processedPackageQueue;
147     synchronized (this.processedPackages) {
148       if (this.processedPackages.containsKey(packageToCheck.getUid())) {
149         processedPackageQueue = this.processedPackages
150             .get(packageToCheck.getUid());
151         for (StoragePackage processedPackage : processedPackageQueue) {
152           if (processedPackage.equals(packageToCheck)) {
153             return true;
154           }
155         }
156       }
157     }
158     return false;
```

```
159   }
160
161   /**
162    * Enqueue a {@link StoragePackage} to the current pending package queue
       ↪ to
163    * be processed. If the {@link StoragePackage} is of type
164    * {@link StoragePackageType#ACK}, then first check if a corresponding
165    * {@link StoragePackageType#DATA}, {@link
       ↪ StoragePackageType#APPLICATION},
166    * or {@link StoragePackageType#CONFIGURATION} package has already been
167    * received. If not, then discard the {@link StoragePackage}, else,
       ↪ enqueue
168    * it in the corresponding pending package queue.
169    *
170    * @param storagePackage
171    */
172   public final void addPendingPackage(StoragePackage storagePackage) {
173     if (storagePackage == null) {
174       logger.error("Cannot add an uninitialized StoragePackage to pending
          ↪ queue!");
175       return;
176     }
177     logger.debug("Adding {} to pending list for {}.", storagePackage,
178         this.requestor.getUid());
179     BlockingQueue<StoragePackage> pendingPackageQueue;
180     synchronized (this.receivedPackages) {
181       if (!this.receivedPackages.containsKey(storagePackage.getUid())) {
182         logger.trace("{} not in received package list.",
183             storagePackage.getUid());
184         if (storagePackage.getPackageType().equals(
185             StoragePackageType.ACK)) {
186           logger.debug(
187               "Received {} for {}, but no pending {}, {}, or {} package
                  ↪ handlers running.",
188               storagePackage.getPackageType(),
189               storagePackage.getUid(), StoragePackageType.DATA,
190               StoragePackageType.APPLICATION,
191               StoragePackageType.CONFIGURATION);
192           return;
193         } else {
194           BlockingQueue<StoragePackage> ackPackageQueue = new
              ↪ LinkedBlockingQueue<StoragePackage>();
195           pendingPackageQueue = new LinkedBlockingQueue<StoragePackage>();
196           this.receivedPackages.put(storagePackage.getUid(),
197               pendingPackageQueue);
198           synchronized (this.receivedAckPackages) {
199             this.receivedAckPackages.put(storagePackage.getUid(),
200                 ackPackageQueue);
201           }
202         }
203       } else {
204         if (storagePackage.getPackageType().equals(
205             StoragePackageType.ACK)) {
206           this.addAckPackage(storagePackage);
207           return;
208         } else {
209           pendingPackageQueue = this.receivedPackages
210               .get(storagePackage.getUid());
211         }
212       }
213       pendingPackageQueue.add(storagePackage);
214       logger.trace("Size of package queue for {} is now {}.",
```

```
215             storagePackage, pendingPackageQueue.size());
216       }
217       synchronized (this.syncObject) {
218         logger.trace("Got lock on {}/{} syncObject.",
219             this.requestor.getUid(), this.requestee.getUid());
220         if (!this.isRunning()) {
221           if (this.storagePackage.getPackageType() ==
          ↪ StoragePackageType.ACK) {
222             // we launched a package handler with an ACK package. add
223             // the ACK package to the ackQueue and exchange it with the
224             // current StoragePackage to be enqueued
225             if (this.requestee.isPackagedStored(storagePackage)) {
226               logger.debug("Package is stored. Exchanging stored package {}
                ↪ with current package {} and enqueuing it.");
227               this.addAckPackage(this.storagePackage);
228               this.storagePackage = this.requestee
229                   .getStoredPackage(storagePackage);
230             } else {
231               logger.debug("Package is not stored yet. Exchanging received
                ↪ package {} with current package {} and enqueuing it.");
232               this.addAckPackage(this.storagePackage);
233               this.storagePackage = storagePackage;
234             }
235           }
236           this.startPackageHandler();
237         }
238       }
239       logger.trace("Released lock on {}/{} syncObject.",
240           this.requestor.getUid(), this.requestee.getUid());
241     }
242 
243     /**
244      * Close the ServerSocket, if it is bound.
245      */
246     public final synchronized void closeServerSocket() {
247       if (this.serverSocket != null && this.serverSocket.isBound()) {
248         FtStorageUtil.closeSilently(this.serverSocket);
249       }
250     }
251 
252     /**
253      * Get the number of packages which are in the queue pending to be
       ↪ written.
254      *
255      * @return Number of packages which are in the queue pending to be
       ↪ written.
256      */
257     public final int getNumPendingWrittenPackages() {
258       int numPendingWrittenPackages = 0;
259       synchronized (this.receivedPackages) {
260         for (UUID uuid : this.receivedPackages.keySet()) {
261           numPendingWrittenPackages += this.receivedPackages.get(uuid)
262               .size();
263         }
264       }
265       return numPendingWrittenPackages;
266     }
267 
268     /**
269      * Get the number of packages in the queue for received ACK packages.
270      *
271      * @return Number of packages in the queue for received ACK packages.
```

```
272      */
273     public final int getNumReceivedAckPackages() {
274       int numReceivedAckPackages = 0;
275       synchronized (this.receivedAckPackages) {
276         for (UUID uuid : this.receivedAckPackages.keySet()) {
277           numReceivedAckPackages += this.receivedAckPackages.get(uuid)
278               .size();
279         }
280       }
281       return numReceivedAckPackages;
282     }
283 
284     public final void clearPackageQueues() {
285       logger.debug("Clearing all package queues...");
286       synchronized (this.receivedAckPackages) {
287         for (BlockingQueue<StoragePackage> queue : this.receivedAckPackages
288             .values()) {
289           queue.clear();
290         }
291         this.receivedAckPackages.clear();
292         logger.debug("Cleared all receivedAckPackages...");
293       }
294       synchronized (this.receivedPackages) {
295         for (BlockingQueue<StoragePackage> queue : this.receivedPackages
296             .values()) {
297           queue.clear();
298         }
299         this.receivedPackages.clear();
300         logger.debug("Cleared all receivedPackages...");
301       }
302       synchronized (this.processedPackages) {
303         for (BlockingQueue<StoragePackage> queue : this.processedPackages
304             .values()) {
305           queue.clear();
306         }
307         this.processedPackages.clear();
308         logger.debug("Cleared all processedPackages...");
309       }
310       synchronized (this.pendingReadPackages) {
311         this.pendingReadPackages.clear();
312         logger.debug("Cleared all pendingReadPackages...");
313       }
314     }
315 
316     private final void clearPackageQueues(UUID uuid) {
317       logger.debug("Clearing all package queues for uuid...", uuid);
318       synchronized (this.receivedAckPackages) {
319         if (this.receivedAckPackages.containsKey(uuid)) {
320           BlockingQueue<StoragePackage> queue = this.receivedAckPackages
321               .get(uuid);
322           queue.clear();
323           this.receivedAckPackages.remove(uuid);
324         }
325         logger.debug("Cleared receivedAckPackages...");
326       }
327       synchronized (this.receivedPackages) {
328         if (this.receivedPackages.containsKey(uuid)) {
329           BlockingQueue<StoragePackage> queue = this.receivedPackages
330               .get(uuid);
331           queue.clear();
332           this.receivedPackages.remove(uuid);
333         }
```

```
334       logger.debug("Cleared receivedPackages...");
335     }
336     synchronized (this.processedPackages) {
337       if (this.receivedPackages.containsKey(uuid)) {
338         BlockingQueue<StoragePackage> queue = this.processedPackages
339             .get(uuid);
340         queue.clear();
341         this.processedPackages.remove(uuid);
342       }
343       logger.debug("Cleared processedPackages...");
344     }
345     synchronized (this.pendingReadPackages) {
346       this.pendingReadPackages.clear();
347       logger.debug("Cleared all pendingReadPackages...");
348     }
349   }
350
351   /**
352    * Query if an ACK package for a given {@code StoragePackage} has been
353    * received or not. Note this only detects whether there is an entry in
       ↪ the
354    * queue of ACK packages for a given {@code UUID}.
355    *
356    * @param storagePackage
357    *            The {@code StoragePackage} whose {@code UUID} is used to
358    *            determine if an ACK package exists.
359    * @return {@code true} if the queue of ACK packages contains a record
       ↪ with
360    *         the passed {@code StoragePackage}'s UUID.
361    */
362   protected boolean isAckReceived(StoragePackage storagePackage) {
363     return this.receivedAckPackages.containsKey(storagePackage.getUid());
364   }
365
366   protected boolean isPackagePending(StoragePackage storagePackage) {
367     return this.receivedPackages.containsKey(storagePackage.getUid());
368   }
369
370   protected boolean isAckPackageQueueEmpty() {
371     boolean allEmpty = true;
372     synchronized (this.syncObject) {
373       logger.trace("Got lock on {}/{} syncObject.",
374           this.requestor.getUid(), this.requestee.getUid());
375       synchronized (this.receivedAckPackages) {
376         for (BlockingQueue<StoragePackage> queue : this.receivedAckPackages
377             .values()) {
378           if (!queue.isEmpty()) {
379             allEmpty = false;
380           }
381         }
382         logger.trace(
383             "Released lock on {}/{} syncObject (returning {}).",
384             this.requestor.getUid(), this.requestee.getUid(),
385             allEmpty);
386         return allEmpty;
387       }
388     }
389   }
390
391   protected boolean isPendingPackageQueueEmpty() {
392     boolean allEmpty = true;
393     synchronized (this.syncObject) {
```

```
394       logger.trace("Got lock on {}/{} syncObject.",
395           this.requestor.getUid(), this.requestee.getUid());
396       synchronized (this.receivedPackages) {
397         for (BlockingQueue<StoragePackage> queue : this.receivedPackages
398             .values()) {
399           if (!queue.isEmpty()) {
400             allEmpty = false;
401           }
402         }
403         logger.trace(
404             "Released lock on {}/{} syncObject (returning {}).",
405             this.requestor.getUid(), this.requestee.getUid(),
406             allEmpty);
407         return allEmpty;
408       }
409     }
410   }
411
412   protected abstract void read() throws ProtocolException;
413
414   protected abstract void receiveReadAck();
415
416   protected abstract void receiveWriteAck();
417
418   protected final void storePackage(StoragePackage storedPackage) {
419     logger.info("{} storing package {} on {}/{}.", this.getClass()
420         .getName(), storedPackage.getUid(), this.requestor.getUid(),
421         this.requestee.getUid());
422     this.requestor.storePackage(storedPackage);
423     this.requestee.storePackage(storedPackage);
424   }
425
426   protected abstract void write() throws ProtocolException;
427
428   protected final boolean writeToTcp(StoragePackage storagePackage,
429       InetAddress targetIp, int targetPort) {
430     logger.debug("Sending {} to {}:{}", storagePackage, targetIp,
431         targetPort);
432     if (storagePackage.getSourceAckListeningIP() == null) {
433       storagePackage.setSourceAckListeningIP(this.serverSocket
434           .getInetAddress());
435     }
436     if (storagePackage.getSourceAckListeningPort() == 0) {
437       storagePackage.setSourceAckListeningPort(this.serverSocket
438           .getLocalPort());
439     }
440     Socket socket = null;
441     ObjectOutputStream out = null;
442     try {
443       socket = new Socket(targetIp, targetPort);
444       out = new ObjectOutputStream(socket.getOutputStream());
445       out.writeObject(storagePackage);
446       out.flush();
447     } catch (IOException e) {
448       logger.error("Could not send object to {}:{}. Message was \"{}\".",
449           targetIp, targetPort, e.getMessage());
450       return false;
451     } finally {
452       FtStorageUtil.closeSilently(out, socket);
453     }
454     logger.debug("Successfully sent {} to {}:{} from {}.", storagePackage,
455         targetIp, targetPort, socket.getLocalSocketAddress());
```

```
456     return true;
457   }
458
459   protected final void writeToTcpAsync(final StoragePackage storagePackage,
460       final InetAddress targetIp, final int targetPort) {
461     logger.debug("Sending (async) {} to {}:{}", storagePackage, targetIp,
462         targetPort);
463     Thread t = new Thread(new Runnable() {
464       @Override
465       public void run() {
466         if (storagePackage.getSourceAckListeningIP() == null) {
467           storagePackage.setSourceAckListeningIP(serverSocket
468               .getInetAddress());
469         }
470         if (storagePackage.getSourceAckListeningPort() == 0) {
471           storagePackage.setSourceAckListeningPort(serverSocket
472               .getLocalPort());
473         }
474         Socket socket = null;
475         ObjectOutputStream out = null;
476         try {
477           socket = new Socket(targetIp, targetPort);
478           out = new ObjectOutputStream(socket.getOutputStream());
479           out.writeObject(storagePackage);
480           out.flush();
481         } catch (IOException e) {
482           logger.error(
483               "Could not send object to {}:{}. Message was \"{}\".",
484               targetIp, targetPort, e.getMessage());
485           return;
486         } finally {
487           FtStorageUtil.closeSilently(out, socket);
488         }
489         logger.debug("Successfully sent StoragePackage to {}:{}.",
490             targetIp, targetPort);
491       }
492     });
493     t.start();
494   }
495
496   public boolean isRunning() {
497     synchronized (this.syncObject) {
498       return this.isRunning;
499     }
500   }
501
502   /**
503    * Attempt to start the abstract package handler and run it in a separate
504    * thread if it is currently not running.
505    *
506    * @return true if the package handler was started, false if it was
        ↪ already
507    *         running
508    */
509   public boolean startPackageHandler() {
510     logger.trace("Attempting to start {}/{}", this.requestor.getUid(),
511         this.requestee.getUid());
512     synchronized (this.syncObject) {
513       logger.trace("Got lock on {}/{} syncObject.",
514           this.requestor.getUid(), this.requestee.getUid());
515       if (!this.isRunning()) {
516         Thread t = new Thread(this);
```

```
517         logger.trace("Restarting {} for {} in {}.", this.getClass()
518             .getName(), this.storagePackage, t.getName());
519         t.start();
520         logger.trace(
521             "Released lock on {}/{} syncObject (returning {}).",
522             this.requestor.getUid(), this.requestee.getUid(), true);
523         return true;
524       } else {
525         logger.trace("{} for {} is already running.", this.getClass()
526             .getName(), this.storagePackage);
527       }
528     }
529     logger.trace("Released lock on {}/{} syncObject (retuning {}).",
530         this.requestor.getUid(), this.requestee.getUid(), false);
531     return false;
532   }
533 
534   @Override
535   public final void run() {
536     boolean run = false;
537     synchronized (this.syncObject) {
538       logger.trace("Got lock on {}/{} syncObject.",
539           this.requestor.getUid(), this.requestee.getUid());
540       if (!this.isRunning()) {
541         this.isRunning = true;
542         run = true;
543       } else {
544         run = false;
545       }
546     }
547     logger.trace("Released lock on {}/{} syncObject.",
548         this.requestor.getUid(), this.requestee.getUid());
549     if (run) {
550       if (this.serverSocket.isClosed()) {
551         try {
552           this.serverSocket = new ServerSocket(0);
553           logger.trace("Reopened listening server socket on {}:{}.",
554               this.serverSocket.getInetAddress(),
555               this.serverSocket.getLocalPort());
556         } catch (IOException e) {
557           logger.error(
558               "Caught unchecked exception {} with message {}.", e
559                   .getClass().getName(), e.getMessage());
560         }
561       }
562       logger.trace("Starting {}.", this.getClass().getName());
563       // determine if we are going to write or read a StoragePackage
564       if (this.storagePackage != null
565           && this.storagePackage.getPackageType() !=
          ↪ StoragePackageType.READ) {
566         this.storagePackage.setSpeculative(true);
567         this.requestor.startStorageEntity();
568         try {
569           this.write();
570           this.receiveWriteAck();
571         } catch (ProtocolException e) {
572           logger.error("Caught {} while writing to {} from {}.", e
573               .getClass().getName(), this.requestee.getUid(),
574               this.requestor.getUid());
575           logger.error("Message was {}.", e.getMessage());
576         }
577       } else if (this.storagePackage != null
```

```
578           && this.storagePackage.getPackageType() ==
          ↪ StoragePackageType.READ) {
579         this.requestor.startStorageEntity();
580         try {
581           this.read();
582           this.receiveReadAck();
583         } catch (ProtocolException e) {
584           logger.error("Caught {} while {} was reading from {}.", e
585               .getClass().getName(), this.requestee.getUid(),
586               this.requestor.getUid());
587           logger.error("Message was {}.", e.getMessage());
588         }
589       } else {
590         logger.warn("Need to set StoragePackage in order to write/read.
          ↪ Not performing any operation...");
591       }
592       boolean removeStoppedPackageHandlers = false;
593       synchronized (this.syncObject) {
594         logger.trace("Got lock on {}/{} syncObject.",
595             this.requestor.getUid(), this.requestee.getUid());
596         if (this.serverSocket.isBound()) {
597           this.closeServerSocket();
598         }
599         this.isRunning = false;
600         // the requestee decides when to clean up the package handlers.
601         // i.e. the requestor delegates the read/write command to the
602         // requestee, which then returns to the requestor when the
603         // operation has completed.
604         if (this.requestor.isPackagedStored(this.storagePackage)) {
605           logger.trace("Package is stored on {}.",
606               this.requestor.getUid());
607           if (this.requestee.isPackagedStored(this.storagePackage)) {
608             logger.trace("Package is stored on {}.",
609                 this.requestee.getUid());
610
611             // cannot have an AbstractStorageEntity calling a
612             // function that blocks on an AbstractPackageHandler
613             // locking on syncObject (since we are already locking
614             // on syncObject here!) ... so we move this after the
615             // syncLoop. We don't care if
616             // this.requestor.removeStoppedPackageHandlers is called
617             // multiple times by different threads, because it is
618             // thread safe. Since the package is already stored, we
619             // can safely clear remove all stopped package handlers
620             // and clear the queues. We risk that a newer package
621             // with the same UUID is received in the meantime, but
622             // the client can send it again in this case.
623             removeStoppedPackageHandlers = true;
624           } else {
625             logger.trace(
626                 "Package stored on requestee {}? {} speculative? {}",
627                 this.requestee.getUid(),
628                 this.requestee
629                     .isPackagedStored(this.storagePackage),
630                 this.requestee
631                     .isPackageSpeculative(this.storagePackage
632                         .getUid()));
633           }
634         } else {
635           logger.trace(
636               "Package stored on requestor {}? {} speculative? {}",
637               this.requestor.getUid(), this.requestor
```

```
638                     .isPackagedStored(this.storagePackage),
639                 this.requestor
640                     .isPackageSpeculative(this.storagePackage
641                         .getUid()));
642           }
643           logger.trace("Exiting {}.", this.getClass());
644         }
645         logger.trace("Released lock on {}/{} syncObject.",
646             this.requestor.getUid(), this.requestee.getUid());
647         if (removeStoppedPackageHandlers) {
648           this.requestee.removePackageHandlers(this.storagePackage
649               .getUid());
650           this.requestor.removeStoppedPackageHandlers(this.storagePackage
651               .getUid());
652           this.clearPackageQueues(this.storagePackage.getUid());
653         }
654       } else {
655         logger.trace("{} for {} already running.", this.getClass()
656             .getName(), this.storagePackage);
657       }
658     }
659
660     protected boolean timerExpired() {
661       if (((new Date()).getTime() - this.timeStarted) > timeoutValue) {
662         logger.info("Timer expired in {}.", this.getClass().getName());
663         return true;
664       }
665       return false;
666     }
667 }
```

Listing A.7: shared.AbstractPackageHandler

```
 1 package edu.illinois.ece.ftstorage.shared;
 2
 3 import java.io.IOException;
 4 import java.io.Serializable;
 5 import java.net.BindException;
 6 import java.net.InetAddress;
 7 import java.net.InetSocketAddress;
 8 import java.net.ServerSocket;
 9 import java.util.ArrayList;
10 import java.util.Map;
11 import java.util.UUID;
12 import java.util.concurrent.ConcurrentHashMap;
13
14 import org.apache.logging.log4j.LogManager;
15 import org.apache.logging.log4j.Logger;
16
17 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
18 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
19
20 public abstract class AbstractStorageEntity implements Serializable,
   ↪ Runnable {
21
22   /**
23    *
24    */
25   private static final long serialVersionUID = -5268319014458194145L;
26   protected final static Logger logger = LogManager.getLogger(Thread
27       .currentThread().getStackTrace()[1].getClassName());
28   protected final static int INITIAL_NUMBER_OF_STORED_PACKAGES = 50;
```

```
29
30    protected boolean isRunning = false;
31    protected final Object syncObject = new Object();
32
33    protected final String uid;
34    protected final String name;
35    protected final String description;
36    protected final Map<UUID, ArrayList<AbstractPackageHandler>>
      ↪ runningPackageHandlers;
37    protected Map<UUID, StoragePackage> storedPackages;
38    protected ServerSocket serverSocket;
39
40    public AbstractStorageEntity(String uid, String name, String
      ↪ description) {
41      this.uid = uid;
42      this.name = name;
43      this.description = description;
44      this.storedPackages = new ConcurrentHashMap<UUID, StoragePackage>(
45          INITIAL_NUMBER_OF_STORED_PACKAGES);
46      this.runningPackageHandlers = new ConcurrentHashMap<UUID,
        ↪ ArrayList<AbstractPackageHandler>>(
47          INITIAL_NUMBER_OF_STORED_PACKAGES);
48      try {
49        this.serverSocket = new ServerSocket(0, 50,
50            InetAddress.getLocalHost());
51      } catch (BindException e) {
52        if (this.serverSocket != null) {
53          logger.error("Could not bind on port {}.",
54              this.serverSocket.getLocalPort());
55        }
56      } catch (IOException e) {
57        logger.error("Caught {}. Error was: {}", e.getClass(),
58            e.getMessage());
59      }
60      if (this.serverSocket != null) {
61        logger.trace("Created {} for {} ({}) on {}.", this.serverSocket
62            .getClass().getName(), this.getClass().getName(), this
63            .getUid(), this.serverSocket.getLocalSocketAddress());
64      } else {
65        logger.error("Failed to create ServerSocket.");
66      }
67    }
68
69    /**
70     * Add a package handler to the list of package handlers for this storage
71     * entity.
72     *
73     * @param abstractPackageHandler
74     * @param packageUuid
75     */
76    protected void addRunningPackageHandler(
77        AbstractPackageHandler abstractPackageHandler, UUID packageUuid) {
78      ArrayList<AbstractPackageHandler> aphl;
79      synchronized (this.runningPackageHandlers) {
80        logger.trace("Got lock on {} runningPackageHandlers.",
81            this.getUid());
82        if (this.runningPackageHandlers.containsKey(packageUuid)) {
83          aphl = this.runningPackageHandlers.get(packageUuid);
84        } else {
85          aphl = new ArrayList<AbstractPackageHandler>(
86              INITIAL_NUMBER_OF_STORED_PACKAGES);
87          this.runningPackageHandlers.put(packageUuid, aphl);
```

```
88        }
89        logger.debug(
90            "Adding new {} on {} for StoragePackage {} with size {}.",
91            abstractPackageHandler.getClass().getName(), this.getUid(),
92            packageUuid, aphl.size());
93        aphl.add(abstractPackageHandler);
94      }
95      logger.trace("Released lock on {} runningPackageHandlers.",
96          this.getUid());
97    }
98
99    /**
100    * Close the server sockets of all package handlers for a given {@code
       ↪ UUID}
101    * (of a {@code StoragePackage}).
102    *
103    * @param uuid
104    *            The {@code UUID} of the {@code StoragePackage} for which
105    *            associated {@code AbstractPackageHandler}s are running.
106    */
107   public void closePackageHandlerSockets(UUID uuid) {
108     synchronized (this.runningPackageHandlers) {
109       logger.trace("Got lock on {} runningPackageHandlers.",
110           this.getUid());
111       if (this.runningPackageHandlers.containsKey(uuid)) {
112         for (AbstractPackageHandler abstractPackageHandler :
         ↪ this.runningPackageHandlers
113             .get(uuid)) {
114           if (abstractPackageHandler != null) {
115             abstractPackageHandler.closeServerSocket();
116           }
117         }
118       }
119       logger.trace("Released lock on {} runningPackageHandlers.",
120           this.getUid());
121       this.runningPackageHandlers.remove(uuid);
122     }
123   }
124
125   /**
126    * Close the server sockets of ALL package handlers.
127    */
128   protected final void closePackageHandlerSockets() {
129     synchronized (this.runningPackageHandlers) {
130       logger.trace("Got lock on {} runningPackageHandlers.",
131           this.getUid());
132       for (UUID uuid : this.runningPackageHandlers.keySet()) {
133         this.closePackageHandlerSockets(uuid);
134       }
135     }
136     logger.trace("Released lock on {} runningPackageHandlers.",
137         this.getUid());
138   }
139
140   public final String getDescription() {
141     return this.description;
142   }
143
144   public final String getName() {
145     return this.name;
146   }
147
```

```
148    public final String getUid() {
149      return this.uid;
150    }
151
152    /**
153     * Returns whether or not a package is stored based on the passed
       ↪ package's
154     * UUID and time stamp.
155     *
156     * @param storagePackage
157     *            The package with a given UUID and time stamp to check
       ↪ whether
158     *            it is stored.
159     * @return True if the passed package's UUID matches a stored package's
       ↪ UUID
160     *         and if the stored packages UUID has the same or lesser time
       ↪ stamp
161     *         than the passed package, false otherwise.
162     */
163    public final boolean isPackagedStored(StoragePackage storagePackage) {
164      synchronized (this.storedPackages) {
165        if (this.storedPackages.containsKey(storagePackage.getUid())) {
166          if (this.storedPackages.get(storagePackage.getUid())
167              .getTimeStamp()
168              .compareTo(storagePackage.getTimeStamp()) <= 0) {
169            return true;
170          } else {
171            logger.debug(
172                "Stored StoragePackage from {} is newer than requested
                   ↪ timestamp from {}.",
173                this.storedPackages.get(storagePackage.getUid())
174                    .getTimeStamp(), storagePackage
175                    .getTimeStamp());
176          }
177        }
178      }
179      return false;
180    }
181
182    public final boolean removeStoragePackage(StoragePackage storagePackage)
       ↪ {
183      synchronized (this.storedPackages) {
184        if (this.isPackagedStored(storagePackage)) {
185          logger.debug("Removing {} from {}.", storagePackage,
186              this.getUid());
187          this.storedPackages.remove(storagePackage.getUid());
188          this.removePackageHandlers(storagePackage.getUid());
189          return true;
190        }
191      }
192      logger.debug("Could not remove {}. Not found on {}.", storagePackage,
193          this.getUid());
194      return false;
195    }
196
197    public final boolean isPackageSpeculative(UUID uuid) {
198      synchronized (this.storedPackages) {
199        // logger.trace("Got lock on {} storedPackages.", this.getUid());
200        if (this.storedPackages.containsKey(uuid)
201            && this.storedPackages.get(uuid).isSpeculative()) {
202          // logger.trace(
203          // "Released lock on {} storedPackages (returning {}).",
```

```
204          // this.getUid(), true);
205          return true;
206        } else {
207          logger.debug("Package is stored on {}? {}", this.getUid(),
208              this.storedPackages.containsKey(uuid));
209        }
210      }
211      // logger.trace("Released lock on {} storedPackages (returning {}).",
212      // this.getUid(), false);
213      return false;
214    }
215
216    public final StoragePackage getStoredPackage(StoragePackage
       ↪ storagePackage) {
217      synchronized (this.storedPackages) {
218        // logger.trace("Got lock on {} storedPackages.", this.getUid());
219        if (this.isPackagedStored(storagePackage)) {
220          // logger.trace(
221          // "Released lock on {} storedPackages (returning {}).",
222          // this.getUid(),
223          // this.storedPackages.get(storagePackage.getUid()));
224          return this.storedPackages.get(storagePackage.getUid());
225        }
226      }
227      // logger.trace("Released lock on {} storedPackages (returning {}).",
228      // this.getUid(), null);
229      return null;
230    }
231
232    public final boolean isRunning() {
233      synchronized (this.syncObject) {
234        logger.trace("Got lock on {} syncObject.", this.getUid());
235        logger.trace("Released lock on {} syncObject (returning {}).",
236            this.getUid(), this.isRunning);
237        return this.isRunning;
238      }
239    }
240
241    public abstract void read(AbstractStorageEntity requestor,
242        StoragePackage storagePackage) throws ProtocolException;
243
244    /**
245     * Remove all {@code AbstractPackageHandlers} and close their
246     * {@code ServerSockets} for a given {@code UUID}.
247     *
248     * @param packageUuid
249     *            The {@code UUID} associated with the
250     *            {@code AbstractPackageHandlers} to close.
251     */
252    protected final void removePackageHandlers(UUID packageUuid) {
253      synchronized (this.runningPackageHandlers) {
254        logger.trace("Got lock on {} runningPackageHandlers.",
255            this.getUid());
256        if (this.runningPackageHandlers.containsKey(packageUuid)) {
257          ArrayList<AbstractPackageHandler> runningPackageHandlers =
             ↪ this.runningPackageHandlers
258              .get(packageUuid);
259          logger.trace("Removing {} {} package handlers on {}.",
260              runningPackageHandlers.size(), packageUuid,
261              this.getUid());
262          for (AbstractPackageHandler runningPackageHandler :
             ↪ runningPackageHandlers) {
```

```
263          runningPackageHandler.closeServerSocket();
264        }
265        runningPackageHandlers.clear();
266        this.runningPackageHandlers.remove(packageUuid);
267      }
268    }
269    logger.trace("Released lock on {} runningPackageHandlers.",
270        this.getUid());
271  }
272
273  /**
274   * Remove all running {@code AbstractPackageHandlers} and close their
275   * {@code ServerSockets} for a given {@code UUID}.
276   *
277   * @param packageUuid
278   *            The {@code UUID} associated with the
279   *            {@code AbstractPackageHandlers} to close.
280   */
281  protected final void removeRunningPackageHandlers(UUID packageUuid) {
282    synchronized (this.runningPackageHandlers) {
283      logger.trace("Got lock on {} runningPackageHandlers.",
284          this.getUid());
285      if (this.runningPackageHandlers.containsKey(packageUuid)) {
286        ArrayList<AbstractPackageHandler> runningPackageHandlers =
           ↪ this.runningPackageHandlers
287            .get(packageUuid);
288        logger.trace("Removing {} running {} package handlers on {}.",
289            runningPackageHandlers.size(), packageUuid,
290            this.getUid());
291        for (int i = 0; i < runningPackageHandlers.size(); i++) {
292          AbstractPackageHandler aph = runningPackageHandlers.get(i);
293          if (aph.isRunning()) {
294            aph.closeServerSocket();
295            runningPackageHandlers.remove(i);
296          }
297        }
298        if (runningPackageHandlers.isEmpty()) {
299          this.runningPackageHandlers.remove(packageUuid);
300        }
301      }
302    }
303    logger.trace("Released lock on {} runningPackageHandlers.",
304        this.getUid());
305  }
306
307  /**
308   * Remove all stopped {@code AbstractPackageHandlers} and close their
309   * {@code ServerSockets} for a given {@code UUID}.
310   *
311   * @param packageUuid
312   *            The {@code UUID} associated with the
313   *            {@code AbstractPackageHandlers} to close.
314   */
315  protected final void removeStoppedPackageHandlers(UUID packageUuid) {
316    synchronized (this.runningPackageHandlers) {
317      logger.trace("Got lock on {} runningPackageHandlers.",
318          this.getUid());
319      if (this.runningPackageHandlers.containsKey(packageUuid)) {
320        ArrayList<AbstractPackageHandler> runningPackageHandlers =
           ↪ this.runningPackageHandlers
321            .get(packageUuid);
322        logger.trace("Removing {} stopped {} package handlers on {}.",
```

```
323             runningPackageHandlers.size(), packageUuid,
324             this.getUid());
325         for (int i = 0; i < runningPackageHandlers.size(); i++) {
326           AbstractPackageHandler aph = runningPackageHandlers.get(i);
327           if (!aph.isRunning()) {
328             aph.closeServerSocket();
329             runningPackageHandlers.remove(i);
330           }
331         }
332         if (runningPackageHandlers.isEmpty()) {
333           this.runningPackageHandlers.remove(packageUuid);
334         }
335       } else {
336         logger.debug("No package handlers present for {} on {}",
337             packageUuid, this.getUid());
338       }
339     }
340     logger.trace("Released lock on {} runningPackageHandlers.",
341         this.getUid());
342   }
343
344   /**
345    * Attempt to start the storage entity and run it in a separate thread
       ↪ if it
346    * is currently not running.
347    *
348    * @return true if the storage entity was started, false if it was
       ↪ already
349    *         running
350    */
351   public boolean startStorageEntity() {
352     logger.trace("Attempting to start {}", this.getUid());
353     synchronized (this.syncObject) {
354       logger.trace("Got lock on {} syncObject.", this.getUid());
355       if (!this.isRunning()) {
356         Thread t = new Thread(this);
357         logger.trace("(Re)starting {} {} in {}.", this.getClass()
358             .getName(), this.getUid(), t.getName());
359         t.start();
360         logger.trace("Released lock on {} syncObject (returning {}).",
361             this.getUid(), true);
362         return true;
363       } else {
364         logger.trace("{} {} is already running.", this.getClass()
365             .getName(), this.getUid());
366         logger.trace("Released lock on {} syncObject (returning {}).",
367             this.getUid(), false);
368         return false;
369       }
370     }
371   }
372
373   /**
374    * Add a given {@code StoragePackage} to the queue of stored packages and
375    *
376    * @param storagePackage
377    */
378   public final void storePackage(StoragePackage storagePackage) {
379     if (storagePackage != null) {
380       logger.debug("Storing package {} on {}.", storagePackage,
381           this.getUid());
382       synchronized (this.storedPackages) {
```

```
383         // logger.trace("Got lock on {} storedPackages.",
384         // this.getUid());
385         if (!this.isPackagedStored(storagePackage)) {
386           this.storedPackages.put(storagePackage.getUid(),
387               storagePackage);
388           // this.removeRunningPackageHandlers(storedPackage.getUid());
389           // logger.debug("Storing speculative package? {}",
390           // storagePackage.isSpeculative());
391         } else {
392           logger.debug("Package {} already stored on {}.",
393               storagePackage.getUid(), this.getUid());
394         }
395       }
396       // logger.trace("Released lock on {} storedPackages.",
397       // this.getUid());
398     } else {
399       logger.warn("Cannot store null value.");
400     }
401   }
402
403   public final InetAddress getSocketAddress() {
404     if (this.serverSocket.isBound()) {
405       InetSocketAddress isa = (InetSocketAddress) this.serverSocket
406           .getLocalSocketAddress();
407       return isa.getAddress();
408     }
409     return null;
410   }
411
412   public final int getSocketPort() {
413     if (this.serverSocket.isBound()) {
414       return this.serverSocket.getLocalPort();
415     }
416     return 0;
417   }
418
419   @Override
420   public String toString() {
421     StringBuilder sb = new StringBuilder();
422     try {
423       sb.append(this.getClass().getName() + " [");
424       sb.append("UID=" + this.uid);
425       sb.append(", ");
426       sb.append("storedPackages=" + this.storedPackages.size());
427       sb.append(']');
428     } catch (IllegalArgumentException e) {
429       e.printStackTrace();
430     }
431     return sb.toString();
432   }
433
434   public abstract void write(AbstractStorageEntity requester,
435       StoragePackage storagePackage) throws ProtocolException;
436 }
```

Listing A.8: shared.AbstractStorageEntity

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
3 import java.util.List;
4
5 import org.apache.logging.log4j.LogManager;
```

```
6 import org.apache.logging.log4j.Logger;
7
8 import edu.illinois.ece.ftstorage.shared.storagePackage.ICodable;
9
10 public abstract class AbstractCodingScheme {
11
12   protected final static Logger logger = LogManager.getLogger(Thread
13       .currentThread().getStackTrace()[1].getClassName());
14
15   public AbstractCodingScheme() {
16   }
17
18   public abstract byte[] decode(ICodable codedObject, List<byte[]> data);
19
20   public abstract List<byte[]> encode(ICodable codableObject,
21       int numDataBlocks);
22 }
```

Listing A.9: shared.codingScheme.AbstractCodingScheme

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
3 import java.io.Serializable;
4
5 public abstract class AbstractCodingSchemeMetadata implements Serializable
  ↪ {
6
7   private static final long serialVersionUID = -214401165878813629L;
8
9   public AbstractCodingSchemeMetadata() {
10   }
11
12 }
```

Listing A.10: shared.codingScheme.AbstractCodingSchemeMetadata

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
3 import java.lang.reflect.Constructor;
4 import java.lang.reflect.InvocationTargetException;
5 import java.util.HashMap;
6 import java.util.Map;
7
8 import org.apache.logging.log4j.LogManager;
9 import org.apache.logging.log4j.Logger;
10
11 @SuppressWarnings({ "unchecked", "rawtypes" })
12 public class CodingSchemeFactory {
13
14   protected final static Logger logger = LogManager.getLogger(Thread
15       .currentThread().getStackTrace()[1].getClassName());
16   private static CodingSchemeFactory instance = new CodingSchemeFactory();
17
18   private Map<CodingSchemeType, Class> codingSchemes;
19
20   public static AbstractCodingScheme get(final CodingSchemeType
     ↪ codingScheme) {
21     Class[] constructorArgs = new Class[] {};
22     Constructor<AbstractCodingScheme> ctor;
23     if (!codingScheme.equals(CodingSchemeType.NONE)) {
24       try {
```

```
25          ctor = getInstance().codingSchemes.get(codingScheme)
26              .getConstructor(constructorArgs);
27          return ctor.newInstance();
28        } catch (SecurityException | InstantiationException
29            | IllegalAccessException | IllegalArgumentException
30            | InvocationTargetException | NoSuchMethodException e) {
31          logger.error(
32              "Class constructor({}) not defined or not visible for {}.
                ↪ Using default class {}. Message was: {}.",
33              constructorArgs, codingScheme,
34              DefaultCodingScheme.class, e.getMessage());
35        }
36      }
37      return new DefaultCodingScheme();
38    }
39
40    public static CodingSchemeFactory getInstance() {
41      if (instance == null) {
42        instance = new CodingSchemeFactory();
43      }
44      return instance;
45    }
46
47    private CodingSchemeFactory() {
48      if (instance == null) {
49        this.codingSchemes = new HashMap<CodingSchemeType, Class>(
50            CodingSchemeType.values().length);
51        for (CodingSchemeType type : CodingSchemeType.values()) {
52          Class clazz;
53          String className = this.getClass().getPackage().getName() + "."
54              + type;
55          try {
56            if (!type.equals(CodingSchemeType.NONE)) {
57              clazz = Class.forName(className);
58              this.codingSchemes.put(type, clazz);
59            }
60          } catch (SecurityException | ClassNotFoundException e) {
61            logger.error(
62                "Class constructor not defined or not visible for {}. Using
                  ↪ default class {}. Message was: {}.",
63                className, DefaultCodingScheme.class,
64                e.getMessage());
65            clazz = DefaultCodingScheme.class;
66          }
67        }
68      }
69    }
70  }
```

Listing A.11: shared.codingScheme.CodingSchemeFactory

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
3 public enum CodingSchemeType {
4   NONE, XOR, ReedSolomon;
5 }
```

Listing A.12: shared.codingScheme.CodingSchemeType

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
```

```
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import edu.illinois.ece.ftstorage.shared.storagePackage.ICodable;
8
9 public class DefaultCodingScheme extends AbstractCodingScheme {
10
11   @Override
12   public byte[] decode(ICodable codedObject, List<byte[]> dataList) {
13     byte[] currentNodeData = codedObject.getData();
14     for (int i = 0; i < dataList.size(); i++) {
15       byte[] otherNodeData = dataList.get(i);
16       if (!Arrays.equals(currentNodeData, otherNodeData)) {
17         logger.warn("Could not decode {} with scheme {}.",
18             codedObject.toString(), codedObject.getCodingScheme());
19         return null;
20       }
21     }
22     return currentNodeData;
23   }
24
25   @Override
26   public List<byte[]> encode(ICodable codableObject, int numDataBlocks) {
27     List<byte[]> bbl = new ArrayList<byte[]>(numDataBlocks);
28     for (int i = 0; i < numDataBlocks; i++) {
29       bbl.add(codableObject.getData());
30     }
31     return bbl;
32   }
33
34 }
```

Listing A.13: shared.codingScheme.DefaultCodingScheme

```
1 package edu.illinois.ece.ftstorage.shared.codingScheme;
2
3 import java.util.List;
4
5 import edu.illinois.ece.ftstorage.shared.storagePackage.ICodable;
6
7 public class ReedSolomon extends AbstractCodingScheme {
8
9   @Override
10   public byte[] decode(ICodable codedObject, List<byte[]> data) {
11     // TODO Auto-generated method stub
12     return null;
13   }
14
15   @Override
16   public List<byte[]> encode(ICodable codableObject, int numDataBlocks) {
17     // TODO Auto-generated method stub
18     return null;
19   }
20
21 }
```

Listing A.14: shared.codingScheme.ReedSolomon

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
```

```
3 public class ClientConfigurationValidationException extends
4     ConfigurationException {
5
6   private static final long serialVersionUID = -5580597664399570703L;
7
8   public ClientConfigurationValidationException() {
9     super();
10  }
11
12  public ClientConfigurationValidationException(String message) {
13    super(message);
14  }
15
16  public ClientConfigurationValidationException(String message,
17      Throwable cause) {
18    super(message, cause);
19  }
20
21  public ClientConfigurationValidationException(Throwable cause) {
22    super(cause);
23  }
24
25 }
```

Listing A.15: shared.exceptions.ClientConfigurationValidationException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 public class ConfigurationException extends
4     org.apache.commons.configuration.ConfigurationException {
5
6   private static final long serialVersionUID = -6565754212654756719L;
7
8   public ConfigurationException() {
9     super();
10  }
11
12  public ConfigurationException(String message) {
13    super(message);
14  }
15
16  public ConfigurationException(String message, Throwable cause) {
17    super(message, cause);
18  }
19
20  public ConfigurationException(Throwable cause) {
21    super(cause);
22  }
23 }
```

Listing A.16: shared.exceptions.ConfigurationException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 public class FTProtocolException extends ProtocolException {
4
5   private static final long serialVersionUID = 8213316015454119936L;
6
7   public FTProtocolException() {
8   }
9
```

```
10   public FTProtocolException(String message) {
11     super(message);
12   }
13
14   public FTProtocolException(Throwable cause) {
15     super(cause);
16   }
17
18   public FTProtocolException(String message, Throwable cause) {
19     super(message, cause);
20   }
21
22   public FTProtocolException(String message, Throwable cause,
23       boolean enableSuppression, boolean writableStackTrace) {
24     super(message, cause, enableSuppression, writableStackTrace);
25   }
26
27 }
```

Listing A.17: shared.exceptions.FTProtocolException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 public class ProtocolException extends Exception {
4
5   private static final long serialVersionUID = -6495021503686634238L;
6
7   public ProtocolException() {
8   }
9
10   public ProtocolException(String message) {
11     super(message);
12   }
13
14   public ProtocolException(Throwable cause) {
15     super(cause);
16   }
17
18   public ProtocolException(String message, Throwable cause) {
19     super(message, cause);
20   }
21
22   public ProtocolException(String message, Throwable cause,
23       boolean enableSuppression, boolean writableStackTrace) {
24     super(message, cause, enableSuppression, writableStackTrace);
25   }
26
27 }
```

Listing A.18: shared.exceptions.ProtocolException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 public class QuitException extends Exception {
4
5   private static final long serialVersionUID = 7577988749155221 32L;
6
7 }
```

Listing A.19: shared.exceptions.QuitException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 /**
4  * Exception is thrown when the {@code StorageClusterConfigurationParser}
   ↪ fails
5  * to create and return a proper configuration of {@code StorageCluster}s
   ↪ or
6  * fails to verify a configuration.
7  */
8 public class StorageClusterConfigurationException extends
9     ConfigurationException {
10
11   private static final long serialVersionUID = 3236392673118633848L;
12
13   public StorageClusterConfigurationException() {
14     super();
15   }
16
17   public StorageClusterConfigurationException(String message) {
18     super(message);
19   }
20
21   public StorageClusterConfigurationException(String message, Throwable
     ↪ cause) {
22     super(message, cause);
23   }
24
25   public StorageClusterConfigurationException(Throwable cause) {
26     super(cause);
27   }
28
29 }
```

Listing A.20: shared.exceptions.StorageClusterConfigurationException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 public class StorageNodeConfigurationException extends
  ↪ ConfigurationException {
4
5   private static final long serialVersionUID = 7779877743840102188L;
6
7   public StorageNodeConfigurationException() {
8     super();
9   }
10
11   public StorageNodeConfigurationException(String message) {
12     super(message);
13   }
14
15   public StorageNodeConfigurationException(String message, Throwable
     ↪ cause) {
16     super(message, cause);
17   }
18
19   public StorageNodeConfigurationException(Throwable cause) {
20     super(cause);
21   }
22 }
```

Listing A.21: shared.exceptions.StorageNodeConfigurationException

```
1 package edu.illinois.ece.ftstorage.shared.exceptions;
2
3 /**
4  * Exception is thrown when the {@code StorageServerConfigurationParser}
   ↪ fails
5  * to create and return a proper configuration of {@code StorageServer}s or
6  * fails to verify a configuration.
7  */
8 public class StorageServerConfigurationException extends
  ↪ ConfigurationException {
9
10   private static final long serialVersionUID = -8109188191602587126L;
11
12   public StorageServerConfigurationException() {
13     super();
14   }
15
16   public StorageServerConfigurationException(String message) {
17     super(message);
18   }
19
20   public StorageServerConfigurationException(String message, Throwable
     ↪ cause) {
21     super(message, cause);
22   }
23
24   public StorageServerConfigurationException(Throwable cause) {
25     super(cause);
26   }
27
28 }
```

Listing A.22: shared.exceptions.StorageServerConfigurationException

```
1 package edu.illinois.ece.ftstorage.shared.ftProtocol;
2
3 import java.io.IOException;
4 import java.util.concurrent.BlockingQueue;
5
6 import edu.illinois.ece.ftstorage.client.StorageClient;
7 import edu.illinois.ece.ftstorage.server.StorageServer;
8 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
9 import edu.illinois.ece.ftstorage.shared.exceptions.FTProtocolException;
10 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
11 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
12 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
13
14 public abstract class AbstractFTProtocolHandler extends
  ↪ AbstractPackageHandler {
15
16   final StorageCluster requestee;
17
18   protected int numMaxFaults = 0;
19   protected int numNodes = 1;
20
21   /**
22    *
23    * Constructor called on the client side to prepare the
24    * {@code AbstractFTProtocolHandler} for reading or writing.
25    *
26    * @param requestor
27    *            The entity requesting to write to or read from a source.
```

```
28      * @param requestee
29      *            The entity requested to read or write a given
       ↪ StoragePackage.
30      * @param storagePackage
31      *            The {@code StoragePackage} to be read or written.
32      * @throws IOException
33      */
34     public AbstractFTProtocolHandler(StorageClient requestor,
35         StorageCluster requestee, StoragePackage storagePackage)
36         throws IOException {
37       super(requestor, requestee, storagePackage);
38       this.requestee = requestee;
39       this.init();
40     }
41
42     /**
43      *
44      * Constructor called on the server side to prepare the
45      * {@code AbstractFTProtocolHandler} for reading or writing.
46      *
47      * @param requestor
48      *            The entity requesting to write to or read from a source.
49      * @param requestee
50      *            The entity requested to read or write a given
       ↪ StoragePackage.
51      * @param storagePackage
52      *            The {@code StoragePackage} to be written.
53      * @throws IOException
54      */
55     public AbstractFTProtocolHandler(StorageServer requestor,
56         StorageCluster requestee, StoragePackage storagePackage)
57         throws IOException {
58       super(requestor, requestee, storagePackage);
59       this.requestee = requestee;
60       this.init();
61     }
62
63     /**
64      * Initialize the FT-Protocol handler, e.g. by setting the number of
       ↪ nodes
65      * or maximum faults the protocol can handle, according to certain
66      * parameters. Also, possibly add or initialize a {@code
       ↪ StoragePackage}'s
67      * {@value AbstractStoragePackageMetaData}.
68      */
69     protected abstract void init();
70
71     /**
72      * Checks if a specific ACK has already been received from any client. We
73      * override superclass because we possibly want to check that a given ACK
74      * comes from a specific node, since a faulty node could send multiple
       ↪ ACKs
75      * and imitate another node.
76      *
77      * @param storagePackage
78      *            The StoragePackage (ACK) to check for if it is already in
       ↪ an
79      *            {@code AbstractFTProtocolHandler}'s
80      *            {@code receivedAckPackages} queue.
81      */
82     @Override
83     protected boolean isAckReceived(StoragePackage storagePackage) {
```

```
84     if (this.receivedAckPackages.containsKey(storagePackage.getUid())) {
85       BlockingQueue<StoragePackage> ackQ = this.receivedAckPackages
86           .get(storagePackage.getUid());
87       for (StoragePackage ack : ackQ) {
88         if (storagePackage.equals(ack)) {
89           return true;
90         }
91       }
92     }
93     return false;
94   }
95 
96   /**
97    * Override superclass because we possibly want to check that a given
98    * {@code StoragePackage} comes from a specific node, since a faulty node
99    * could send multiple ACKs and imitate another node. This information is
100   * kept in the AbstractStoragePackageMetadata. The equivalence is
101   * automatically checked in the StoragePackage.equals() method.
102   */
103  @Override
104  protected boolean isPackagePending(StoragePackage storagePackage) {
105    if (this.receivedPackages.containsKey(storagePackage.getUid())) {
106      logger.trace("{} is pending", storagePackage);
107      synchronized (this.receivedPackages) {
108        BlockingQueue<StoragePackage> pendingQ = this.receivedPackages
109            .get(storagePackage.getUid());
110        for (StoragePackage pendingPackage : pendingQ) {
111          if (storagePackage.equals(pendingPackage)) {
112            logger.trace("{} is already pending", storagePackage);
113            return true;
114          }
115        }
116      }
117    }
118    return false;
119  }
120 
121  /**
122   * Read a {@link StoragePackage} by means of its {@code UUID}. If the
123   * requestor is on the client side, then call {@link
124   * this#readFromCluster(StorageCluster, UUID)} and if the read request is
125   * performed on the server side, {@link this#readFromNode(StorageServer,
126   * UUID)}.
127   */
128  @Override
129  protected void read() throws FTProtocolException {
130    try {
131      if (this.requestor instanceof StorageClient) {
132        this.readFromCluster((StorageClient) this.requestor);
133      } else if (this.requestor instanceof StorageServer) {
134        this.readFromNode((StorageServer) this.requestor);
135      } else {
136        logger.error("Reading from {} has not been implemented.",
137            this.requestor.getClass().getName());
138      }
139    } catch (FTProtocolException e) {
140      logger.error("Caught {} while {} was reading from {}.", e
141          .getClass().getName(), this.requestee.getUid(),
142          this.requestor.getUid());
143      throw e;
144    }
145  }
```

```
146
147   /**
148    * Read a StoragePackage by means of its {@code UUID} on the client side.
149    *
150    * @param requestor
151    *            The storage entity requesting to read a {@link
      ↪ StoragePackage}
152    *            .
153    * @param storagePackageUuid
154    *            The {@code UUID} of the {@link StoragePackage} to be read.
155    */
156   protected abstract void readFromCluster(StorageClient requestor)
157       throws FTProtocolException;
158
159   /**
160    * Read a StoragePackage by means of its {@code UUID} on the server side.
161    *
162    * @param requestor
163    *            The storage entity requesting to read a {@link
      ↪ StoragePackage}
164    *            .
165    * @param storagePackageUuid
166    *            The {@code UUID} of the {@link StoragePackage} to be read.
167    */
168   protected abstract void readFromNode(StorageServer requestor)
169       throws FTProtocolException;
170
171   @Override
172   protected void receiveReadAck() {
173     if (this.requestor instanceof StorageClient) {
174       this.receiveReadAckFromNodeToCluster();
175     } else if (this.requestor instanceof StorageServer) {
176       this.receiveReadAckFromNodeToNode();
177     } else {
178       logger.error("Reading from {} has not been implemented.",
179           this.requestor.getClass().getName());
180     }
181   }
182
183   protected abstract void receiveReadAckFromNodeToCluster();
184
185   protected abstract void receiveReadAckFromNodeToNode();
186
187   @Override
188   protected void receiveWriteAck() {
189     if (this.requestor instanceof StorageClient) {
190       this.receiveWriteAckFromNodeToCluster();
191     } else if (this.requestor instanceof StorageServer) {
192       this.receiveWriteAckFromNodeToNode();
193     } else {
194       logger.error("Reading from {} has not been implemented.",
195           this.requestor.getClass().getName());
196     }
197   }
198
199   protected abstract void receiveWriteAckFromNodeToCluster();
200
201   protected abstract void receiveWriteAckFromNodeToNode();
202
203   @Override
204   protected void write() throws ProtocolException {
205     logger.trace("Writing from {}.", this.requestor);
```

```
206     try {
207       if (this.requestor instanceof StorageClient) {
208         this.writeToCluster((StorageClient) this.requestor);
209       } else if (this.requestor instanceof StorageServer) {
210         this.writeToNode((StorageServer) this.requestor);
211       } else {
212         logger.error("Reading from {} has not been implemented.",
213             this.requestor.getClass().getName());
214       }
215     } catch (FTProtocolException e) {
216       logger.error("Caught {} while writing to {} from {}.", e.getClass()
217           .getName(), this.requestee.getUid(), this.requestor
218           .getUid());
219       throw e;
220     }
221   }
222 
223   protected abstract void writeAckToCluster(StoragePackage storagePackage);
224 
225   protected abstract void writeAckToNode(StoragePackage storagePackage);
226 
227   protected abstract void writeToCluster(StorageClient requestor)
228       throws FTProtocolException;
229 
230   protected abstract void writeToNode(StorageServer requestor)
231       throws ProtocolException;
232 }
```

Listing A.23: shared.ftProtocol.AbstractFTProtocolHandler

```
1 package edu.illinois.ece.ftstorage.shared.ftProtocol;
2 
3 import java.io.IOException;
4 import java.net.InetAddress;
5 import java.net.UnknownHostException;
6 import java.util.ArrayList;
7 import java.util.Arrays;
8 import java.util.Date;
9 import java.util.HashMap;
10 import java.util.LinkedList;
11 import java.util.List;
12 import java.util.Map;
13 import java.util.concurrent.BlockingQueue;
14 
15 import edu.illinois.ece.ftstorage.client.StorageClient;
16 import edu.illinois.ece.ftstorage.server.StorageServer;
17 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
18 import edu.illinois.ece.ftstorage.shared.exceptions.FTProtocolException;
19 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
20 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
21 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
22 import
   ↪ edu.illinois.ece.ftstorage.shared.storagePackage.AbstractStoragePackageMetadata;
23 import edu.illinois.ece.ftstorage.shared.storagePackage.HashingSchemeType;
24 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
25 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
26 
27 public class DefaultFTProtocol extends AbstractFTProtocolHandler {
28 
29   boolean verified = false;
30 
```

```
31    // maximum milliseconds to elapse from initialization of the class
      ↪ before a
32    // timeout occurs
33    private static final long timeoutValue = 1000 * 3000;
34
35    public DefaultFTProtocol(StorageClient requestor, StorageCluster
      ↪ requestee,
36        StoragePackage storagePackage) throws IOException {
37      super(requestor, requestee, storagePackage);
38      storagePackage.setSpeculative(true);
39    }
40
41    public DefaultFTProtocol(StorageServer requestor, StorageCluster
      ↪ requestee,
42        StoragePackage storagePackage) throws IOException {
43      super(requestor, requestee, storagePackage);
44      storagePackage.setSpeculative(true);
45    }
46
47    @Override
48    protected void init() {
49      this.numNodes = this.requestee.getNodeList().size();
50      this.numMaxFaults = (this.numNodes - 1) / 3;
51    }
52
53    /**
54     * Client side
55     */
56    @Override
57    protected void readFromCluster(StorageClient requestor) {
58      logger.debug("Received {} on {} and reading from {}...",
59          this.storagePackage, requestor.getUid(), this.requestee);
60      this.storagePackage.setClientListeningIP(this.requestee
61          .getSocketAddress());
62      this.storagePackage.setClientListeningPort(this.requestee
63          .getSocketPort());
64      this.writeToTcpAsync(this.storagePackage, this.requestee
65          .getPrimaryNode().getStorageServerAddress(), this.requestee
66          .getPrimaryNode().getStorageServerPort());
67    }
68
69    /**
70     * Server side
71     */
72    @Override
73    protected void readFromNode(StorageServer requestor) {
74      logger.debug("Received {} on {} and reading from {}...",
75          this.storagePackage, requestor.getUid(), this.requestee);
76      if
        ↪ (requestor.getStorageNode().equals(this.requestee.getPrimaryNode()))
        ↪ {
77        boolean forwardMessage = true;
78        int clusterSize = this.requestee.getNodeList().size();
79        InetAddress clientListeningIP = this.storagePackage
80            .getClientListeningIP();
81        int clientListeningPort = this.storagePackage
82            .getClientListeningPort();
83        for (int i = 0; i < clusterSize; i++) {
84          InetAddress serverListeningIp = this.requestee.getNodeList()
85              .get(i).getStorageServerAddress();
86          int serverListeningPort = this.requestee.getNodeList().get(i)
87              .getStorageServerPort();
```

```
88          try {
89            InetAddress broadCastAddress = InetAddress
90                .getByAddress(new byte[] { 0, 0, 0, 0 });
91            if ((clientListeningIP.equals(serverListeningIp) ||
              ↪ clientListeningIP
92                .equals(broadCastAddress))
93                && (clientListeningPort == serverListeningPort)) {
94              // if the read request came from a known server, we do
95              // not forward the request to any other server
96              logger.trace(
97                  "Read request came from a server listening on {}:{}. Not
                    ↪ forwarding message to other servers.",
98                  clientListeningIP, clientListeningPort);
99              forwardMessage = false;
100             break;
101           }
102         } catch (UnknownHostException e) {
103           logger.warn(
104               "An exception {} has occurred. Message was: {}.", e
105                   .getClass().getName(), e.getMessage());
106         }
107       }
108       if (forwardMessage == true) {
109         logger.trace("Forwarding read request to other servers...");
110         for (int i = 0; i < clusterSize; i++) {
111           StorageNode storageNode = this.requestee.getNodeList().get(
112               i);
113           if (!storageNode.getUid().equals(
114               requestor.getStorageNode().getUid())) {
115             this.writeToTcpAsync(this.storagePackage,
116                 storageNode.getStorageServerAddress(),
117                 storageNode.getStorageServerPort());
118           }
119         }
120       }
121     }
122     if (!this.requestee.isPackagedStored(this.storagePackage)
123         || (this.storagePackage.getCodingScheme() !=
          ↪ CodingSchemeType.NONE)) {
124       // if we use coding or if the package is not storeds, then we need
125       // to request the other packages from other servers
126       if (!this.requestee.isPackagedStored(this.storagePackage)) {
127         logger.trace(
128             "Package {} is not stored locally. Requesting it from other
                ↪ servers...",
129             this.storagePackage);
130       } else {
131         logger.trace(
132             "We are using {} {} and no hashing. Decoding is happening on
                ↪ server-side. Requesting packages from other servers...",
133             CodingSchemeType.class.getName(), this.storagePackage
134                 .getCodingScheme().getClass().getName());
135         if (this.storagePackage.getHashingScheme() !=
            ↪ HashingSchemeType.NONE) {
136
137         }
138       }
139       int clusterSize = this.requestee.getNodeList().size();
140       for (int i = 0; i < clusterSize; i++) {
141         StorageNode storageNode = this.requestee.getNodeList().get(i);
142         if (!storageNode.getUid().equals(
143             requestor.getStorageNode().getUid())) {
```

```
144         this.storagePackage.setClientListeningIP(requestor
145             .getSocketAddress());
146         this.storagePackage.setClientListeningPort(requestor
147             .getSocketPort());
148         this.writeToTcpAsync(this.storagePackage,
149             storageNode.getStorageServerAddress(),
150             storageNode.getStorageServerPort());
151       }
152     }
153   } else {
154     // add this.storagePackage so that in receiveReadAckFromNodeToNode
155     // we will return the stored value in this.requestee to the client
156     // this.addPendingPackage(this.storagePackage);
157     if (this.requestee.isPackagedStored(this.storagePackage)) {
158       StoragePackage storedPackage = this.requestee
159           .getStoredPackage(this.storagePackage);
160       if (storedPackage != null) {
161         // just in case another thread decided to remove the stored
162         // package in the meantime, we wann be on the safe side
163         this.storagePackage.setData(storedPackage.getData());
164         this.storagePackage.setHashValue();
165       } else {
166         logger.warn(
167             "Another thread removed the stored package {} from {} in the
                ↪ meantime!",
168             this.storagePackage, this.requestee.getUid());
169       }
170     } else {
171       logger.warn(
172           "Another thread removed the stored package {} from {} in the
              ↪ meantime!",
173           this.storagePackage, this.requestee.getUid());
174     }
175     this.verified = true;
176   }
177 }
178
179 /**
180  * Client side
181  */
182 @Override
183 protected void receiveReadAckFromNodeToCluster() {
184   logger.trace("Waiting for responses from servers...");
185   int numReceivedAcks = 0;
186   List<StoragePackage> codedPackageList = new
      ↪ LinkedList<StoragePackage>();
187   BlockingQueue<StoragePackage> ackPackageQueue;
188   synchronized (this.receivedAckPackages) {
189     ackPackageQueue = this.receivedAckPackages.get(this.storagePackage
190         .getUid());
191   }
192   while (!this.verified && !this.timerExpired()) {
193     StoragePackage pendingPackage = ackPackageQueue.peek();
194     if (pendingPackage != null) {
195       ackPackageQueue.poll();
196       logger.trace("Processing {}.", pendingPackage);
197       byte[] key;
198       int numMatchingPackages = 0;
199       if (this.storagePackage.getCodingScheme() !=
          ↪ CodingSchemeType.NONE) {
200         // if we use some kind of coding scheme, the decoding
201         // happens on the server-side.
```

```
202           if (this.storagePackage.getHashingScheme() !=
              ↪ HashingSchemeType.NONE) {
203             key = pendingPackage.getData();
204           } else {
205             pendingPackage.setHashValue();
206             key = pendingPackage.getHashValue();
207           }
208           if (this.pendingReadPackages.containsKey(key)) {
209             numMatchingPackages = this.pendingReadPackages.get(key);
210             numMatchingPackages++;
211           }
212           numReceivedAcks++;
213           this.pendingReadPackages.put(key, numMatchingPackages);
214           if (numMatchingPackages > this.numMaxFaults) {
215             logger.trace("Received {} matching read packages.",
216                 numMatchingPackages);
217             this.storagePackage.setData(pendingPackage.getData());
218             this.storagePackage.setHashValue();
219             this.verified = true;
220           }
221         } else {
222           // we could just use the "if" statement code, but we just
223           // want to show here that the default decoding works
224           codedPackageList.add(pendingPackage);
225           numReceivedAcks++;
226           if (codedPackageList.size() > this.numMaxFaults) {
227             StoragePackage decodedPackage;
228             if ((decodedPackage = pendingPackage
229                 .decode(codedPackageList)) != null) {
230               logger.trace("Successfully decoded {}.",
231                   decodedPackage);
232               this.storagePackage.setData(decodedPackage
233                   .getData());
234               this.storagePackage.setHashValue();
235               this.verified = true;
236             }
237           }
238         }
239         if (numReceivedAcks > this.numMaxFaults && this.verified) {
240           logger.info("Marking read {} as not speculative.",
241               this.storagePackage);
242           this.storagePackage.setSpeculative(false);
243           break;
244         }
245       }
246     }
247     this.storePackage(this.storagePackage);
248     // this.requestor.storePackage(this.storagePackage);
249     // this.reque
250   }
251
252   /**
253    * Server side
254    */
255   @Override
256   protected void receiveReadAckFromNodeToNode() {
257     if (!this.requestee.isPackagedStored(this.storagePackage)
258         || (this.storagePackage.getCodingScheme() !=
          ↪ CodingSchemeType.NONE)) {
259       logger.trace("Waiting for responses from servers...");
260     } else {
261       logger.debug("{} is stored on {}.", this.storagePackage,
```

```
262          this.requestee.getUid());
263      }
264      int numReceivedAcks = 0;
265      Map<InetAddress, Integer> clientList = new HashMap<>();
266      // multiple clients might make a request before a package has been
267      // committed. we respond to each client, including the original
268      // requestor.
269      clientList.put(this.storagePackage.getClientListeningIP(),
270          this.storagePackage.getClientListeningPort());
271      List<StoragePackage> codedPackageList = new
         ↪ LinkedList<StoragePackage>();
272      BlockingQueue<StoragePackage> pendingPackageQueue = null;
273      synchronized (this.receivedPackages) {
274        pendingPackageQueue = this.receivedPackages.get(this.storagePackage
275            .getUid());
276      }
277      while (!this.verified && !this.timerExpired()) {
278        StoragePackage pendingPackage = pendingPackageQueue.peek();
279        if (pendingPackage != null) {
280          pendingPackageQueue.poll();
281          logger.trace("Processing {}.", pendingPackage);
282          byte[] key;
283          int numMatchingPackages = 0;
284          if (this.storagePackage.getCodingScheme() ==
             ↪ CodingSchemeType.NONE) {
285            if (pendingPackage.getClientListeningIP().equals(
286                this.storagePackage.getClientListeningIP())
287                && pendingPackage.getClientListeningPort() ==
                   ↪ this.storagePackage
288                    .getClientListeningPort()) {
289              if (!this.storagePackage.getHashingScheme().equals(
290                  HashingSchemeType.NONE)) {
291                key = pendingPackage.getData();
292              } else {
293                pendingPackage.setHashValue();
294                key = pendingPackage.getHashValue();
295              }
296              if (this.pendingReadPackages.containsKey(key)) {
297                numMatchingPackages = this.pendingReadPackages
298                    .get(key);
299                numMatchingPackages++;
300              }
301              numReceivedAcks++;
302              this.pendingReadPackages.put(key, numMatchingPackages);
303              if (numMatchingPackages > this.numMaxFaults) {
304                logger.trace("Received {} matching read packages.",
305                    numMatchingPackages);
306                this.verified = true;
307              }
308            } else if (!clientList.containsKey(pendingPackage
309                .getClientListeningIP())) {
310              clientList.put(pendingPackage.getClientListeningIP(),
311                  pendingPackage.getClientListeningPort());
312            }
313          } else {
314            if (pendingPackage.getClientListeningIP().equals(
315                this.storagePackage.getClientListeningIP())
316                && pendingPackage.getClientListeningPort() ==
                   ↪ this.storagePackage
317                    .getClientListeningPort()) {
318              // only attempt to decode packages designated for
319              // the same client
```

```
320             codedPackageList.add(pendingPackage);
321           } else if (!clientList.containsKey(pendingPackage
322               .getClientListeningIP())) {
323             // another client (or server) requested the same package
324             // in the meantime. No need to decode everything twice,
325             // so we just return the decoded data to this node as
326             // well
327             clientList.put(pendingPackage.getClientListeningIP(),
328                 pendingPackage.getClientListeningPort());
329           }
330           if (codedPackageList.size() > this.numMaxFaults) {
331             numReceivedAcks++;
332             StoragePackage decodedPackage;
333             if ((decodedPackage = pendingPackage
334                 .decode(codedPackageList)) != null) {
335               logger.trace("Successfully decoded {}.",
336                   decodedPackage);
337               this.storagePackage.setData(decodedPackage
338                   .getData());
339               this.storagePackage.setHashValue();
340               this.verified = true;
341             }
342           }
343         }
344         if (numReceivedAcks > this.numMaxFaults
345             && this.verified == true) {
346           logger.info("Marking {} as non-speculative.",
347               this.storagePackage);
348           this.storagePackage.setSpeculative(false);
349           this.closeServerSocket();
350           break;
351         }
352       }
353     }
354     logger.trace("Responding to clients from {}...",
355         this.requestor.getUid());
356     DefaultFTProtocolMetadata metadata = new DefaultFTProtocolMetadata(
357         this.requestor.getUid());
358     this.storagePackage.setMetadata(metadata);
359     StoragePackage ackPackage = this.storagePackage.createAckPackage();
360     for (InetAddress clientAddress : clientList.keySet()) {
361       this.writeToTcpAsync(ackPackage, clientAddress,
362           clientList.get(clientAddress));
363     }
364   }
365
366   /**
367    * Client side
368    */
369   @Override
370   public void receiveWriteAckFromNodeToCluster() {
371     logger.trace("Waiting for write ACKs...");
372     int numVerifiedAcks = 0;
373     int numProcessedAcks = 0;
374     ArrayList<StoragePackage> decodingQueue = new
        ↪ ArrayList<StoragePackage>(
375         this.requestee.getNodeList().size());
376     BlockingQueue<StoragePackage> ackPackageQueue = null;
377     StoragePackage ackPackage = null;
378     synchronized (this.receivedAckPackages) {
379       if (this.receivedAckPackages.containsKey(this.storagePackage
380           .getUid())) {
```

```
381             ackPackageQueue = this.receivedAckPackages
382                 .get(this.storagePackage.getUid());
383           } else {
384             logger.error("ACK package queue is null.");
385           }
386         }
387         while (!this.verified && !this.timerExpired()) {
388           ackPackage = ackPackageQueue.peek();
389           boolean currentPackageVerified = false;
390           if (ackPackage != null) {
391             ackPackageQueue.poll();
392             logger.debug("{} ACK packages remaining to be processed...",
393                 ackPackageQueue.size());
394             if (ackPackage.getPackageType().equals(StoragePackageType.ACK)) {
395               decodingQueue.add(ackPackage);
396               numProcessedAcks++;
397               currentPackageVerified = false;
398               AbstractStoragePackageMetadata metadata = ackPackage
399                   .getMetadata(DefaultFTProtocolMetadata.class);
400               DefaultFTProtocolMetadata defaultFTProtocolMetadata =
                  ↪ (DefaultFTProtocolMetadata) metadata;
401               if (!this.storagePackage.getHashingScheme().equals(
402                   HashingSchemeType.NONE)) {
403                 // if we are using hashing to verify data quickly
404                 ackPackage.setData(this.storagePackage.getData());
405                 if (ackPackage.verifyHashValue()) {
406                   currentPackageVerified = true;
407                 }
408               } else {
409                 // if we are not using hashing we need to verify the
410                 // entire data structure
411                 if (this.storagePackage.getCodingScheme() ==
                    ↪ CodingSchemeType.NONE) {
412                   if (Arrays.equals(ackPackage.getData(),
413                       this.storagePackage.getData())) {
414                     currentPackageVerified = true;
415                   }
416                 } else {
417                   if (decodingQueue.size() > this.numMaxFaults) {
418                     StoragePackage decodedPackage = this.storagePackage
419                         .decode(decodingQueue);
420                     if (decodedPackage != null) {
421                       for (StoragePackage codedPackage : decodingQueue) {
422                         numVerifiedAcks++;
423                         InetAddress sourceListeningAddress = codedPackage
424                             .getSourceAckListeningIP();
425                         int sourceListeningPort = codedPackage
426                             .getSourceAckListeningPort();
427                         ackPackage
428                             .setSourceAckListeningIP(this.requestee
429                                 .getSocketAddress());
430                         ackPackage
431                             .setSourceAckListeningPort(this.requestee
432                                 .getSocketPort());
433                         logger.trace(
434                             "Matching package received from {}",
435                             defaultFTProtocolMetadata
436                                 .getSourceUuid());
437                         metadata = new DefaultFTProtocolMetadata(
438                             this.requestor.getUid());
439                         ackPackage.setMetadata(metadata);
440                         this.writeToTcpAsync(ackPackage,
```

```
441                         sourceListeningAddress,
442                         sourceListeningPort);
443                     logger.debug(
444                         "Processed/verified {}/{} ACKs so far...",
445                         numProcessedAcks,
446                         numVerifiedAcks);
447                   }
448                   this.storagePackage.setData(decodedPackage
449                       .getData());
450                 }
451               }
452             }
453           }
454           if (currentPackageVerified == true) {
455             numVerifiedAcks++;
456             InetAddress sourceListeningAddress = ackPackage
457                 .getSourceAckListeningIP();
458             int sourceListeningPort = ackPackage
459                 .getSourceAckListeningPort();
460             logger.trace(
461                 "Matching package received from {} ({}:{})",
462                 defaultFTProtocolMetadata.getSourceUuid(),
463                 sourceListeningAddress, sourceListeningPort);
464             logger.debug("Processed/verified {}/{} ACKs so far...",
465                 numProcessedAcks, numVerifiedAcks);
466             if (this.requestee
467                 .isPackagedStored(this.storagePackage)
468                 && !this.requestee
469                     .isPackageSpeculative(this.storagePackage
470                         .getUid())) {
471               StoragePackage storedPackage = this.requestee
472                   .getStoredPackage(this.storagePackage);
473               // The package was already stored and verified. It
474               // is possible that in future executions less than f
475               // ACKs are received and the loop continues until
476               // the timeout occurs.
477               if (storedPackage.getTimeStamp().equals(
478                   ackPackage.getTimeStamp())
479                   && !storedPackage.isSpeculative()) {
480                 this.closeServerSocket();
481                 break;
482               }
483             }
484           } else {
485             logger.warn(
486                 "Invalid hash/data value detected for package: {} and
                 ↪ HashingScheme {}.",
487                 ackPackage, ackPackage.getHashingScheme());
488             if (this.storagePackage.getHashingScheme() ==
             ↪ HashingSchemeType.NONE) {
489               logger.trace("Comparing {} with ACK package {}.",
490                   this.storagePackage.getData(),
491                   ackPackage.getHashValue());
492             } else {
493               logger.trace("Comparing {} with ACK package {}.",
494                   this.storagePackage.getHashValue(),
495                   ackPackage.getHashValue());
496             }
497           }
498         }
499       }
500       if (numVerifiedAcks > this.numMaxFaults) {
```

```
501         logger.trace(
502             "Storing StoragePackage {} and marking as non-speculative.",
503             this.storagePackage.getUid());
504         this.storagePackage.setSpeculative(false);
505         this.storePackage(this.storagePackage);
506         this.verified = true;
507         DefaultFTProtocolMetadata clientDefaultFTProtocolMetadata = new
            ↪ DefaultFTProtocolMetadata(
508             this.requestor.getUid());
509         for (StorageNode node : this.requestee.getNodeList()) {
510           // send ACKs to all servers
511           this.storagePackage.setClientListeningIP(this.requestee
512               .getSocketAddress());
513           this.storagePackage.setClientListeningPort(this.requestee
514               .getSocketPort());
515           ackPackage.setSourceAckListeningIP(node
516               .getStorageServerAddress());
517           ackPackage.setSourceAckListeningPort(node
518               .getStorageServerPort());
519           ackPackage.setMetadata(clientDefaultFTProtocolMetadata);
520           this.writeToTcp(ackPackage, node.getStorageServerAddress(),
521               node.getStorageServerPort());
522         }
523         this.closeServerSocket();
524         break;
525       }
526       if (this.requestee.isPackagedStored(this.storagePackage)
527           && !this.requestee.isPackageSpeculative(this.storagePackage
528               .getUid())
529           && this.requestor.isPackagedStored(this.storagePackage)
530           && !this.requestor.isPackageSpeculative(this.storagePackage
531               .getUid()) && ackPackageQueue.isEmpty()) {
532         logger.trace("Package is stored and not speculative.");
533         this.closeServerSocket();
534         this.verified = true;
535         break;
536       }
537     }
538   }
539
540   /**
541    * Server side
542    */
543   @Override
544   public void receiveWriteAckFromNodeToNode() {
545     logger.trace("Waiting for write ACKs...");
546     boolean packageWasStored = false;
547     ArrayList<StoragePackage> decodingQueue = new
        ↪ ArrayList<StoragePackage>(
548         this.requestee.getNodeList().size());
549     BlockingQueue<StoragePackage> ackPackageQueue = null;
550     StoragePackage ackPackage = null;
551     synchronized (this.receivedAckPackages) {
552       if (this.receivedAckPackages.containsKey(this.storagePackage
553           .getUid())) {
554         ackPackageQueue = this.receivedAckPackages
555             .get(this.storagePackage.getUid());
556       } else {
557         logger.error("ACK package queue is null (has not been initialized
            ↪ yet, or has been deleted).");
558       }
559     }
```

```
560     BlockingQueue<StoragePackage> pendingPackageQueue = null;
561     synchronized (this.receivedPackages) {
562       if (this.receivedPackages.containsKey(this.storagePackage.getUid()))
          ↪ {
563         pendingPackageQueue = this.receivedPackages
564             .get(this.storagePackage.getUid());
565       } else {
566         logger.error("Pending package queue is null.");
567       }
568     }
569     int numProcessedPackages = 0;
570     int numMatchingPackages = 0;
571     int numVerifiedAcks = 0;
572     int numProcessedAcks = 0;
573     while (!this.verified && !this.timerExpired()) {
574       // the first package received from the primary counts as the first
575       // matching package;
576       boolean currentPackageVerified = false;
577       ackPackage = ackPackageQueue.peek();
578       if (ackPackage != null) {
579         ackPackageQueue.poll();
580         decodingQueue.add(ackPackage);
581         logger.debug(
582             "{} ACKs remaining to be processed...Processing {}.",
583             ackPackageQueue.size(), ackPackage);
584         AbstractStoragePackageMetadata metadata = ackPackage
585             .getMetadata(DefaultFTProtocolMetadata.class);
586         DefaultFTProtocolMetadata defaultFTProtocolMetadata =
            ↪ (DefaultFTProtocolMetadata) metadata;
587         if (ackPackage.getPackageType().equals(StoragePackageType.ACK)) {
588           numProcessedAcks++;
589           currentPackageVerified = false;
590           if (!this.storagePackage.getHashingScheme().equals(
591               HashingSchemeType.NONE)) {
592             // if we are using hashing verify data quickly
593             ackPackage.setData(this.storagePackage.getData());
594             if (ackPackage.verifyHashValue()) {
595               currentPackageVerified = true;
596             }
597           } else {
598             // if we are not using hashing we need to verify the
599             // entire data structure
600             if (this.storagePackage.getCodingScheme() ==
                ↪ CodingSchemeType.NONE) {
601               if (Arrays.equals(ackPackage.getData(),
602                   this.storagePackage.getData())) {
603                 currentPackageVerified = true;
604               }
605             } else {
606               StoragePackage decodedPackage = this.storagePackage
607                   .decode(decodingQueue);
608               if (decodedPackage != null) {
609                 currentPackageVerified = true;
610                 for (StoragePackage codedPackage : decodingQueue) {
611                   numVerifiedAcks++;
612                   ackPackage
613                       .setClientListeningIP(codedPackage
614                           .getClientListeningIP());
615                   ackPackage
616                       .setClientListeningPort(codedPackage
617                           .getClientListeningPort());
618                   ackPackage
```

```
619                         .setSourceAckListeningIP(this.requestee
620                             .getSocketAddress());
621                     ackPackage
622                         .setSourceAckListeningPort(this.requestee
623                             .getSocketPort());
624                     logger.trace(
625                         "Matching package received from {}",
626                         defaultFTProtocolMetadata
627                             .getSourceUuid());
628                     metadata = new DefaultFTProtocolMetadata(
629                         this.requestor.getUid());
630                     ackPackage.setMetadata(metadata);
631                     this.writeAckToCluster(ackPackage);
632                     logger.debug(
633                         "Processed/verified {}/{} ACKs so far...",
634                         numProcessedAcks, numVerifiedAcks);
635                   }
636                   this.storagePackage.setData(decodedPackage
637                       .getData());
638                 }
639               }
640             }
641             if (currentPackageVerified == true) {
642               // verify that the ACK came from the client. In a
643               // production environment we would obviously use
644               // something more sophisticated, but here we just
645               // check that a malicious server did not send an ACK
646               // to commit a package.
647               boolean ackSentFromClient = false;
648               AbstractStoragePackageMetadata md = this.storagePackage
649                   .getMetadata(DefaultFTProtocolMetadata.class);
650               DefaultFTProtocolMetadata firstReceivedMetadata = null;
651               if (md != null
652                   && md instanceof DefaultFTProtocolMetadata) {
653                 firstReceivedMetadata = (DefaultFTProtocolMetadata) md;
654                 DefaultFTProtocolMetadata currentAckMetadata =
                    ↪ (DefaultFTProtocolMetadata) ackPackage
655                     .getMetadata(DefaultFTProtocolMetadata.class);
656                 if (firstReceivedMetadata
657                     .equals(currentAckMetadata)) {
658                   ackSentFromClient = true;
659                 } else {
660                   logger.info(
661                       "Mismatching metadata. Comparing {} to {}.",
662                       firstReceivedMetadata.getSourceUuid(),
663                       ackPackage.getUid());
664                 }
665               }
666               if (ackSentFromClient == true) {
667                 numVerifiedAcks++;
668               } else {
669                 logger.info(
670                     "ACK not verified. Package received from IP: Port
                        ↪ {}:{}.",
671                     ackPackage.getSourceAckListeningIP(),
672                     ackPackage.getSourceAckListeningPort());
673               }
674               logger.debug("Processed {} ACKs so far...",
675                   numVerifiedAcks);
676             } else {
677               logger.debug(
```

```
678                 "Invalid hash/data value detected for package: {} and
                    ↪ HashingScheme {}.",
679                 ackPackage, ackPackage.getHashingScheme());
680           }
681           ackPackage.setData(new byte[] {});
682         } else {
683           logger.warn("Processing a {} not of type {}.",
684               StoragePackage.class.getName(),
685               StoragePackageType.ACK);
686         }
687       }
688       StoragePackage pendingPackage = pendingPackageQueue.peek();
689       if (pendingPackage != null) {
690         numProcessedPackages++;
691         pendingPackage = pendingPackageQueue.poll();
692         logger.trace("Processing {}.", pendingPackage);
693         if (pendingPackage.getSourceAckListeningIP().equals(
694             this.requestee.getPrimaryNode()
695                 .getStorageServerAddress())) {
696           logger.trace(
697               "Received data from primary node {}. Exchanging current
                  ↪ StoragePackage with pending package's data.",
698               pendingPackage.getSourceAckListeningIP());
699           StoragePackage temp = this.storagePackage;
700           this.storagePackage = pendingPackage;
701           pendingPackage = temp;
702           if (!pendingPackage.getSourceAckListeningIP().equals(
703               this.requestee.getPrimaryNode()
704                   .getStorageServerAddress())) {
705             pendingPackageQueue.add(pendingPackage);
706           } else {
707             logger.info(
708                 "Received package has same sourceAckListeningIP as
                    ↪ requestee ({}). Perhaps there is a faulty server, or
                    ↪ the primary node sent data twice.",
709                 pendingPackage.getSourceAckListeningIP());
710           }
711         } else {
712           currentPackageVerified = false;
713           if (!this.storagePackage.getHashingScheme().equals(
714               HashingSchemeType.NONE)) {
715             // if we are using hashing verify data quickly
716             pendingPackage.setData(this.storagePackage.getData());
717             if (pendingPackage.verifyHashValue()) {
718               currentPackageVerified = true;
719             } else {
720               logger.trace("Package {} hash value NOT verified!",
721                   pendingPackage);
722             }
723           } else {
724             // if we are not using hashing we need to verify
725             // the
726             // entire data structure
727             if (Arrays.equals(pendingPackage.getData(),
728                 this.storagePackage.getData())) {
729               currentPackageVerified = true;
730             } else {
731               logger.trace("Package {} data NOT verified!",
732                   pendingPackage);
733             }
734           }
735           if (!this.isPackageProcessed(pendingPackage)
```

```
736             && currentPackageVerified) {
737           logger.trace("Adding processed package {}.",
738               pendingPackage);
739           this.addProcessedPackage(pendingPackage);
740           numMatchingPackages++;
741         } else {
742           // check if a different client sent the current package
743           if (!this.storagePackage.getClientListeningIP().equals(
744               pendingPackage.getClientListeningIP())
745               && this.storagePackage.getClientListeningPort() !=
                  ↪ pendingPackage
746                   .getClientListeningPort()) {
747             logger.info(
748                 "A different client ({}:{}) is trying to write the same
                    ↪ StoragePackage (from {}:{}) which has not been
                    ↪ verified yet.",
749                 pendingPackage.getClientListeningIP(),
750                 pendingPackage.getClientListeningPort(),
751                 this.storagePackage.getClientListeningIP(),
752                 this.storagePackage
753                     .getClientListeningPort());
754             // reply to the new client with the previous
755             // client's request
756             pendingPackage.setData(this.storagePackage
757                 .getData());
758             pendingPackage.setHashingScheme(this.storagePackage
759                 .getHashingScheme());
760             pendingPackage.setHashValue();
761             this.writeAckToCluster(pendingPackage);
762           } else {
763             logger.trace(
764                 "Package {} has been processed and client IPs/ports are
                    ↪ the same for package handlers current package and
                    ↪ current processed package: {}:{}",
765                 pendingPackage,
766                 pendingPackage.getClientListeningIP(),
767                 pendingPackage.getClientListeningPort());
768           }
769         }
770       }
771       logger.debug(
772           "{} packages remaining to be processed. {} processed ({}
              ↪ matching) packages processed so far)...Just processed {}.",
773           pendingPackageQueue.size(), numProcessedPackages,
774           numMatchingPackages, pendingPackage);
775     }
776     // need >= 2f matching packages or an ACK from the client
777     if (numVerifiedAcks >= 1) {
778       if (this.requestor.isPackageSpeculative(this.storagePackage
779           .getUid())
780           || this.requestee
781               .isPackageSpeculative(this.storagePackage
782                   .getUid())) {
783         logger.info("Marking {} as not speculative.",
784             this.storagePackage);
785       } else {
786         // package is either not stored or marked as non-speculative
787         logger.debug(
788             "Package is already marked as non-speculative
                ↪ (requestor/requestee: {}/{}) or not stored
                ↪ (requestor/requestee: {}/{}).",
789             this.requestor
```

```
790                 .isPackageSpeculative(this.storagePackage
791                     .getUid()), this.requestee
792                 .isPackageSpeculative(this.storagePackage
793                     .getUid()), this.requestor
794                 .isPackagedStored(this.storagePackage),
795             this.requestee
796                 .isPackagedStored(this.storagePackage));
797         logger.info(
798             "Commit received from client before having verified {}
                ↪ packages. Marking {} as not speculative and storing.",
799             (this.numNodes - 1 - this.numMaxFaults),
800             this.storagePackage);
801       }
802       this.storagePackage.setSpeculative(false);
803       this.closeServerSocket();
804       this.storePackage(this.storagePackage);
805       packageWasStored = true;
806       this.verified = true;
807     }
808     // 3f + 1 - 1 - f = 2f
809     if ((numMatchingPackages >= (this.numNodes - 1 - this.numMaxFaults))
810         && !packageWasStored) {
811       logger.info("Received {} matching packages.",
812           numMatchingPackages);
813       this.storePackage(this.storagePackage);
814       this.writeAckToCluster(pendingPackage);
815       // we don't want the package to be stored multiple times in the
816       // same loop
817       packageWasStored = true;
818     }
819     if (this.requestee.isPackagedStored(this.storagePackage)
820         && this.requestor.isPackagedStored(this.storagePackage)
821         && !this.requestee.isPackageSpeculative(this.storagePackage
822             .getUid())
823         && !this.requestor.isPackageSpeculative(this.storagePackage
824             .getUid()) && this.isAckPackageQueueEmpty()
825         && this.isPendingPackageQueueEmpty()) {
826       // check if the queues are empty because we could have added a
827       // new package in the meantime
828       logger.debug("Exiting early because package is already stored and
          ↪ verified.");
829       break;
830     }
831   }
832 }
833
834 @Override
835 public void writeAckToCluster(StoragePackage storagePackage) {
836   StoragePackage ackPackage = storagePackage.createAckPackage();
837   if (!ackPackage.getHashingScheme().equals(HashingSchemeType.NONE)) {
838     ackPackage.setData(new byte[] {});
839   }
840   logger.debug(
841       "Sending {} for StoragePackage to StorageCluster from {} ({}:{}).",
842       ackPackage.getPackageType(), this.requestor.getUid(),
843       this.requestor.getSocketAddress(),
844       this.requestor.getSocketPort());
845   ackPackage.setSourceAckListeningIP(this.requestor.getSocketAddress());
846   ackPackage.setSourceAckListeningPort(this.requestor.getSocketPort());
847   DefaultFTProtocolMetadata md = new DefaultFTProtocolMetadata(
848       this.requestor.getUid());
849   ackPackage.setMetadata(md);
```

```
850     this.writeToTcp(ackPackage, ackPackage.getClientListeningIP(),
851         ackPackage.getClientListeningPort());
852   }
853
854   @Override
855   public void writeAckToNode(StoragePackage storagePackage) {
856     StoragePackage ackPackage = storagePackage.createAckPackage();
857     if (!ackPackage.getHashingScheme().equals(HashingSchemeType.NONE)) {
858       ackPackage.setData(new byte[] {});
859     }
860     DefaultFTProtocolMetadata md = new DefaultFTProtocolMetadata(
861         this.requestor.getUid());
862     ackPackage.setMetadata(md);
863     InetAddress responseListeningAddress = ackPackage
864         .getSourceAckListeningIP();
865     int responseListeningPort = ackPackage.getSourceAckListeningPort();
866     logger.debug("Sending {} for StoragePackage from {} to {}:{}.",
867         ackPackage.getPackageType(), this.requestor.getUid(),
868         ackPackage.getSourceAckListeningIP(),
869         ackPackage.getSourceAckListeningPort());
870     this.writeToTcp(ackPackage, responseListeningAddress,
871         responseListeningPort);
872   }
873
874   /**
875    * Client side
876    */
877   @Override
878   public void writeToCluster(StorageClient requestor) {
879     logger.debug("Received {} from {}:{} and writing to {}...",
880         this.storagePackage,
881         this.storagePackage.getSourceAckListeningIP(),
882         this.storagePackage.getSourceAckListeningPort(), this.requestee);
883     if (this.storagePackage.getPackageType() != StoragePackageType.ACK) {
884       // we cannot store ACK packages
885       // we want the response to go to the StorageCluster instance on the
886       // client side
887       this.storagePackage.setSourceAckListeningIP(this.requestee
888           .getSocketAddress());
889       this.storagePackage.setSourceAckListeningPort(this.requestee
890           .getSocketPort());
891       this.storagePackage.setClientListeningIP(this.requestee
892           .getSocketAddress());
893       this.storagePackage.setClientListeningPort(this.requestee
894           .getSocketPort());
895       DefaultFTProtocolMetadata md = new DefaultFTProtocolMetadata(
896           this.requestor.getUid());
897       this.storagePackage.setMetadata(md);
898       this.writeToTcp(this.storagePackage, this.requestee
899           .getPrimaryNode().getStorageServerAddress(), this.requestee
900           .getPrimaryNode().getStorageServerPort());
901     } else if (this.storagePackage.getPackageType() ==
        ↪ StoragePackageType.ACK) {
902       if (!this.requestee.isPackagedStored(this.storagePackage)) {
903         logger.debug("Package has not yet been stored.");
904       } else {
905         logger.debug(
906             "Exchanging this StoragePackage with requestor's ({}'s) stored
                ↪ package.",
907             this.requestor.getUid());
908         this.addPendingPackage(this.storagePackage);
909         this.storagePackage = this.requestor
```

```
910             .getStoredPackage(this.storagePackage);
911       }
912       // We received an ACK from the client confirming that
913       // the package can be stored as non-speculative. We need the already
914       // stored StoragePackage to compare this one to in order to check
915       // whether the stored data matches the ACK from the client.
916       // we added this.verified=true because we run into a
917       // loop until timeout if the APH has been restarted.
918     } else {
919       // have this here just in case we add/change the cases above
920       logger.warn(
921           "Handling of {} of type {} has not been implemented yet.",
922           this.storagePackage.getClass(),
923           this.storagePackage.getPackageType());
924     }
925   }
926 
927   /**
928    * Server side
929    *
930    * @throws ProtocolException
931    */
932   @Override
933   public void writeToNode(StorageServer requestor) throws
      ↪ ProtocolException {
934     logger.debug("Received {} on {} and writing to {}...",
935         this.storagePackage, requestor.getUid(), this.requestee);
936     boolean writeData = false;
937     if (!this.isPackageProcessed(this.storagePackage)) {
938       DefaultFTProtocolMetadata clientMetaData =
        ↪ (DefaultFTProtocolMetadata) this.storagePackage
939           .getMetadata(DefaultFTProtocolMetadata.class);
940       if (this.storagePackage.getPackageType() != StoragePackageType.ACK) {
941         if (this.requestee.isPackagedStored(this.storagePackage)) {
942           StoragePackage storedPackage = this.requestee
943               .getStoredPackage(this.storagePackage);
944           if (storedPackage.getTimeStamp().compareTo(
945               this.storagePackage.getTimeStamp()) < 0) {
946             // this.storagePackage.getTimeStamp() is after
947             // storedPackage.getTimeStamp()
948             logger.trace(
949                 "{} overwriting {} sent in a previous round, at time {}.",
950                 this.storagePackage, storedPackage,
951                 storedPackage.getTimeStamp());
952             if (storedPackage.isSpeculative()) {
953               // we are trying to overwrite a package that has not
954               // yet been verified by the client
955               if (!this.storagePackage.getClientListeningIP()
956                   .equals(storedPackage
957                       .getClientListeningIP())
958                   && this.storagePackage
959                       .getClientListeningPort() != storedPackage
960                       .getClientListeningPort()) {
961                 logger.info("A different client is trying to write the
                  ↪ same StoragePackage which has not been verified yet.");
962                 // reply to the new client with the previous
963                 // client's request
964                 this.storagePackage.setData(storedPackage
965                     .getData());
966                 this.storagePackage
967                     .setHashingScheme(storedPackage
968                         .getHashingScheme());
```

```
969                 this.storagePackage.setHashValue();
970                 this.writeAckToCluster(this.storagePackage);
971               } else {
972                 logger.trace("The original client is overwriting the
                    ↪ previous data.");
973                 writeData = true;
974               }
975             }
976           } else {
977             logger.trace(
978                 "Package handler was restarted for stored package {}.",
979                 storedPackage);
980             this.verified = true;
981             // the package handler has stopped running because the
982             // package has been verified or because it timed out
983           }
984         } else {
985           writeData = true;
986         }
987         if (writeData) {
988           int clusterSize = this.requestee.getNodeList().size();
989           if (requestor.getStorageNode().equals(
990               this.requestee.getPrimaryNode())) {
991             logger.trace(
992                 "Writing to primary node with client metadata {}.",
993                 clientMetaData.getSourceUuid());
994             this.storagePackage.setMetadata(clientMetaData);
995             // if we are currently on the primary node in the
996             // cluster then encode the package
997             List<StoragePackage> codedStoragePackages = this.storagePackage
998                 .encode(clusterSize);
999             if (codedStoragePackages.size() != clusterSize) {
1000              // this should not happen if the encoder is correct
1001              throw new FTProtocolException(
1002                  "Unexpected runtime state in "
1003                      + this.getClass().getName()
1004                      + ": codedStoragePackages size ("
1005                      + codedStoragePackages.size()
1006                      + ") not equal expected value of cluster size ("
1007                      + clusterSize
1008                      + "). Check validity of "
1009                      + this.storagePackage
1010                          .getCodingScheme()
1011                      + " "
1012                      + this.storagePackage
1013                          .getCodingScheme()
1014                          .getClass().getName()
1015                      + " implementing encoder.");
1016            }
1017            for (int i = 0; i < clusterSize; i++) {
1018              StorageNode storageNode = this.requestee
1019                  .getNodeList().get(i);
1020              StoragePackage codedPackage = codedStoragePackages
1021                  .get(i);
1022              if (storageNode.equals(requestor.getStorageNode())) {
1023                storageNode.write(requestor, codedPackage);
1024              } else {
1025                codedPackage.setSourceAckListeningIP(requestor
1026                    .getSocketAddress());
1027                codedPackage
1028                    .setSourceAckListeningPort(requestor
1029                        .getSocketPort());
```

```
1030                this.writeToTcpAsync(codedPackage,
1031                    storageNode.getStorageServerAddress(),
1032                    storageNode.getStorageServerPort());
1033              }
1034            }
1035          } else {
1036            logger.trace("Writing to secondary node.");
1037            InetAddress originalSourceAckListeningIP = this.storagePackage
1038                .getSourceAckListeningIP();
1039            int originalSourceAckListeningPort = this.storagePackage
1040                .getSourceAckListeningPort();
1041            for (StorageNode storageNode : this.requestee
1042                .getNodeList()) {
1043              if (storageNode.equals(requestor.getStorageNode())) {
1044                this.storagePackage.setMetadata(clientMetaData);
1045                this.storagePackage
1046                    .setSourceAckListeningIP(originalSourceAckListeningIP);
1047                this.storagePackage
1048                    .setSourceAckListeningPort(originalSourceAckListeningPort);
1049                storageNode.write(requestor,
1050                    this.storagePackage);
1051              } else {
1052                DefaultFTProtocolMetadata md = new
                 ↪ DefaultFTProtocolMetadata(
1053                    this.requestor.getUid());
1054                this.storagePackage.setMetadata(md);
1055                this.storagePackage
1056                    .setSourceAckListeningIP(requestor
1057                        .getSocketAddress());
1058                this.storagePackage
1059                    .setSourceAckListeningPort(requestor
1060                        .getSocketPort());
1061                this.writeToTcp(this.storagePackage,
1062                    storageNode.getStorageServerAddress(),
1063                    storageNode.getStorageServerPort());
1064              }
1065            }
1066            this.storagePackage.setMetadata(clientMetaData);
1067            this.storagePackage
1068                .setSourceAckListeningIP(originalSourceAckListeningIP);
1069            this.storagePackage
1070                .setSourceAckListeningPort(originalSourceAckListeningPort);
1071          }
1072        }
1073      }
1074    } else {
1075      logger.trace("Package has already been processed.");
1076    }
1077    if (this.requestee.isPackagedStored(this.storagePackage)
1078        && this.storagePackage.getPackageType() == StoragePackageType.ACK)
        ↪ {
1079      logger.debug("We received {} that has already been saved",
1080          this.storagePackage);
1081      this.addPendingPackage(this.storagePackage);
1082      logger.debug(
1083          "Exchanging this StoragePackage with requestee's ({}'s) stored
          ↪ package",
1084          this.requestee.getUid());
1085      this.storagePackage = this.requestee
1086          .getStoredPackage(this.storagePackage);
1087    }
1088  }
```

```
1089
1090   @Override
1091   protected boolean timerExpired() {
1092     if (((new Date()).getTime() - this.timeStarted) > timeoutValue) {
1093       logger.info("Timer expired in {}.", this.getClass().getName());
1094       return true;
1095     }
1096     return false;
1097   }
1098 }
```

Listing A.24: shared.ftProtocol.DefaultFTProtocol

```
1 package edu.illinois.ece.ftstorage.shared.ftProtocol;
2
3 import
  ↪ edu.illinois.ece.ftstorage.shared.storagePackage.AbstractStoragePackageMetadata;
4
5 public class DefaultFTProtocolMetadata extends
  ↪ AbstractStoragePackageMetadata {
6
7   private static final long serialVersionUID = -5367532946718649058L;
8   private String sourceUuid;
9
10   public DefaultFTProtocolMetadata(String sourceUuid) {
11     this.sourceUuid = sourceUuid;
12   }
13
14   public String getSourceUuid() {
15     return this.sourceUuid;
16   }
17
18   public void setSourceUuid(String sourceUuid) {
19     this.sourceUuid = sourceUuid;
20   }
21
22   @Override
23   public boolean equals(Object obj) {
24     if (obj != null && obj instanceof DefaultFTProtocolMetadata) {
25       DefaultFTProtocolMetadata other = (DefaultFTProtocolMetadata) obj;
26       if (!other.getSourceUuid().equals(this.getSourceUuid())) {
27         return false;
28       }
29     } else {
30       return false;
31     }
32     return true;
33   }
34
35   @Override
36   protected DefaultFTProtocolMetadata clone() {
37     DefaultFTProtocolMetadata clone = new DefaultFTProtocolMetadata(
38         this.sourceUuid);
39     return clone;
40   }
41 }
```

Listing A.25: shared.ftProtocol.DefaultFTProtocolMetadata

```
1 package edu.illinois.ece.ftstorage.shared.ftProtocol;
2
```

```
3 import java.io.IOException;
4 import java.lang.reflect.Constructor;
5 import java.lang.reflect.InvocationTargetException;
6 import java.util.HashMap;
7 import java.util.Map;
8
9 import org.apache.logging.log4j.LogManager;
10 import org.apache.logging.log4j.Logger;
11
12 import edu.illinois.ece.ftstorage.client.StorageClient;
13 import edu.illinois.ece.ftstorage.server.StorageServer;
14 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
15 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
16
17 @SuppressWarnings({ "unchecked", "rawtypes" })
18 public class FTProtocolHandlerFactory {
19
20   protected final static Logger logger = LogManager.getLogger(Thread
21       .currentThread().getStackTrace()[1].getClassName());
22   private static FTProtocolHandlerFactory instance = new
     ↪ FTProtocolHandlerFactory();
23
24   /**
25    * Retrieve an initialized instance of an {@code
      ↪ AbstractFTProtocolHandler}
26    * for writing on the client-side.
27    *
28    * @param requestor
29    *            The {@code AbstractStorageEntity} requesting a package to
      ↪ be
30    *            written.
31    * @param requestee
32    *            The {@code AbstractStorageEntity} requested to write a
33    *            package.
34    * @param storagePackage
35    *            The package to be written.
36    * @param ftProtocol
37    *            The {@code FTProtocolType} of the
38    *            {@code AbstractFTProtocolHandler}.
39    * @return Initialized instance of an {@code AbstractFTProtocolHandler}
      ↪ for
40    *         writing on the client-side.
41    * @throws IOException
42    */
43   public static AbstractFTProtocolHandler get(final StorageClient
     ↪ requestor,
44       final StorageCluster requestee,
45       final StoragePackage storagePackage, final FTProtocolType ftProtocol)
46       throws IOException {
47     Class[] constructorArgs = new Class[] { StorageClient.class,
48         StorageCluster.class, StoragePackage.class };
49     Constructor<AbstractFTProtocolHandler> ctor;
50     try {
51       ctor = getInstance().ftProtocolHandlers.get(ftProtocol)
52           .getConstructor(constructorArgs);
53       return ctor.newInstance(requestor, requestee, storagePackage);
54     } catch (SecurityException | InstantiationException
55         | IllegalAccessException | IllegalArgumentException
56         | InvocationTargetException | NoSuchMethodException e) {
57       logger.error(
58           "Class constructor({}) not defined or not visible for {}. Using
            ↪ default class {}. Message was: {}.",
```

```
59          constructorArgs, ftProtocol, DefaultFTProtocol.class,
60          e.getMessage());
61    }
62    return new DefaultFTProtocol(requestor, requestee, storagePackage);
63  }
64
65  /**
66   * Retrieve an initialized instance of an {@code
   ↪ AbstractFTProtocolHandler}
67   * for writing on the server-side.
68   *
69   * @param requestor
70   *            The {@code AbstractStorageEntity} requesting a package to
   ↪ be
71   *            written.
72   * @param requestee
73   *            The {@code AbstractStorageEntity} requested to write a
74   *            package.
75   * @param storagePackage
76   *            The package to be written.
77   * @param ftProtocol
78   *            The {@code FTProtocolType} of the
79   *            {@code AbstractFTProtocolHandler}.
80   * @return Initialized instance of an {@code AbstractFTProtocolHandler}
   ↪ for
81   *         writing on the client-side.
82   * @throws IOException
83   */
84  public static AbstractFTProtocolHandler get(final StorageServer
  ↪ requestor,
85      final StorageCluster requestee,
86      final StoragePackage storagePackage, final FTProtocolType ftProtocol)
87      throws IOException {
88    Class[] constructorArgs = new Class[] { StorageServer.class,
89        StorageCluster.class, StoragePackage.class };
90    Constructor<AbstractFTProtocolHandler> ctor;
91    try {
92      ctor = getInstance().ftProtocolHandlers.get(ftProtocol)
93          .getConstructor(constructorArgs);
94      return ctor.newInstance(requestor, requestee, storagePackage);
95    } catch (SecurityException | InstantiationException
96        | IllegalAccessException | IllegalArgumentException
97        | InvocationTargetException | NoSuchMethodException e) {
98      logger.error(
99          "Class constructor({}) not defined or not visible for {}. Using
          ↪ default class {}. Message was: {}.",
100         constructorArgs, ftProtocol, DefaultFTProtocol.class,
101         e.getMessage());
102   }
103   return new DefaultFTProtocol(requestor, requestee, storagePackage);
104 }
105
106 /**
107  *
108  * @return Initialized instance of {@code FTProtocolHandlerFactory}.
109  */
110 public static FTProtocolHandlerFactory getInstance() {
111   if (instance == null) {
112     instance = new FTProtocolHandlerFactory();
113   }
114   return instance;
115 }
```

```
116
117   private Map<FTProtocolType, Class> ftProtocolHandlers;
118
119   private FTProtocolHandlerFactory() {
120     if (instance == null) {
121       this.ftProtocolHandlers = new HashMap<FTProtocolType, Class>(
122           FTProtocolType.values().length);
123       for (FTProtocolType type : FTProtocolType.values()) {
124         Class clazz;
125         String className = this.getClass().getPackage().getName() + "."
126             + type;
127         try {
128           if (!type.equals(FTProtocolType.NONE)) {
129             clazz = Class.forName(className);
130             this.ftProtocolHandlers.put(type, clazz);
131           }
132         } catch (SecurityException | ClassNotFoundException e) {
133           logger.error(
134               "Class constructor not defined or not visible for {}. Using
                ↪ default class {}. Message was: {}.",
135               className, DefaultFTProtocol.class, e.getMessage());
136           clazz = DefaultFTProtocol.class;
137         }
138       }
139     }
140   }
141 }
```

Listing A.26: shared.ftProtocol.FTProtocolHandlerFactory

```
1 package edu.illinois.ece.ftstorage.shared.ftProtocol;
2
3 public enum FTProtocolType {
4   NONE, DefaultFTProtocol, NCBA;
5 }
```

Listing A.27: shared.ftProtocol.FTProtocolType

```
1 package edu.illinois.ece.ftstorage.shared.storageCluster;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7 import java.util.LinkedList;
8 import java.util.List;
9
10 import org.apache.commons.configuration.ConfigurationException;
11 import org.apache.logging.log4j.LogManager;
12 import org.apache.logging.log4j.Logger;
13
14 import edu.illinois.ece.ftstorage.client.StorageClient;
15 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
16 import edu.illinois.ece.ftstorage.shared.AbstractStorageEntity;
17 import edu.illinois.ece.ftstorage.shared.exceptions.ProtocolException;
18 import
   ↪ edu.illinois.ece.ftstorage.shared.ftProtocol.AbstractFTProtocolHandler;
19 import
   ↪ edu.illinois.ece.ftstorage.shared.ftProtocol.FTProtocolHandlerFactory;
20 import edu.illinois.ece.ftstorage.shared.ftProtocol.FTProtocolType;
21 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
```

```
22 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
23 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
24
25 public class StorageCluster extends AbstractStorageEntity {
26
27   /**
28    *
29    */
30   private static final long serialVersionUID = 9187532932014406709L;
31
32   protected final static Logger logger = LogManager.getLogger(Thread
33       .currentThread().getStackTrace()[1].getClassName());
34
35   final List<StorageNode> nodeList;
36   final FTProtocolType ftProtocol;
37   StorageNode primaryNode;
38   StorageClient requestingClient;
39
40   public StorageCluster(String uid, String name, String description)
41       throws ConfigurationException, IOException {
42     this(uid, name, description, FTProtocolType.NONE);
43   }
44
45   public StorageCluster(String uid, String name, String description,
46       FTProtocolType ftProtocol) throws ConfigurationException,
47       IOException {
48     super(uid, name, description);
49     this.nodeList = new LinkedList<StorageNode>();
50     this.ftProtocol = ftProtocol;
51     if (!this.serverSocket.isBound()) {
52       this.serverSocket = new ServerSocket(0);
53     }
54   }
55
56   public void addNode(StorageNode node) {
57     this.nodeList.add(node);
58   }
59
60   public void deleteNode(StorageNode node) {
61     this.nodeList.remove(node);
62   }
63
64   public FTProtocolType getFtProtocol() {
65     return this.ftProtocol;
66   }
67
68   public List<StorageNode> getNodeList() {
69     return this.nodeList;
70   }
71
72   public StorageNode getPrimaryNode() {
73     if (this.primaryNode == null) {
74       this.primaryNode = this.nodeList.get(0);
75     }
76     return this.primaryNode;
77   }
78
79   public boolean isFTCluster() {
80     return !this.ftProtocol.equals(FTProtocolType.NONE);
81   }
82
83   private Object listen() throws IOException, ClassNotFoundException {
```

```
84      Socket socket = this.serverSocket.accept();
85      ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
86      Object receivedObject = in.readObject();
87      logger.trace("Received object at StorageCluster ({}) {}.",
88          this.serverSocket.getLocalSocketAddress(),
89          receivedObject.toString());
90      in.close();
91      socket.close();
92      return receivedObject;
93    }
94
95    @Override
96    public void read(AbstractStorageEntity requestor,
97        StoragePackage storagePackage) throws ProtocolException {
98      logger.trace("Reading from {} with FTProtocol {}.", this.getUid(),
99          this.ftProtocol);
100     if (this.isPackagedStored(storagePackage)) {
101       if (!storagePackage.getPackageType()
102           .equals(StoragePackageType.READ)) {
103         logger.info("{} is stored on {}.", storagePackage,
104             this.getUid());
105       } else {
106         logger.debug("Calling read method with {}.", storagePackage);
107       }
108     } else {
109       logger.info("Package is not stored on {}.", this.getUid());
110     }
111     synchronized (this.runningPackageHandlers) {
112       logger.trace("Got lock on {} runningPackageHandlers.",
113           this.getUid());
114       synchronized (this.syncObject) {
115         logger.trace("Got lock on {} syncObject.", this.getUid());
116         if (this.runningPackageHandlers.containsKey(storagePackage
117             .getUid())) {
118           logger.trace(
119               "{} package handler(s) already running on {} for {} .",
120               this.runningPackageHandlers.get(
121                   storagePackage.getUid()).size(),
122               this.getUid(), storagePackage.getUid());
123           for (AbstractPackageHandler packageHandler :
              ↪ this.runningPackageHandlers
124               .get(storagePackage.getUid())) {
125             packageHandler.addPendingPackage(storagePackage);
126           }
127         } else {
128           Thread t;
129           if (this.isFTCluster()) {
130             try {
131               if (this.requestingClient != null) {
132                 requestor = this.requestingClient;
133               }
134               if (!(requestor instanceof StorageClient)) {
135                 throw new ProtocolException("Cannot read from "
136                     + StorageCluster.class.getName()
137                     + " from "
138                     + requestor.getClass().getName()
139                     + ". Expected "
140                     + StorageClient.class.getName() + ".");
141               }
142               StorageClient clientRequestor = (StorageClient) requestor;
143               this.requestingClient = clientRequestor;
```

```
144               AbstractFTProtocolHandler ftProtocolHandler =
                  ↪ FTProtocolHandlerFactory
145                   .get(clientRequestor, this, storagePackage,
146                       this.getFtProtocol());
147               this.addRunningPackageHandler(ftProtocolHandler,
148                   storagePackage.getUid());
149               t = new Thread(ftProtocolHandler);
150               logger.debug("Starting new {} for {} on {} in {}",
151                   ftProtocolHandler.getClass().getName(),
152                   storagePackage, requestor.getUid(),
153                   t.getName());
154               t.start();
155             } catch (IOException e) {
156               logger.error("Caught {}. Error was: {}",
157                   e.getClass(), e.getMessage());
158             }
159           } else {
160             for (StorageNode clusterPeer : this.getNodeList()) {
161               try {
162                 StorageClusterPackageHandler storageClusterPackageHandler
                    ↪ = new StorageClusterPackageHandler(
163                     this, clusterPeer, storagePackage);
164                 this.addRunningPackageHandler(
165                     storageClusterPackageHandler,
166                     storagePackage.getUid());
167                 t = new Thread(storageClusterPackageHandler);
168                 logger.debug(
169                     "Starting new {} for {} on {} in {}",
170                     storageClusterPackageHandler.getClass()
171                         .getName(), storagePackage,
172                     requestor.getUid(), t.getName());
173                 t.start();
174               } catch (IOException e) {
175                 logger.error("Caught {}. Error was: {}",
176                     e.getClass(), e.getMessage());
177               }
178             }
179           }
180         }
181       }
182     }
183   }
184
185   @Override
186   public void run() {
187     boolean run = false;
188     logger.trace("Attempting to get lock on {} syncObject.",
        ↪ this.getUid());
189     synchronized (this.syncObject) {
190       logger.trace("Got lock on {} syncObject.", this.getUid());
191       if (!this.isRunning()) {
192         this.isRunning = true;
193         run = true;
194         logger.trace("Starting {} {}.", this.getClass().getName(),
195             this.getUid());
196       } else {
197         run = false;
198         logger.trace("{} {} already running.", this.getClass()
199             .getName(), this.getUid());
200       }
201     }
202     logger.trace("Released lock on {} syncObject.", this.getUid());
```

```
203     if (run) {
204       while (true) {
205         try {
206           logger.trace("Listening on {}:{}",
207               this.serverSocket.getInetAddress(),
208               this.serverSocket.getLocalPort());
209           Object receivedObject = this.listen();
210           if (receivedObject instanceof StoragePackage) {
211             StoragePackage storagePackage = (StoragePackage)
                ↪ receivedObject;
212             logger.trace("Received {} on {} ({}).", storagePackage,
213                 this.getUid(),
214                 this.serverSocket.getLocalSocketAddress());
215             if (storagePackage.getPackageType() !=
                ↪ StoragePackageType.READ) {
216               this.write(this, storagePackage);
217             } else {
218               this.read(this, storagePackage);
219             }
220           } else {
221             logger.warn(
222                 "Handling of {} has not been implemented yet.",
223                 receivedObject.getClass());
224           }
225         } catch (IOException | ClassNotFoundException
226             | ProtocolException e) {
227           logger.fatal("Caught {}. Error was: {}", e.getClass(),
228               e.getMessage());
229         }
230       }
231     }
232   }
233
234   public void setPrimaryNode(StorageNode primaryNode) {
235     this.primaryNode = primaryNode;
236   }
237
238   @Override
239   public String toString() {
240     return "StorageCluster [uid=" + this.uid + ", name=" + this.name
241         + ", description=" + this.description + ", nodeList.size()="
242         + this.nodeList.size() + "]";
243   }
244
245   @Override
246   public void write(AbstractStorageEntity requestor,
247       StoragePackage storagePackage) throws ProtocolException {
248     logger.trace("Writing to {} with FTProtocol {}.", this.getUid(),
249         this.ftProtocol);
250     if (this.isPackagedStored(storagePackage)) {
251       if (storagePackage.getPackageType() != StoragePackageType.ACK) {
252         logger.info("{} is stored on {}.", storagePackage,
253             this.getUid());
254       } else {
255         logger.debug("Calling write method with {}.", storagePackage);
256       }
257     } else {
258       logger.debug("Package is not stored on {}.", this.getUid());
259     }
260     synchronized (this.runningPackageHandlers) {
261       logger.trace("Got lock on {} runningPackageHandlers.",
262           this.getUid());
```

```
263       synchronized (this.syncObject) {
264         logger.trace("Got lock on {} syncObject.", this.getUid());
265         if (this.runningPackageHandlers.containsKey(storagePackage
266             .getUid())) {
267           logger.trace(
268               "{} package handler(s) already running on {} for {} .",
269               this.runningPackageHandlers.get(
270                   storagePackage.getUid()).size(),
271               this.getUid(), storagePackage.getUid());
272           for (AbstractPackageHandler packageHandler :
              ↪ this.runningPackageHandlers
273               .get(storagePackage.getUid())) {
274             packageHandler.addPendingPackage(storagePackage);
275           }
276         } else {
277           Thread t;
278           if (this.isFTCluster()) {
279             try {
280               if (this.requestingClient != null) {
281                 requestor = this.requestingClient;
282               }
283               if (!(requestor instanceof StorageClient)) {
284                 throw new ProtocolException("Cannot write to "
285                     + StorageCluster.class.getName()
286                     + " from "
287                     + requestor.getClass().getName()
288                     + ". Expected "
289                     + StorageClient.class.getName() + ".");
290               }
291               StorageClient clientRequestor = (StorageClient) requestor;
292               this.requestingClient = clientRequestor;
293               AbstractFTProtocolHandler ftProtocolHandler =
                  ↪ FTProtocolHandlerFactory
294                   .get(clientRequestor, this, storagePackage,
295                       this.getFtProtocol());
296               this.addRunningPackageHandler(ftProtocolHandler,
297                   storagePackage.getUid());
298               t = new Thread(ftProtocolHandler);
299               logger.debug("Starting new {} for {} on {} in {}",
300                   ftProtocolHandler.getClass().getName(),
301                   storagePackage, requestor.getUid(),
302                   t.getName());
303               t.start();
304             } catch (IOException e) {
305               logger.error("Caught {}. Error was: {}",
306                   e.getClass(), e.getMessage());
307             }
308           } else {
309             int clusterSize = this.getNodeList().size();
310             List<StoragePackage> codedStoragePackages = storagePackage
311                 .encode(clusterSize);
312             if (codedStoragePackages.size() != clusterSize) {
313               // this should not happen if the encoder is correct
314               throw new ProtocolException(
315                   "Unexpected runtime state in "
316                       + this.getClass().getName()
317                       + ": codedStoragePackages size ("
318                       + codedStoragePackages.size()
319                       + ") not equal expected value of cluster size ("
320                       + clusterSize
321                       + "). Check validity of "
322                       + storagePackage.getCodingScheme()
```

```
323                        + " "
324                        + storagePackage.getCodingScheme()
325                            .getClass().getName()
326                        + " implementing encoder.");
327              }
328              for (int i = 0; i < clusterSize; i++) {
329                StorageNode storageNode = this.getNodeList().get(i);
330                StoragePackage codedPackage = codedStoragePackages
331                    .get(i);
332                try {
333                  StorageClusterPackageHandler storageClusterPackageHandler
                   ↪ = new StorageClusterPackageHandler(
334                      this, storageNode, codedPackage);
335                  this.addRunningPackageHandler(
336                      storageClusterPackageHandler,
337                      codedPackage.getUid());
338                  t = new Thread(storageClusterPackageHandler);
339                  logger.debug(
340                      "Starting new {} for {} on {} in {}",
341                      storageClusterPackageHandler.getClass()
342                          .getName(), storagePackage,
343                      requestor.getUid(), t.getName());
344                  t.start();
345                } catch (IOException e) {
346                  logger.error("Caught {}. Error was: {}",
347                      e.getClass(), e.getMessage());
348                }
349              }
350            }
351          }
352        }
353        logger.trace("Released lock on {} syncObject.", this.getUid());
354      }
355      logger.trace("Released lock on {} runningPackageHandlers.",
356          this.getUid());
357    }
358  }
```

Listing A.28: shared.storageCluster.StorageCluster

```
1 package edu.illinois.ece.ftstorage.shared.storageCluster;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.net.UnknownHostException;
6 import java.util.HashMap;
7 import java.util.Iterator;
8 import java.util.LinkedList;
9 import java.util.List;
10 import java.util.Map;
11 import java.util.UUID;
12
13 import javax.xml.parsers.DocumentBuilderFactory;
14 import javax.xml.parsers.ParserConfigurationException;
15
16 import org.apache.commons.configuration.ConfigurationException;
17 import org.apache.commons.configuration.HierarchicalConfiguration;
18 import org.apache.commons.configuration.XMLConfiguration;
19 import org.apache.logging.log4j.LogManager;
20 import org.apache.logging.log4j.Logger;
21 import org.w3c.dom.Document;
22 import org.w3c.dom.Node;
```

```
23
24 import
   ↪ edu.illinois.ece.ftstorage.shared.exceptions.StorageClusterConfigurationException;
25 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
26 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNodeFactory;
27 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
28 import edu.illinois.ece.ftstorage.shared.util.FtStorageUtil;
29
30 public class StorageClusterConfigurationParser {
31
32   protected final static Logger logger = LogManager.getLogger(Thread
33       .currentThread().getStackTrace()[1].getClassName());
34
35   private String uid;
36
37   private static Map<String, StorageClusterConfigurationParser> instances;
38
39   private static final String defaultConfigFilePath =
     ↪ "META-INF/clusterConfig.xml";
40
41   private String configFilePath = "";
42
43   private XMLConfiguration config;
44
45   public static StorageClusterConfigurationParser getInstance(String uid)
46       throws ConfigurationException {
47     StorageClusterConfigurationParser instance;
48     if (instances == null) {
49       instances = new HashMap<String, StorageClusterConfigurationParser>();
50       instance = new StorageClusterConfigurationParser();
51     } else {
52       if (instances.containsKey(uid)) {
53         instance = instances.get(uid);
54       } else {
55         instance = new StorageClusterConfigurationParser();
56         instances.put(uid, instance);
57       }
58     }
59     instance.uid = uid;
60     return instance;
61   }
62
63   public static StorageClusterConfigurationParser getInstance()
64       throws ConfigurationException {
65     return StorageClusterConfigurationParser.getInstance("default");
66   }
67
68   private boolean validate(XMLConfiguration configuration)
69       throws StorageClusterConfigurationException {
70     Iterator<String> iter = configuration
71         .getKeys("Clusters.Cluster.Nodes.Node[@uid]");
72     HashMap<String, Integer> uidCounter = new HashMap<String, Integer>();
73     while (iter.hasNext()) {
74       String keyValue = iter.next();
75       for (String uid : configuration.getStringArray(keyValue)) {
76         if (uidCounter.containsKey(uid)) {
77           if (uidCounter.get(uid) > 0) {
78             throw new StorageClusterConfigurationException(
79                 "ERROR: Found duplicate StorageNode UID ("
80                     + uid + ") in config file.");
81           }
82         } else {
```

```
83          uidCounter.put(uid.toString(), 1);
84        }
85      }
86    }
87    iter = configuration.getKeys("Clusters.Cluster[@uid]");
88    uidCounter = new HashMap<String, Integer>();
89    while (iter.hasNext()) {
90      String keyValue = iter.next();
91      for (String uid : configuration.getStringArray(keyValue)) {
92        if (uidCounter.containsKey(uid)) {
93          throw new StorageClusterConfigurationException(
94              "ERROR: Found duplicate StorageCluster UID (" + uid
95                  + ") in config file.");
96        } else {
97          uidCounter.put(uid.toString(), 1);
98        }
99      }
100   }
101   return true;
102 }
103
104 private StorageClusterConfigurationParser() throws
    ↪ ConfigurationException {
105   this.config = this.configure();
106 }
107
108 private XMLConfiguration configure() throws ConfigurationException {
109   File configFile;
110   XMLConfiguration configuration = null;
111   try {
112     if ((configFile = new File(this.configFilePath)).isFile()) {
113       this.configFilePath = configFile.getCanonicalPath();
114       configuration = this.configure(configFile.getCanonicalPath());
115     } else if ((configFile = new File(defaultConfigFilePath)).isFile()) {
116       this.configFilePath = configFile.getCanonicalPath();
117       configuration = this.configure(configFile.getCanonicalPath());
118     } else if ((configFile = new File("src/main/resources/"
119         + defaultConfigFilePath)).isFile()) {
120       this.configFilePath = configFile.getCanonicalPath();
121       // assume we are in the project root directory
122       configuration = this.configure(configFile.getCanonicalPath());
123     } else {
124       this.configFilePath = configFile.getCanonicalPath();
125       throw new StorageClusterConfigurationException(
126           "ERROR: No cluster config file found. Please check your
           ↪ classpath and/or verify the location of the file.");
127     }
128   } catch (IOException e) {
129     throw new StorageClusterConfigurationException(
130         "ERROR: Could not open or read configuration file at "
131             + this.configFilePath, e);
132   }
133   return configuration;
134 }
135
136 private XMLConfiguration configure(String configurationFilePath)
137     throws ConfigurationException {
138   XMLConfiguration configuration = null;
139   configuration = new XMLConfiguration(configurationFilePath);
140   if (this.validate(configuration) == false) {
141     return null;
142   }
```

```
143     return configuration;
144   }
145
146   public List<StorageCluster> getClusters() throws ConfigurationException,
147       IOException {
148     List<StorageCluster> scl = new LinkedList<StorageCluster>();
149     String primaryNodeUid;
150     for (HierarchicalConfiguration cluster : this.config
151         .configurationsAt("Clusters.Cluster")) {
152       primaryNodeUid = cluster.getString("primaryNode");
153       StorageCluster storageCluster = StorageClusterFactory
154           .fromXmlConfiguration(cluster);
155       boolean primaryNodeDefinedAndFound = false;
156       StorageNode node;
157       for (HierarchicalConfiguration clusterNode : cluster
158           .configurationsAt("Nodes.Node")) {
159         try {
160           node = StorageNodeFactory.fromXmlConfigurationAndCluster(
161               clusterNode, storageCluster);
162           if (node.getUid().equals(primaryNodeUid)) {
163             storageCluster.setPrimaryNode(node);
164             primaryNodeDefinedAndFound = true;
165           }
166         } catch (UnknownHostException e) {
167           logger.error(
168               "Error getting InetAddress for StorageNode in config file.
                  ↪ Message was: {}.",
169               e.getMessage());
170         }
171       }
172       if (!primaryNodeDefinedAndFound) {
173         storageCluster.setPrimaryNode(storageCluster.getNodeList().get(
174             0));
175       }
176       scl.add(storageCluster);
177     }
178     return scl;
179   }
180
181   public XMLConfiguration getConfig() throws ConfigurationException {
182     if (instances == null) {
183       getInstance(this.uid);
184     }
185     return this.config;
186   }
187
188   public HashMap<UUID, StoragePackage> getPackages()
189       throws ConfigurationException {
190     // TODO load packages that are stored
191     HashMap<UUID, StoragePackage> sph = new HashMap<UUID,
        ↪ StoragePackage>();
192     // String uid;
193     // String name;
194     // String description;
195     // for (HierarchicalConfiguration xmlPackage : config
196     // .configurationsAt("Clusters.Cluster")) {
197     // name = xmlPackage.getString("[@name]");
198     // uid = xmlPackage.getString("[@uid]");
199     // description = xmlPackage.getString("Description");
200     // StoragePackage storageCluster = new StoragePackage(uid, name,
201     // description);
202     // StoragePackage storagePackage = new StoragePackage(
```

```
203     // InetAddress.getLocalHost(), 9100, xmlPackage);
204     // StoragePackage storagePackage = new
205     // StoragePackage("hello world!");
206     // (InetAddress senderIP, int senderPort, InetAddress targetIP,
207     // int targetPort, UUID id, byte[] data, byte[] hashValue)
208     // for (HierarchicalConfiguration clusterNode : xmlPackage
209     // .configurationsAt("Nodes.Node")) {
210     // StorageNode node = null;
211     // try {
212     // node = StorageNodeFactory.fromXmlConfigurationAndCluster(
213     // clusterNode, storageCluster);
214     // } catch (StorageNodeConfigurationException e) {
215     // throw new ConfigurationException(
216     // "ERROR: Could not create new storage node.", e);
217     // }
218     // if (node != null) {
219     // storageCluster.addNode(node);
220     // }
221     // }
222 
223     // sph.put(storagePackage.getUid(), storagePackage);
224     // }
225     return sph;
226   }
227 
228   private void saveConfigFile() throws ConfigurationException {
229     this.validate(this.config);
230     this.config.save();
231     logger.debug("Successfully saved config file to {}.",
232         this.config.getFile());
233   }
234 
235   private void setConfig(XMLConfiguration config) {
236     try {
237       String configFilePath = this.config.getFileName();
238       this.config.setFileName(configFilePath + ".bak");
239       this.saveConfigFile();
240     } catch (ConfigurationException e) {
241       logger.error("Caught {}. Error was: {}", e.getClass(),
242           e.getMessage());
243       return;
244     }
245     logger.debug("Successfully created backup of old config file at {}.",
246         this.config.getFile());
247     this.config = config;
248     try {
249       this.saveConfigFile();
250     } catch (ConfigurationException e) {
251       logger.error("Caught {}. Error was: {}", e.getClass(),
252           e.getMessage());
253       return;
254     }
255     logger.debug("Successfully saved new config file.");
256     if (!logger.isDebugEnabled()) {
257       logger.info("Successfully created backup and saved new config
          ↪ file.");
258     }
259   }
260 
261   public void setConfigFilePath(String configFilePath) {
262     this.configFilePath = configFilePath;
263   }
```

```
264
265   public void setDocument(Document newDocument,
266       String newDocumentSubfolderPath) throws ConfigurationException,
267       StorageClusterConfigurationException {
268     Node nodeType = newDocument.getFirstChild().getAttributes()
269         .getNamedItem("type");
270     if (nodeType == null) {
271       throw new StorageClusterConfigurationException(
272           "Attribute 'type' is not set for root node in XML configuration.
          ↪ Expecting 'ClusterConfig' as a value.");
273     }
274     if (nodeType.getNodeValue().equals("ClusterConfig")) {
275       logger.debug("Setting new ClusterConfig.");
276       XMLConfiguration newConfig = (XMLConfiguration) this.getConfig()
277           .clone();
278       try {
279         newConfig.setDocumentBuilder(DocumentBuilderFactory
280             .newInstance().newDocumentBuilder());
281         newConfig
282             .getDocumentBuilder()
283             .getDOMImplementation()
284             .createDocument(newDocument.getNamespaceURI(), null,
285                 newDocument.getDoctype())
286             .importNode(newDocument.getFirstChild(), false);
287         newConfig.setFileName(FtStorageUtil
288             .getPathWithNewParentFolderName(this.config.getFile(),
289                 newDocumentSubfolderPath));
290         this.setConfig(newConfig);
291       } catch (ParserConfigurationException e) {
292         logger.error("Caught {}. Error was: {}", e.getClass(),
293             e.getMessage());
294       }
295     } else {
296       String message = String
297           .format("ERROR: Could not set new document in %s. Attribute
          ↪ 'type' has an unrecognized value %s. Please verify the
          ↪ structure of the document, in particular the root node
          ↪ element %s.",
298               this.getClass(), nodeType, newDocument
299                   .getFirstChild().getNodeName());
300       throw new StorageClusterConfigurationException(message);
301     }
302   }
303 }
```

Listing A.29: shared.storageCluster.StorageClusterConfigurationParser

```
1 package edu.illinois.ece.ftstorage.shared.storageCluster;
2
3 import java.io.IOException;
4
5 import org.apache.commons.configuration.ConfigurationException;
6 import org.apache.commons.configuration.HierarchicalConfiguration;
7 import org.apache.logging.log4j.LogManager;
8 import org.apache.logging.log4j.Logger;
9
10 import edu.illinois.ece.ftstorage.shared.ftProtocol.FTProtocolType;
11
12 public class StorageClusterFactory {
13
14   protected final static Logger logger = LogManager.getLogger(Thread
15       .currentThread().getStackTrace()[1].getClassName());
```

```
16
17   public static StorageCluster fromXmlConfiguration(
18       HierarchicalConfiguration clusterElement)
19       throws ConfigurationException, IOException {
20     StorageCluster storageCluster;
21     String clusterUid = clusterElement.getString("[@uid]");
22     String clusterName = clusterElement.getString("[@name]");
23     FTProtocolType clusterFtProtocol = FTProtocolType.NONE;
24     try {
25       clusterFtProtocol = FTProtocolType.valueOf(clusterElement
26           .getString("[@ftprotocol]"));
27     } catch (IllegalArgumentException | NullPointerException e) {
28       logger.warn(
29           "FTProtocol "
30               + clusterElement.getString("[@ftprotocol]")
31               + " not recognized or not defined in config file. Using
                 ↪ default value. Message was: {}.",
32           e.getMessage());
33     }
34     String clusterDescription = clusterElement.getString("Description");
35     storageCluster = new StorageCluster(clusterUid, clusterName,
36         clusterDescription, clusterFtProtocol);
37     return storageCluster;
38   }
39 }
```

Listing A.30: shared.storageCluster.StorageClusterFactory

```
1 package edu.illinois.ece.ftstorage.shared.storageCluster;
2
3 import java.io.IOException;
4 import java.net.InetAddress;
5 import java.util.ArrayList;
6 import java.util.Arrays;
7 import java.util.LinkedList;
8 import java.util.List;
9 import java.util.concurrent.BlockingQueue;
10
11 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
12 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
13 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
14 import edu.illinois.ece.ftstorage.shared.storagePackage.HashingSchemeType;
15 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
16 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackageType;
17
18 public class StorageClusterPackageHandler extends AbstractPackageHandler {
19
20   final StorageCluster requestor;
21   final StorageNode requestee;
22   private boolean verified;
23
24   public StorageClusterPackageHandler(StorageCluster requestor,
25       StorageNode requestee, StoragePackage storagePackage)
26       throws IOException {
27     super(requestor, requestee, storagePackage);
28     this.requestor = requestor;
29     this.requestee = requestee;
30   }
31
32   @Override
33   public void read() {
34     if (!this.requestor.isPackagedStored(this.storagePackage)
```

```
35          && !this.requestee.isPackagedStored(this.storagePackage)) {
36        logger.debug("Received {} on {} and reading from {}...",
37            this.storagePackage, this.requestor.getUid(),
38            this.requestee);
39        this.storagePackage.setClientListeningIP(this.requestor
40            .getSocketAddress());
41        this.storagePackage.setClientListeningPort(this.requestor
42            .getSocketPort());
43        this.writeToTcpAsync(this.storagePackage,
44            this.requestee.getStorageServerAddress(),
45            this.requestee.getStorageServerPort());
46      } else {
47        logger.debug(
48            "{} is already stored locally on requestor {} ({}) or
            ↪ requestestee {} ({}). Not sending remote request.",
49            this.requestor.getUid(),
50            this.requestor.isPackagedStored(this.storagePackage),
51            this.requestee.getUid(),
52            this.requestee.isPackagedStored(this.storagePackage));
53      }
54    }
55
56    @Override
57    public void receiveReadAck() {
58      logger.trace("Waiting for responses from servers...");
59      int numNodes = this.requestor.getNodeList().size();
60      int numMaxFaults = (numNodes - 1) / 3;
61      int numReceivedAcks = 0;
62      List<StoragePackage> codedPackageList = new
        ↪ LinkedList<StoragePackage>();
63      BlockingQueue<StoragePackage> ackPackageQueue;
64      synchronized (this.receivedAckPackages) {
65        ackPackageQueue = this.receivedAckPackages.get(this.storagePackage
66            .getUid());
67      }
68      while (!this.verified && !this.timerExpired()) {
69        StoragePackage pendingPackage = ackPackageQueue.peek();
70        if (pendingPackage != null) {
71          ackPackageQueue.poll();
72          logger.trace("Processing {}.", pendingPackage);
73          byte[] key;
74          int numMatchingPackages = 0;
75          if (this.storagePackage.getCodingScheme() !=
            ↪ CodingSchemeType.NONE) {
76            // if we use some kind of coding scheme, the decoding
77            // happens on the server-side.
78            if (this.storagePackage.getHashingScheme() !=
              ↪ HashingSchemeType.NONE) {
79              key = pendingPackage.getData();
80            } else {
81              pendingPackage.setHashValue();
82              key = pendingPackage.getHashValue();
83            }
84            if (this.pendingReadPackages.containsKey(key)) {
85              numMatchingPackages = this.pendingReadPackages.get(key);
86              numMatchingPackages++;
87            }
88            numReceivedAcks++;
89            this.pendingReadPackages.put(key, numMatchingPackages);
90            if (numMatchingPackages > 0) {
91              logger.trace("Received {} matching read packages.",
92                  numMatchingPackages);
```

```
93            this.storagePackage.setData(pendingPackage.getData());
94            this.storagePackage.setHashValue();
95            this.verified = true;
96          }
97        } else {
98          // we could just use the "if" statement code, but we just
99          // want to show here that the default decoding works
100         codedPackageList.add(pendingPackage);
101         numReceivedAcks++;
102         if (codedPackageList.size() >= numMaxFaults) {
103           // we need to have as many packages to decode as
104           // in the faulty case, but we require one
105           // package less than when we assume that faults
106           // occur
107           StoragePackage decodedPackage;
108           if ((decodedPackage = pendingPackage
109               .decode(codedPackageList)) != null) {
110             logger.trace("Successfully decoded {}.",
111                 decodedPackage);
112             this.storagePackage.setData(decodedPackage
113                 .getData());
114             this.storagePackage.setHashValue();
115             this.verified = true;
116           }
117         }
118       }
119       if (numReceivedAcks > 0 && this.verified) {
120         logger.info("Marking read {} as not speculative.",
121             this.storagePackage);
122         this.storagePackage.setSpeculative(false);
123         break;
124       }
125     }
126   }
127   this.storePackage(this.storagePackage);
128   // this.requestor.storePackage(this.storagePackage);
129   // this.requestee.storePackage(this.storagePackage);
130 }
131
132 @Override
133 public void receiveWriteAck() {
134   logger.trace("Waiting for write ACKs...");
135   int numNodes = this.requestor.getNodeList().size();
136   int numMaxFaults = (numNodes - 1) / 3;
137   int numVerifiedAcks = 0;
138   int numProcessedAcks = 0;
139   ArrayList<StoragePackage> decodingQueue = new
    ↪ ArrayList<StoragePackage>(
140       this.requestor.getNodeList().size());
141   BlockingQueue<StoragePackage> ackPackageQueue = null;
142   StoragePackage ackPackage = null;
143   synchronized (this.receivedAckPackages) {
144     if (this.receivedAckPackages.containsKey(this.storagePackage
145         .getUid())) {
146       ackPackageQueue = this.receivedAckPackages
147           .get(this.storagePackage.getUid());
148     } else {
149       logger.error("ACK package queue is null.");
150     }
151   }
152   while (!this.verified && !this.timerExpired()) {
153     ackPackage = ackPackageQueue.peek();
```

```
154       boolean currentPackageVerified = false;
155       if (ackPackage != null) {
156         ackPackageQueue.poll();
157         logger.debug("{} packages remaining to be processed...",
158             ackPackageQueue.size());
159         if (ackPackage.getPackageType().equals(StoragePackageType.ACK)) {
160           decodingQueue.add(ackPackage);
161           numProcessedAcks++;
162           currentPackageVerified = false;
163           if (this.storagePackage.getHashingScheme() !=
          ↪ HashingSchemeType.NONE) {
164             // if we are using hashing to verify data quickly
165             ackPackage.setData(this.storagePackage.getData());
166             if (ackPackage.verifyHashValue()) {
167               currentPackageVerified = true;
168             }
169           } else {
170             // if we are not using hashing we need to verify the
171             // entire data structure
172             if (this.storagePackage.getCodingScheme() ==
            ↪ CodingSchemeType.NONE) {
173               if (Arrays.equals(ackPackage.getData(),
174                   this.storagePackage.getData())) {
175                 currentPackageVerified = true;
176               }
177             } else {
178               if (decodingQueue.size() >= numMaxFaults) {
179                 // we need to have as many packages to decode as
180                 // in the faulty case, but we require one
181                 // package less than when we assume that faults
182                 // occur
183                 StoragePackage decodedPackage = this.storagePackage
184                     .decode(decodingQueue);
185                 if (decodedPackage != null) {
186                   for (StoragePackage codedPackage : decodingQueue) {
187                     numVerifiedAcks++;
188                     this.writeToTcpAsync(
189                         ackPackage,
190                         codedPackage
191                             .getSourceAckListeningIP(),
192                         codedPackage
193                             .getSourceAckListeningPort());
194                     logger.debug(
195                         "Processed/verified {}/{} ACKs so far...",
196                         numProcessedAcks,
197                         numVerifiedAcks);
198                   }
199                   this.storagePackage.setData(decodedPackage
200                       .getData());
201                 }
202               }
203             }
204           }
205           if (currentPackageVerified == true) {
206             numVerifiedAcks++;
207             InetAddress sourceListeningAddress = ackPackage
208                 .getSourceAckListeningIP();
209             int sourceListeningPort = ackPackage
210                 .getSourceAckListeningPort();
211             logger.trace("Matching package received from {}:{}",
212                 sourceListeningAddress, sourceListeningPort);
213             logger.debug("Processed/verified {}/{} ACKs so far...",
```

```
214                 numProcessedAcks, numVerifiedAcks);
215             if (this.requestee
216                 .isPackagedStored(this.storagePackage)
217                 && !this.requestee
218                     .isPackageSpeculative(this.storagePackage
219                         .getUid())) {
220               StoragePackage storedPackage = this.requestee
221                   .getStoredPackage(this.storagePackage);
222               // The package was already stored and verified. It
223               // is possible that in future executions less than f
224               // ACKs are received and the loop continues until
225               // the timeout occurs.
226               if (storedPackage.getTimeStamp().equals(
227                   ackPackage.getTimeStamp())
228                   && !storedPackage.isSpeculative()) {
229                 this.closeServerSocket();
230                 break;
231               }
232             }
233           } else {
234             logger.warn(
235                 "Invalid hash/data value detected for package: {} and
                 ↪ HashingScheme {}.",
236                 ackPackage, ackPackage.getHashingScheme());
237             if (this.storagePackage.getHashingScheme() ==
             ↪ HashingSchemeType.NONE) {
238               logger.trace("Comparing {} with ACK package {}.",
239                   this.storagePackage.getData(),
240                   ackPackage.getHashValue());
241             } else {
242               logger.trace("Comparing {} with ACK package {}.",
243                   this.storagePackage.getHashValue(),
244                   ackPackage.getHashValue());
245             }
246           }
247         }
248       }
249       if (numVerifiedAcks > 0) {
250         logger.trace("Storing StoragePackage {}.",
251             this.storagePackage.getUid());
252         this.storagePackage.setSpeculative(false);
253         this.storePackage(this.storagePackage);
254         this.verified = true;
255         this.closeServerSocket();
256         break;
257       }
258       if (this.requestee.isPackagedStored(this.storagePackage)
259           && !this.requestee.isPackageSpeculative(this.storagePackage
260               .getUid())
261           && this.requestor.isPackagedStored(this.storagePackage)
262           && !this.requestor.isPackageSpeculative(this.storagePackage
263               .getUid()) && ackPackageQueue.isEmpty()) {
264         logger.trace("Package is stored and not speculative.");
265         this.closeServerSocket();
266         this.verified = true;
267         break;
268       }
269     }
270   }
271
272   @Override
273   public void write() {
```

```
274     if (this.storagePackage.getPackageType() != StoragePackageType.ACK) {
275       this.storagePackage.setSourceAckListeningIP(this.requestor
276           .getSocketAddress());
277       this.storagePackage.setSourceAckListeningPort(this.requestor
278           .getSocketPort());
279       this.storagePackage.setClientListeningIP(this.requestor
280           .getSocketAddress());
281       this.storagePackage.setClientListeningPort(this.requestor
282           .getSocketPort());
283       this.writeToTcp(this.storagePackage,
284           this.requestee.getStorageServerAddress(),
285           this.requestee.getStorageServerPort());
286     } else {
287       if (!this.requestee.isPackagedStored(this.storagePackage)) {
288         logger.debug(
289             "Cannot store {}. Package has not been written to {}.",
290             this.storagePackage, this.requestee);
291       } else {
292         logger.debug(
293             "{} is already stored on {}. Will not wait for ACKs.",
294             this.storagePackage, this.requestee.getUid());
295       }
296       // we received an ACK after it has already been stored
297       this.verified = true;
298     }
299   }
300 }
```

Listing A.31: shared.storageCluster.StorageClusterPackageHandler

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
2
3 import java.net.InetAddress;
4
5 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
6
7 public class DefaultStorageNode extends LocalStorageNode {
8
9   private static final long serialVersionUID = -778819870646977656L;
10
11  public DefaultStorageNode(String uid, String name, String description) {
12    super(uid, name, description);
13  }
14
15  public DefaultStorageNode(String uid, String name, String description,
16      InetAddress storageServerAddress, int storageServerPort) {
17    super(uid, name, description, storageServerAddress, storageServerPort);
18  }
19
20  public DefaultStorageNode(String uid, String name, String description,
21      StorageCluster storageCluster, InetAddress storageServerAddress,
22      int storageServerPort) {
23    super(uid, name, description, storageCluster, storageServerAddress,
24        storageServerPort);
25  }
26
27 }
```

Listing A.32: shared.storageNode.DefaultStorageNode

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
```

```
2
3 import java.io.File;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.net.InetAddress;
7 import java.util.Arrays;
8
9 import org.apache.commons.io.FileUtils;
10
11 import edu.illinois.ece.ftstorage.shared.AbstractStorageEntity;
12 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
13 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
14
15 public class LocalStorageNode extends StorageNode {
16
17   /**
18    *
19    */
20   private static final long serialVersionUID = 5957510865786105666L;
21
22   protected LocalStorageNode(String uid, String name, String description) {
23     super(uid, name, description);
24   }
25
26   public LocalStorageNode(String uid, String name, String description,
27       InetAddress storageServerAddress, int storageServerPort) {
28     super(uid, name, description, storageServerAddress, storageServerPort);
29   }
30
31   public LocalStorageNode(String uid, String name, String description,
32       StorageCluster storageCluster, InetAddress storageServerAddress,
33       int storageServerPort) {
34     super(uid, name, description, storageCluster, storageServerAddress,
35         storageServerPort);
36   }
37
38   @Override
39   public void read(AbstractStorageEntity requestor,
40       StoragePackage storagePackage) {
41     if (this.storedPackages.containsKey(storagePackage.getUid())
42         || requestor.isPackagedStored(storagePackage)) {
43       StringBuilder sb = new StringBuilder(256);
44       sb.append(this.getStorageCluster().getUid());
45       sb.append(File.separatorChar);
46       sb.append(this.getUid());
47       sb.append(File.separatorChar);
48       sb.append(storagePackage.getUid());
49       File dataFile = new File(sb.toString() + File.separatorChar
50           + "data");
51       dataFile.getParentFile().mkdirs();
52       File hashFile = new File(sb.toString() + File.separatorChar
53           + "hash");
54       hashFile.getParentFile().mkdirs();
55       byte[] data = new byte[] {};
56       byte[] hash = new byte[] {};
57       try {
58         if (dataFile.exists()) {
59           data = FileUtils.readFileToByteArray(dataFile);
60         }
61         if (hashFile.exists()) {
62           hash = FileUtils.readFileToByteArray(hashFile);
63         }
```

```
64        } catch (IOException e) {
65          logger.error(
66              "Could not read from local file system. Error was: {}.",
67              e.getMessage());
68        }
69        storagePackage.setData(data);
70        storagePackage.setHashValue();
71        if (hash != null && storagePackage.getHashValue() != null
72            && Arrays.equals(hash, storagePackage.getHashValue())) {
73          logger.info(
74              "Calculated hash value does not equal stored value. Possibly a
              ↪ different hashing scheme is used than before. (Comparing:
              ↪ {}/{})",
75              storagePackage.getHashValue(), hash);
76        }
77      }
78    }
79
80    @Override
81    public void write(AbstractStorageEntity requestor,
82        StoragePackage storagePackage) {
83      logger.trace("{} writing to {}.", requestor.getUid(), this.getUid());
84      synchronized (this.storedPackages) {
85        if (!this.isPackagedStored(storagePackage)) {
86          StringBuilder sb = new StringBuilder(256);
87          sb.append(this.getStorageCluster().getUid());
88          sb.append(File.separatorChar);
89          sb.append(this.getUid());
90          sb.append(File.separatorChar);
91          sb.append(storagePackage.getUid());
92          File dataFile = new File(sb.toString() + File.separatorChar
93              + "data");
94          dataFile.getParentFile().mkdirs();
95          File hashFile = new File(sb.toString() + File.separatorChar
96              + "hash");
97          hashFile.getParentFile().mkdirs();
98          FileOutputStream fos;
99          try {
100           fos = new FileOutputStream(dataFile);
101           fos.write(storagePackage.getData());
102           fos = new FileOutputStream(hashFile);
103           fos.write(storagePackage.getHashValue());
104           this.storePackage(storagePackage);
105           logger.info("Successfully stored data in file {}.",
106               dataFile.getCanonicalFile());
107         } catch (SecurityException | IOException e) {
108           logger.error(
109               "Could not write to local file system. Error was: {}.",
110               e.getMessage());
111         }
112       } else {
113         logger.trace("{} is already stored on {}.", storagePackage,
114             this.getUid());
115       }
116     }
117   }
118 }
```

Listing A.33: shared.storageNode.LocalStorageNode

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
2
```

```
3 import java.lang.reflect.Field;
4 import java.net.InetAddress;
5
6 import org.apache.logging.log4j.LogManager;
7 import org.apache.logging.log4j.Logger;
8
9 import edu.illinois.ece.ftstorage.shared.AbstractStorageEntity;
10 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
11
12 public abstract class StorageNode extends AbstractStorageEntity {
13
14   private static final long serialVersionUID = -4845690869027863643L;
15
16   protected final static Logger logger = LogManager.getLogger(Thread
17       .currentThread().getStackTrace()[1].getClassName());
18   public final static int DEFAULT_SERVER_STORAGE_PORT = 10001;
19   public final static InetAddress DEFAULT_SERVER_STORAGE_IP = InetAddress
20       .getLoopbackAddress();
21
22   StorageCluster storageCluster;
23   InetAddress storageServerAddress;
24   int storageServerPort;
25
26   protected StorageNode(String uid, String name, String description) {
27     super(uid, name, description);
28     this.storageServerAddress = DEFAULT_SERVER_STORAGE_IP;
29     this.storageServerPort = DEFAULT_SERVER_STORAGE_PORT;
30   }
31
32   public StorageNode(String uid, String name, String description,
33       InetAddress storageServerAddress, int storageServerPort) {
34     this(uid, name, description);
35     this.storageServerAddress = storageServerAddress;
36     this.storageServerPort = storageServerPort;
37   }
38
39   public StorageNode(String uid, String name, String description,
40       StorageCluster storageCluster, InetAddress storageServerAddress,
41       int storageServerPort) {
42     this(uid, name, description, storageServerAddress, storageServerPort);
43     this.storageCluster = storageCluster;
44   }
45
46   public StorageCluster getStorageCluster() {
47     return this.storageCluster;
48   }
49
50   public InetAddress getStorageServerAddress() {
51     return this.storageServerAddress;
52   }
53
54   public int getStorageServerPort() {
55     return this.storageServerPort;
56   }
57
58   public void setStorageCluster(StorageCluster storageCluster) {
59     this.storageCluster = storageCluster;
60   }
61
62   public void setStorageServerAddress(InetAddress storageServerAddress) {
63     this.storageServerAddress = storageServerAddress;
64   }
```

```
65
66   public void setStorageServerPort(int storageServerPort) {
67     this.storageServerPort = storageServerPort;
68   }
69
70   @Override
71   public String toString() {
72     StringBuilder sb = new StringBuilder();
73     try {
74       Field f[] = this.getClass().getSuperclass().getDeclaredFields();
75       sb.append("StorageNode [");
76       for (Field field : f) {
77         sb.append(field.getName());
78         sb.append("=");
79         sb.append(field.get(this));
80         sb.append(", ");
81       }
82       sb.append(']');
83     } catch (IllegalArgumentException | IllegalAccessException e) {
84       e.printStackTrace();
85     }
86     return sb.toString();
87   }
88
89   @Override
90   public void run() {
91   }
92 }
```

Listing A.34: shared.storageNode.StorageNode

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
2
3 import java.lang.reflect.Constructor;
4 import java.lang.reflect.InvocationTargetException;
5 import java.net.InetAddress;
6 import java.net.UnknownHostException;
7 import java.util.HashMap;
8 import java.util.Map;
9 import java.util.NoSuchElementException;
10 import java.util.UUID;
11
12 import org.apache.commons.configuration.HierarchicalConfiguration;
13 import org.apache.logging.log4j.LogManager;
14 import org.apache.logging.log4j.Logger;
15
16 import edu.illinois.ece.ftstorage.shared.storageCluster.StorageCluster;
17
18 @SuppressWarnings({ "unchecked", "rawtypes" })
19 public class StorageNodeFactory {
20
21   protected final static Logger logger = LogManager.getLogger(Thread
22       .currentThread().getStackTrace()[1].getClassName());
23   private static StorageNodeFactory instance;
24
25   private static StorageNode fromXmlConfiguration(
26       HierarchicalConfiguration nodeElement) {
27     StorageNode storageNode;
28     StorageNodeType storageNodeType = StorageNodeType.DefaultStorageNode;
29
30     try {
31       storageNodeType = StorageNodeType.valueOf(nodeElement
```

```
32          .getString("[@type]"));
33    } catch (IllegalArgumentException | NullPointerException e) {
34      logger.warn(
35          "FTProtocol "
36              + nodeElement.getString("[@type]")
37              + " not recognized or not defined in config file. Using
                ↪ default value. Message was: {}.",
38          e.getMessage());
39    }
40    String nodeUid = nodeElement.getString("[@uid]");
41    String nodeName = nodeElement.getString("[@name]");
42    String nodeDescription = nodeElement.getString("Description");
43    InetAddress storageServerAddress =
      ↪ StorageNode.DEFAULT_SERVER_STORAGE_IP;
44    int storageServerPort = StorageNode.DEFAULT_SERVER_STORAGE_PORT;
45    try {
46      storageServerPort = nodeElement.getInt("ListeningPort");
47    } catch (NoSuchElementException e) {
48      // ignore Exception: ListeningPort was not defined and default value
49      // will be used
50      logger.debug("Using default value {} for StorageNode {}.",
51          StorageNode.DEFAULT_SERVER_STORAGE_PORT, nodeUid);
52    }
53    try {
54      storageServerAddress = InetAddress.getByName(nodeElement
55          .getString("IPAddress"));
56    } catch (NoSuchElementException e) {
57      // ignore Exception: IPAddress was not defined and default value
58      // will be used
59      logger.info(
60          "Undefined element found. Using default value {} for StorageNode
            ↪ {}. Message was: {}. ",
61          StorageNode.DEFAULT_SERVER_STORAGE_IP, nodeUid,
62          e.getMessage());
63    } catch (UnknownHostException e) {
64      try {
65        storageServerAddress = InetAddress.getLocalHost();
66      } catch (UnknownHostException e1) {
67        logger.error(
68            "Error getting localhost InetAddress for StorageNode. Please
              ↪ make sure that you have at least one network card
              ↪ configured! Message was: {}.",
69            e1.getMessage());
70      }
71    }
72    storageNode = get(nodeUid, nodeName, nodeDescription,
73        storageServerAddress, storageServerPort, storageNodeType);
74    return storageNode;
75  }
76
77  public static StorageNode fromXmlConfigurationAndCluster(
78      HierarchicalConfiguration nodeElement, StorageCluster storageCluster)
79      throws UnknownHostException {
80    StorageNode storageNode = fromXmlConfiguration(nodeElement);
81    storageNode.setStorageCluster(storageCluster);
82    storageCluster.addNode(storageNode);
83    return storageNode;
84  }
85
86  private static StorageNode get(final String uid, final String name,
87      final String description, final InetAddress inetAddress,
88      final int storageServerPort, final StorageNodeType storageNodeType) {
```

```
89      Class[] constructorArgs = new Class[] { String.class, String.class,
90          String.class, InetAddress.class, int.class };
91      Constructor<StorageNode> ctor;
92      try {
93        ctor = getInstance().storageNodeTypes.get(storageNodeType)
94            .getConstructor(constructorArgs);
95        return ctor.newInstance(uid, name, description, inetAddress,
96            storageServerPort);
97      } catch (SecurityException | InstantiationException
98          | IllegalAccessException | IllegalArgumentException
99          | InvocationTargetException | NoSuchMethodException e) {
100       logger.error(
101           "Class constructor({}) not defined or not visible for {}. Using
            ↪ default class {}. Message was: {}.",
102           constructorArgs, storageNodeType, DefaultStorageNode.class,
103           e.getMessage());
104     }
105     return new DefaultStorageNode(uid, name, description, inetAddress,
106         storageServerPort);
107   }
108
109   // private static StorageNode get(final String uid, final String name,
110   // final String description,
111   // final StorageNode.StorageNodeType storageNodeType) {
112   // Class[] constructorArgs = new Class[] { String.class, String.class,
113   // String.class };
114   // Constructor<StorageNode> ctor;
115   // try {
116   // ctor = getInstance().storageNodeTypes.get(storageNodeType)
117   // .getConstructor(constructorArgs);
118   // return ctor.newInstance(uid, name, description);
119   // } catch (SecurityException | InstantiationException
120   // | IllegalAccessException | IllegalArgumentException
121   // | InvocationTargetException | NoSuchMethodException e) {
122   // logger.error(
123   // "Class constructor({}) not defined or not visible for {}. Using
      ↪ default class {}. Message was: {}.",
124   // constructorArgs, storageNodeType, DefaultStorageNode.class,
125   // e.getMessage());
126   // }
127   // return new DefaultStorageNode(uid, name, description);
128   // }
129
130   public static StorageNode getDefaultStorageNode() {
131     StorageNode storageNode = new LocalStorageNode(UUID.randomUUID()
132         .toString(), "DefaultStorageNode", "DefaultStorageNode");
133     return storageNode;
134   }
135
136   public static StorageNodeFactory getInstance() {
137     if (instance == null) {
138       instance = new StorageNodeFactory();
139     }
140     return instance;
141   }
142
143   private Map<StorageNodeType, Class> storageNodeTypes;
144
145   private StorageNodeFactory() {
146     this.storageNodeTypes = new HashMap<StorageNodeType, Class>(
147         StorageNodeType.values().length);
148     for (StorageNodeType type : StorageNodeType.values()) {
```

```
149       Class clazz;
150       String className = this.getClass().getPackage().getName() + "."
151           + type;
152       try {
153         clazz = Class.forName(className);
154         this.storageNodeTypes.put(type, clazz);
155       } catch (SecurityException | ClassNotFoundException e) {
156         logger.error(
157             "Class constructor not defined or not visible for {}. Using
                ↪ default class {}. Message was: {}.",
158             className, DefaultStorageNode.class, e.getMessage());
159         clazz = LocalStorageNode.class;
160       }
161     }
162   }
163 }
```

Listing A.35: shared.storageNode.StorageNodeFactory

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
2
3 import java.io.IOException;
4
5 import edu.illinois.ece.ftstorage.shared.AbstractPackageHandler;
6 import edu.illinois.ece.ftstorage.shared.storagePackage.StoragePackage;
7
8 public class StorageNodePackageHandler extends AbstractPackageHandler {
9
10   StorageNode requester;
11   StorageNode requestee;
12
13   public StorageNodePackageHandler(StorageNode requester,
14       StorageNode requestee, StoragePackage storagePackage)
15       throws IOException {
16     super(requester, requestee, storagePackage);
17     this.requester = requester;
18     this.requestee = requestee;
19   }
20
21   @Override
22   public void read() {
23
24   }
25
26   @Override
27   public void receiveReadAck() {
28
29   }
30
31   @Override
32   public void receiveWriteAck() {
33
34   }
35
36   @Override
37   public void write() {
38
39   }
40
41 }
```

Listing A.36: shared.storageNode.StorageNodePackageHandler

```
1 package edu.illinois.ece.ftstorage.shared.storageNode;
2
3 public enum StorageNodeType {
4   DefaultStorageNode, AmazonS3StorageNode, AzureStorageNode,
    ↪ BoxStorageNode, DropboxStorageNode, LocalStorageNode;
5 }
```

Listing A.37: shared.storageNode.StorageNodeType

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 import java.io.Serializable;
4
5 public abstract class AbstractStoragePackageMetadata implements
  ↪ Serializable,
6     Cloneable {
7
8   private static final long serialVersionUID = -2364388622470772052L;
9
10   public AbstractStoragePackageMetadata() {
11   }
12
13   @Override
14   public abstract boolean equals(Object obj);
15
16   @Override
17   protected abstract AbstractStoragePackageMetadata clone()
18       throws CloneNotSupportedException;
19 }
```

Listing A.38: shared.storagePackage.AbstractStoragePackageMetadata

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 public enum HashingSchemeType {
4   NONE, MD5, SHA1;
5 }
```

Listing A.39: shared.storagePackage.HashingSchemeType

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 import java.util.List;
4
5 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
6
7 /**
8  * Interface declaring methods to encode/decode an object, verify and set
   ↪ its
9  * hash value, retrieve the object's data, hashed data, as well as its
   ↪ coding
10  * and hashing scheme.
11  *
12  */
13 public interface ICodable {
14
15   /**
16    * Decode the object's data, according to its coding scheme, taking into
17    * account a list of encoded objects.
18    *
19    * @param encodedObjects
```

```
20     * @return An {@code ICodable} object containing the decoded data.
21     */
22    ICodable decode(List<? extends ICodable> encodedObjects);
23
24    /**
25     * Encode the object's data, according to its coding scheme, into a given
26     * number of blocks.
27     *
28     * @param numberOfBlocks
29     *            Number of data blocks to create.
30     * @return List of {@code ICodable}, each with an encoded data block.
31     */
32    List<? extends ICodable> encode(int numberOfBlocks);
33
34    /**
35     *
36     * @return This object's {@code CodingScheme}.
37     */
38    CodingSchemeType getCodingScheme();
39
40    /**
41     *
42     * @return This object's data.
43     */
44    byte[] getData();
45
46    /**
47     *
48     * @return This object's {@code HashingScheme}.
49     */
50    HashingSchemeType getHashingScheme();
51
52    /**
53     *
54     * @return This object's set hash/digest value.
55     */
56    byte[] getHashValue();
57
58    /**
59     *
60     * @param codingScheme
61     *            Set the object's {@code CodingScheme} to this value.
62     */
63    void setCodingScheme(CodingSchemeType codingScheme);
64
65    /**
66     *
67     * @param data
68     *            Set the object's data to this value.
69     */
70    void setData(byte[] data);
71
72    /**
73     *
74     * @param hashingScheme
75     *            Set the object's {@code HashingScheme} to this value.
76     */
77    void setHashingScheme(HashingSchemeType hashingScheme);
78
79    /**
80     * Set the object's hash/digest value based on the current data and
81     * {@code HashingScheme}.
```

```
82      */
83     void setHashValue();
84
85     /**
86      *
87      * @return {@code true} if the object's currently set data hashes
        ↪ correctly
88      *         to the currently set hash value.
89      */
90     boolean verifyHashValue();
91
92     /**
93      *
94      * @param metaData
95      *            Set the object's metadata to this value.
96      */
97     void setMetadata(AbstractStoragePackageMetadata metadata);
98
99     /**
100     *
101     * @return The object's metadata for a given subclass of
102     *         {@link AbstractStoragePackageMetadata}.
103     */
104    AbstractStoragePackageMetadata getMetadata(
105        Class<? extends AbstractStoragePackageMetadata> metadata);
106 }
```

Listing A.40: shared.storagePackage.ICodable

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 import java.io.ByteArrayOutputStream;
4 import java.io.File;
5 import java.io.FileInputStream;
6 import java.io.IOException;
7 import java.io.InputStream;
8 import java.io.ObjectOutput;
9 import java.io.ObjectOutputStream;
10 import java.io.Serializable;
11 import java.net.InetAddress;
12 import java.util.Arrays;
13 import java.util.Date;
14 import java.util.HashMap;
15 import java.util.LinkedList;
16 import java.util.List;
17 import java.util.Map;
18 import java.util.UUID;
19
20 import org.apache.commons.configuration.XMLConfiguration;
21 import org.apache.logging.log4j.LogManager;
22 import org.apache.logging.log4j.Logger;
23
24 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeFactory;
25 import edu.illinois.ece.ftstorage.shared.codingScheme.CodingSchemeType;
26 import edu.illinois.ece.ftstorage.shared.util.HashingUtil;
27
28 public final class StoragePackage implements Serializable, ICodable,
   ↪ Cloneable {
29
30     /**
31      *
32      */
```

```
33   private static final long serialVersionUID = 6941250846825013364L;
34
35   protected final static Logger logger = LogManager.getLogger(Thread
36       .currentThread().getStackTrace()[1].getClassName());
37
38   protected final UUID uuid;
39   protected final StoragePackageType packageType;
40   // do not set the timeStamp here because it will change when we cast it
41   // after transmitting the object
42   protected Date timeStamp;
43
44   protected byte[] data;
45   protected byte[] hashValue;
46   // allow the data to be used speculatively before it has been verified
47   protected boolean dataSpeculative;
48
49   protected CodingSchemeType codingScheme;
50   protected HashingSchemeType hashingScheme;
51   protected Map<String, AbstractStoragePackageMetadata> metadata;
52
53   protected InetAddress clientListeningIP;
54   protected int clientListeningPort;
55   protected InetAddress sourceAckListeningIP;
56   protected int sourceAckListeningPort;
57
58   /**
59    * StoragePackage for reading.
60    *
61    * @param uuid
62    *           UUID of StoragePackage to be read.
63    */
64   public StoragePackage(UUID uuid) {
65     this(uuid, CodingSchemeType.NONE, HashingSchemeType.NONE);
66   }
67
68   /**
69    * StoragePackage for reading.
70    *
71    * @param uuid
72    *           UUID of StoragePackage to be read.
73    * @param codingSchemeType
74    *           {@link CodingSchemeType} of StoragePackage to be read.
75    * @param hashingSchemeType
76    *           {@link HashingSchemeType} of StoragePackage to be read.
77    */
78   public StoragePackage(UUID uuid, CodingSchemeType codingSchemeType,
79       HashingSchemeType hashingSchemeType) {
80     this(uuid, new byte[] {}, new byte[] {}, null, 0, null, 0,
81         codingSchemeType, hashingSchemeType, StoragePackageType.READ);
82   }
83
84   public StoragePackage(String data) {
85     this(data.getBytes());
86   }
87
88   public StoragePackage(String data, CodingSchemeType codingSchemeType,
89       HashingSchemeType hashingSchemeType) {
90     this(UUID.randomUUID(), data.getBytes(), HashingUtil.encode(
91         hashingSchemeType, data.getBytes()), null, 0, null, 0,
92         codingSchemeType, hashingSchemeType, StoragePackageType.DATA);
93   }
94
```

```
95   public StoragePackage(byte[] data) {
96     this(UUID.randomUUID(), data, new byte[] {}, null, 0, null, 0,
97         CodingSchemeType.NONE, HashingSchemeType.NONE,
98         StoragePackageType.DATA);
99   }
100
101  public StoragePackage(byte[] data, CodingSchemeType codingSchemeType,
102      HashingSchemeType hashingSchemeType) {
103    this(UUID.randomUUID(), data, HashingUtil.encode(hashingSchemeType,
104        data), null, 0, null, 0, codingSchemeType, hashingSchemeType,
105        StoragePackageType.DATA);
106  }
107
108  public StoragePackage(UUID uuid, byte[] data, byte[] hashValue) {
109    this(uuid, data, hashValue, null, 0, null, 0, CodingSchemeType.NONE,
110        HashingSchemeType.NONE, StoragePackageType.DATA);
111  }
112
113  public StoragePackage(XMLConfiguration xmlConifguration) {
114    this(UUID.nameUUIDFromBytes(new String("CONFIGURATION").getBytes()),
115        new byte[] {}, new byte[] {}, null, 0, null, 0,
116        CodingSchemeType.NONE, HashingSchemeType.NONE,
117        StoragePackageType.CONFIGURATION);
118    ByteArrayOutputStream bos = new ByteArrayOutputStream();
119    ObjectOutput out = null;
120    try {
121      out = new ObjectOutputStream(bos);
122      out.writeObject(xmlConifguration.getDocument());
123      byte[] data = bos.toByteArray();
124      this.data = data;
125      this.setHashValue();
126    } catch (IOException e) {
127      logger.error(
128          "{} of type {} has not been properly initialized (could not set
             ↪ data property). Error was: {}.",
129          this.getClass(), this.getPackageType(), e.getMessage());
130    }
131  }
132
133  public StoragePackage(XMLConfiguration xmlConifguration,
134      CodingSchemeType codingScheme, HashingSchemeType hashingScheme) {
135    this(xmlConifguration);
136    this.codingScheme = codingScheme;
137    this.hashingScheme = hashingScheme;
138    this.setHashValue();
139  }
140
141  public StoragePackage(File file) {
142    this(UUID.nameUUIDFromBytes(new String("APPLICATION").getBytes()),
143        new byte[] {}, new byte[] {}, null, 0, null, 0,
144        CodingSchemeType.NONE, HashingSchemeType.NONE,
145        StoragePackageType.APPLICATION);
146    ByteArrayOutputStream baos = new ByteArrayOutputStream(
147        (int) file.length());
148    InputStream input = null;
149    try {
150      input = new FileInputStream(file);
151      // initialize a buffer of size 128KB
152      byte[] buf = new byte[1024];
153      int bytesRead = input.read(buf);
154      while (bytesRead != -1) {
155        baos.write(buf, 0, bytesRead);
```

```
156        bytesRead = input.read(buf);
157        // byte[] data = Files.readAllBytes(file.toPath());
158        baos.write(bytesRead);
159      }
160      this.data = baos.toByteArray();
161    } catch (IOException e) {
162      logger.error(
163          "{} of type {} has not been properly initialized (could not set
           ↪ data property). Error was: {}.",
164          this.getClass(), this.getPackageType(), e.getMessage());
165      e.printStackTrace();
166    }
167  }
168
169  public StoragePackage(File file, CodingSchemeType codingScheme,
170      HashingSchemeType hashingScheme) {
171    this(file);
172    this.codingScheme = codingScheme;
173    this.hashingScheme = hashingScheme;
174    this.setHashValue();
175  }
176
177  public StoragePackage(UUID uid, byte[] data, byte[] hashValue,
178      InetAddress senderIP, int senderPort,
179      InetAddress clientListeningIP, int clientListeningPort,
180      CodingSchemeType codingScheme, HashingSchemeType hashingScheme,
181      StoragePackageType packageType) {
182    this.metadata = new HashMap<String, AbstractStoragePackageMetadata>();
183    this.uuid = uid;
184    this.data = data;
185    this.hashValue = hashValue;
186    this.sourceAckListeningIP = senderIP;
187    this.sourceAckListeningPort = senderPort;
188    this.clientListeningIP = clientListeningIP;
189    this.clientListeningPort = clientListeningPort;
190    this.codingScheme = codingScheme;
191    this.hashingScheme = hashingScheme;
192    this.packageType = packageType;
193    this.dataSpeculative = true;
194    this.timeStamp = new Date();
195  }
196
197  public StoragePackage createAckPackage() {
198    StoragePackage newStoragePackage = new StoragePackage(this.uuid,
199        this.data, this.hashValue, this.sourceAckListeningIP,
200        this.sourceAckListeningPort, this.clientListeningIP,
201        this.clientListeningPort, this.codingScheme,
202        this.hashingScheme, StoragePackageType.ACK);
203    for (AbstractStoragePackageMetadata metadata : this.metadata.values())
       ↪ {
204      newStoragePackage.setMetadata(metadata);
205    }
206    newStoragePackage.timeStamp = this.timeStamp;
207    return newStoragePackage;
208  }
209
210  @Override
211  public StoragePackage decode(List<? extends ICodable> encodedObjects) {
212    List<byte[]> bl = new LinkedList<byte[]>();
213    for (ICodable iCodable : encodedObjects) {
214      bl.add(iCodable.getData());
215    }
```

```
216     byte[] decodedData = CodingSchemeFactory.get(this.codingScheme).decode(
217         this, bl);
218     if (decodedData != null) {
219       this.setData(decodedData);
220       return this;
221     }
222     return null;
223   }
224 
225   @Override
226   public List<StoragePackage> encode(int numberOfBlocks) {
227     List<StoragePackage> spl = new LinkedList<StoragePackage>();
228     for (byte[] codedBytes : CodingSchemeFactory.get(this.codingScheme)
229         .encode(this, numberOfBlocks)) {
230       StoragePackage codedStoragePackage = new StoragePackage(this.uuid,
231           codedBytes, this.hashValue, this.sourceAckListeningIP,
232           this.sourceAckListeningPort, this.clientListeningIP,
233           this.clientListeningPort, this.codingScheme,
234           this.hashingScheme, this.packageType);
235       codedStoragePackage.timeStamp = this.getTimeStamp();
236       for (AbstractStoragePackageMetadata metadata : this.metadata
237           .values()) {
238         codedStoragePackage.setMetadata(metadata);
239       }
240       spl.add(codedStoragePackage);
241     }
242     return spl;
243   }
244 
245   public InetAddress getClientListeningIP() {
246     return this.clientListeningIP;
247   }
248 
249   public int getClientListeningPort() {
250     return this.clientListeningPort;
251   }
252 
253   @Override
254   public CodingSchemeType getCodingScheme() {
255     return this.codingScheme;
256   }
257 
258   @Override
259   public byte[] getData() {
260     return this.data;
261   }
262 
263   @Override
264   public HashingSchemeType getHashingScheme() {
265     return this.hashingScheme;
266   }
267 
268   @Override
269   public byte[] getHashValue() {
270     return this.hashValue;
271   }
272 
273   /**
274    *
275    * Get the object's metadata for an associated subclass of
276    * {@link AbstractStoragePackageMetadata}.
277    *
```

```
278    * {@inheritDoc}
279    *
280    */
281   @Override
282   public AbstractStoragePackageMetadata getMetadata(
283       Class<? extends AbstractStoragePackageMetadata> metadata) {
284     if (metadata != null) {
285       String className = metadata.getName();
286       if (this.metadata.containsKey(className)) {
287         return this.metadata.get(className);
288       }
289     }
290     return null;
291   }
292
293   /**
294    *
295    * @return The object's current package type.
296    */
297   public StoragePackageType getPackageType() {
298     return this.packageType;
299   }
300
301   public InetAddress getSourceAckListeningIP() {
302     return this.sourceAckListeningIP;
303   }
304
305   public int getSourceAckListeningPort() {
306     return this.sourceAckListeningPort;
307   }
308
309   public UUID getUid() {
310     return this.uuid;
311   }
312
313   public void setClientListeningIP(InetAddress clientListeningIP) {
314     logger.trace("Setting client ACK listening IP to {}",
        ↪ clientListeningIP);
315     this.clientListeningIP = clientListeningIP;
316   }
317
318   public void setClientListeningPort(int clientListeningPort) {
319     logger.trace("Setting client ACK listening port to {}",
320         clientListeningPort);
321     this.clientListeningPort = clientListeningPort;
322   }
323
324   @Override
325   public void setCodingScheme(CodingSchemeType codingScheme) {
326     this.codingScheme = codingScheme;
327   }
328
329   @Override
330   public void setData(byte[] data) {
331     this.data = data;
332   }
333
334   @Override
335   public void setHashingScheme(HashingSchemeType hashingScheme) {
336     this.hashingScheme = hashingScheme;
337   }
338
```

```
339    @Override
340    public void setHashValue() {
341      this.hashValue = HashingUtil.encode(this.hashingScheme, this.data);
342    }
343
344    public boolean isSpeculative() {
345      return this.dataSpeculative;
346    }
347
348    public void setSpeculative(boolean speculative) {
349      // logger.trace("Marking {} speculative: {}.", this, speculative);
350      this.dataSpeculative = speculative;
351    }
352
353    /**
354     *
355     * Set the object's metadata for an associated subclass of
356     * {@link AbstractStoragePackageMetadata}.
357     *
358     * {@inheritDoc}
359     */
360    @Override
361    public void setMetadata(AbstractStoragePackageMetadata md) {
362      if (md != null) {
363        this.metadata.put(md.getClass().getName(), md);
364      }
365    }
366
367    public void setSourceAckListeningIP(InetAddress sourceAckListeningIP) {
368      logger.trace("Setting source ACK listening IP to {}",
369          sourceAckListeningIP);
370      this.sourceAckListeningIP = sourceAckListeningIP;
371    }
372
373    public void setSourceAckListeningPort(int sourceAckListeningPort) {
374      logger.trace("Setting source ACK listening port to {}",
375          sourceAckListeningPort);
376      this.sourceAckListeningPort = sourceAckListeningPort;
377    }
378
379    @Override
380    public String toString() {
381      StringBuilder sb = new StringBuilder();
382      try {
383        sb.append("StoragePackage [");
384        sb.append("UUID=" + this.uuid);
385        sb.append(", ");
386        sb.append("Type=" + this.packageType);
387        sb.append(']');
388      } catch (IllegalArgumentException e) {
389        e.printStackTrace();
390      }
391      return sb.toString();
392    }
393
394    @Override
395    public boolean verifyHashValue() {
396      return HashingUtil
397          .verify(this.hashingScheme, this.data, this.hashValue);
398    }
399
400    public Date getTimeStamp() {
```

```
401     return this.timeStamp;
402   }
403
404   @Override
405   public boolean equals(Object obj) {
406     if (obj != null && obj instanceof StoragePackage) {
407       StoragePackage other = (StoragePackage) obj;
408       if (!other.getTimeStamp().equals(this.timeStamp)) {
409         logger.trace(other.getTimeStamp() + "!= " + this.timeStamp);
410         return false;
411       }
412       if (!other.getUid().equals(this.getUid())) {
413         logger.trace(other.getUid() + "!= " + this.timeStamp);
414         return false;
415       }
416       if (!other.getSourceAckListeningIP().equals(
417           this.getSourceAckListeningIP())) {
418         logger.trace(other.getSourceAckListeningIP() + "!= "
419             + this.getSourceAckListeningIP());
420         return false;
421       }
422       if (other.getSourceAckListeningPort() != this
423           .getSourceAckListeningPort()) {
424         logger.trace(other.getSourceAckListeningPort() + "!= "
425             + this.getSourceAckListeningPort());
426         return false;
427       }
428       if (!other.getCodingScheme().equals(this.getCodingScheme())) {
429         logger.trace(other.getCodingScheme() + "!= "
430             + this.getCodingScheme());
431         return false;
432       }
433       if (!other.getClientListeningIP().equals(
434           this.getClientListeningIP())) {
435         logger.trace(other.getClientListeningIP() + "!= "
436             + this.getClientListeningIP());
437         return false;
438       }
439       if (other.getClientListeningPort() != this.getClientListeningPort())
        ↪ {
440         logger.trace(other.getClientListeningPort() + "!= "
441             + this.getClientListeningPort());
442         return false;
443       }
444       if (!other.getHashingScheme().equals(this.getHashingScheme())) {
445         logger.trace(other.getHashingScheme() + "!= "
446             + this.getHashingScheme());
447         return false;
448       }
449       if (!other.getPackageType().equals(this.getPackageType())) {
450         logger.trace(other.getPackageType() + "!= "
451             + this.getPackageType());
452         return false;
453       }
454       if (!Arrays.equals(other.getHashValue(), this.getHashValue())) {
455         logger.trace(other.getHashValue() + "!= " + this.getHashValue());
456         return false;
457       }
458       for (AbstractStoragePackageMetadata metadata : this.metadata
459           .values()) {
460         if (other.getMetadata(metadata.getClass()) != null
461             && !other.getMetadata(metadata.getClass()).equals(
```

```
462                 metadata)) {
463           logger.trace(other.getMetadata(metadata.getClass()) + "!= "
464               + this.getMetadata(metadata.getClass()));
465           return false;
466         }
467       }
468       if (!Arrays.equals(other.getData(), this.getData())) {
469         // keep this until the end. with large byte arrays this could
470         // take a while
471         logger.trace("other.getData()!=this.getData())");
472         return false;
473       }
474     } else {
475       return false;
476     }
477     logger.trace(obj.toString() + " == " + this);
478     return true;
479   }
480
481   @Override
482   public StoragePackage clone() {
483     StoragePackage newStoragePackage = new StoragePackage(this.uuid,
484         this.data, this.hashValue, this.sourceAckListeningIP,
485         this.sourceAckListeningPort, this.clientListeningIP,
486         this.clientListeningPort, this.codingScheme,
487         this.hashingScheme, this.packageType);
488     for (AbstractStoragePackageMetadata metadata : this.metadata.values())
        ↪ {
489       try {
490         newStoragePackage.setMetadata(metadata.clone());
491       } catch (CloneNotSupportedException e) {
492         logger.error(
493             "Could not set metadata for class {} because cloning is not
                ↪ supported. Error message was: {}",
494             e.getMessage());
495       }
496     }
497     newStoragePackage.timeStamp = this.timeStamp;
498     return newStoragePackage;
499   }
500 }
```

Listing A.41: shared.storagePackage.StoragePackage

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 import org.apache.logging.log4j.LogManager;
4 import org.apache.logging.log4j.Logger;
5
6 public class StoragePackageFactory {
7
8   protected final static Logger logger = LogManager.getLogger(Thread
9       .currentThread().getStackTrace()[1].getClassName());
10
11 }
```

Listing A.42: shared.storagePackage.StoragePackageFactory

```
1 package edu.illinois.ece.ftstorage.shared.storagePackage;
2
3 import java.io.Serializable;
```

```
4
5 public enum StoragePackageType implements Serializable {
6   ACK, CONFIGURATION, DATA, APPLICATION, READ;
7 }
```

Listing A.43: shared.storagePackage.StoragePackageType

```
1 package edu.illinois.ece.ftstorage.shared.util;
2
3 import java.io.BufferedWriter;
4 import java.io.File;
5 import java.io.FileWriter;
6 import java.io.IOException;
7
8 import org.apache.logging.log4j.LogManager;
9 import org.apache.logging.log4j.Logger;
10
11 import edu.illinois.ece.ftstorage.shared.storageNode.StorageNode;
12
13 /**
14  * Creates configuration files in {@value #clusterConfigFileName} and
15  * {@value #serverConfigFileName}. Takes as parameters the maximum number
   ↪ of
16  * failures (f) and whether the configuration files should be created
17  * individually. Based on these parameters, f clusters will be created. The
18  * first (non-fault tolerant) cluster will consist of f nodes, whereas any
19  * clusters that tolerate more than f failures will contain 3f+1 nodes. If
20  * individual configuration files are to be created, then the default
   ↪ listening
21  * port will be used for all servers (
22  * {@value StorageNode#DEFAULT_SERVER_STORAGE_PORT}) and the IP address
   ↪ range
23  * will be in 192.168.122.1/24. If one configuration file is to be
   ↪ created, the
24  * local loopback address will be used as the server IP addresses
   ↪ (127.0.0.1)
25  * and the TCP port range will increase starting from 10000. Currently
   ↪ there are
26  * no sanity checks for ranges, meaning that if the number of nodes
   ↪ exceeds 255,
27  * invalid IP addresses will be written into the configuration file.
28  *
29  * @author adjokic
30  *
31  */
32 public class ConfigUtil {
33
34   private final static Logger logger = LogManager.getLogger(Thread
35       .currentThread().getStackTrace()[1].getClassName());
36
37   final static String clusterConfigFileName = "clusterConfig.xml";
38   final static String serverConfigFileName = "serverConfig.xml";
39
40   private void writeToFile(String fileName, String text, boolean append) {
41     fileName = "config" + File.separator + fileName;
42     logger.debug("Writing to {}.", fileName);
43     File outFile = new File(fileName);
44     FileWriter fw = null;
45     BufferedWriter bw = null;
46     try {
47       (outFile.getParentFile()).mkdirs();
48       fw = new FileWriter(fileName);
```

```
49        bw = new BufferedWriter(fw);
50        if (append) {
51          bw.append(text);
52        } else {
53          bw.write(text);
54        }
55      } catch (IOException e) {
56        logger.fatal("Caught {}. Error was: {}", e.getClass(),
57            e.getMessage());
58      } finally {
59        try {
60          bw.flush();
61          if (fw != null) {
62            fw.close();
63          }
64          if (bw != null) {
65            bw.close();
66          }
67        } catch (IOException e) {
68          logger.fatal("Caught {}. Error was: {}", e.getClass(),
69              e.getMessage());
70        }
71      }
72    }
73
74    private int clusterConfig(int f, boolean individualFiles) {
75      StringBuilder sb = new StringBuilder();
76      int nodeCounter = 0;
77      sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
78      sb.append("\n");
79      sb.append("<configuration type=\"ClusterConfig\">");
80      sb.append("\n");
81      sb.append("\t<Clusters>");
82      for (int i = 0; i <= f; i++) {
83        String clusterName = String.format("c%02d", i);
84        String nodeName = String.format("n%02d", nodeCounter);
85        sb.append("\n");
86        sb.append("\t\t<Cluster name=\""
87            + clusterName
88            + "\" uid=\""
89            + clusterName
90            + "\" ftprotocol=\""
91            + (i > 0 ? "DefaultFTProtocol\" primaryNode=\"" + nodeName
92                + "\"" : "NONE\"") + ">");
93        sb.append("\n");
94        sb.append("\t\t\t<Description>Local Storage "
95            + (i > 0 ? "BFT " : "") + "Cluster " + (i + 1)
96            + (i > 0 ? "; f=" + i : "") + "</Description>");
97        sb.append("\n");
98        sb.append("\t\t\t<Nodes>");
99        for (int j = 0; j < (i > 0 ? (3 * i + 1) : f); j++) {
100         // if we have zero tolerated faults, we want as many nodes in
101         // the cluster as the maximum faults for a given cluster in the
102         // global configuration
103         nodeName = String.format("n%02d", nodeCounter);
104         sb.append("\n");
105         sb.append("\t\t\t\t<Node name=\"" + nodeName + "\" uid=\""
106             + nodeName + "\" type=\"LocalStorageNode\">");
107         sb.append("\n");
108         sb.append("\t\t\t\t\t");
109         if (individualFiles) {
110           sb.append("<IPAddress>");
```

```
111           // start counting at 192.168.122.2
112           sb.append("192.168.122." + (nodeCounter + 2));
113           sb.append("</IPAddress>");
114           sb.append("<ListeningPort>10001</ListeningPort>");
115         } else {
116           sb.append("<IPAddress>127.0.0.1</IPAddress>");
117           sb.append("<ListeningPort>" + (10000 + nodeCounter)
118               + "</ListeningPort>");
119         }
120         sb.append("\n");
121         sb.append("\t\t\t\t</Node>");
122         nodeCounter++;
123       }
124       sb.append("\n");
125       sb.append("\t\t\t</Nodes>");
126       sb.append("\n");
127       sb.append("\t\t</Cluster>");
128     }
129     sb.append("\n");
130     sb.append("\t</Clusters>");
131     sb.append("\n");
132     sb.append("</configuration>");
133     sb.append("\n");
134     this.writeToFile(clusterConfigFileName, sb.toString(), false);
135     return nodeCounter;
136   }
137 
138   private String serverConfig(int numServers, boolean individualFiles) {
139     System.out.println("Creating file for " + numServers + " servers in "
140         + (individualFiles ? "individual" : "one") + " file");
141     String serverConfig;
142     StringBuilder sb = new StringBuilder();
143     logger.info("Creating file with {} and ({}) individual files.",
144         numServers, individualFiles);
145     for (int i = 0; i < numServers; i++) {
146       String servername = String.format("s%02d", i);
147       String nodeName = String.format("n%02d", i);
148       logger.trace("Processing {}/{}.", servername, nodeName);
149       String fileName;
150       if (individualFiles) {
151         fileName = servername + File.separator + serverConfigFileName;
152       } else {
153         fileName = serverConfigFileName;
154       }
155       if (individualFiles || i == 0) {
156         sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
157         sb.append("\n");
158         sb.append("<configuration type=\"ServerConfig\">");
159         sb.append("\n");
160         sb.append("\t<Servers>");
161       }
162       sb.append("\n");
163       sb.append("\t\t<Server name=\"");
164       sb.append(servername);
165       sb.append("\" uid=\"");
166       sb.append(servername);
167       sb.append("\" type=\"DefaultServer\" storageNode=\"");
168       sb.append(nodeName);
169       sb.append("\">");
170       sb.append("\n");
171       sb.append("\t\t<Description>Default Server ");
172       sb.append(servername);
```

```
173      sb.append("</Description>");
174      sb.append("\n");
175      sb.append("\t\t</Server>");
176      sb.append("\n");
177      if (individualFiles || i == numServers - 1) {
178        sb.append("\t</Servers>");
179        sb.append("\n");
180        sb.append("</configuration>");
181        this.writeToFile(fileName, sb.toString(), !individualFiles);
182        if (individualFiles) {
183          sb = new StringBuilder();
184        }
185      }
186    }
187    serverConfig = sb.toString();
188    return serverConfig;
189  }
190
191  private void osConfig(int numServers) {
192    System.out.println("Creating hostname files for " + numServers
193        + " servers.");
194    StringBuilder sb = new StringBuilder();
195    int nodeCounter = 0;
196    sb.append("127.0.0.1\t\t" + "localhost.localdomain localhost\n");
197    sb.append("::1\t\t" + "localhost6.localdomain6 localhost6\n");
198    sb.append("130.126.139.141" + "\t" + "dawn1\n");
199    sb.append("192.168.122.1" + "\t" +
    ↪ "csl-dawn1a-459.csl.illinois.edu\n");
200    for (int i = 0; i < numServers; i++) {
201      String hostName = String.format("vm-test%02d", i);
202      sb.append("192.168.122." + (nodeCounter + 2) + "\t");
203      sb.append(hostName);
204      sb.append(" ");
205      sb.append(hostName + ".dawn.local");
206      sb.append("\n");
207      nodeCounter++;
208    }
209    String fileName = "os" + File.separator + "hosts";
210    this.writeToFile(fileName, sb.toString(), false);
211    for (int i = 0; i < numServers; i++) {
212      // write file hostname
213      String hostName = String.format("vm-test%02d.dawn.local", i);
214      String servername = String.format("s%02d", i);
215      fileName = servername + File.separator + "hostname";
216      this.writeToFile(fileName, hostName, false);
217
218      // write file network (for /etc/sysconfig/network in CentOS)
219      fileName = servername + File.separator + "network";
220      sb = new StringBuilder();
221      sb.append("NETWORKING=yes\n");
222      sb.append("NETWORKING_IPV6=no\n");
223      sb.append("HOSTNAME=" + hostName + "\n");
224      sb.append("NTPSERVERARGS=iburst");
225      this.writeToFile(fileName, sb.toString(), false);
226    }
227  }
228
229  public static void main(String[] args) {
230    int numServers = 0;
231    boolean individualFiles = true;
232    ConfigUtil cu = new ConfigUtil();
233    numServers = cu.clusterConfig(4, individualFiles);
```

```
234     cu.serverConfig(numServers, individualFiles);
235     cu.osConfig(numServers);
236     System.out.println("Done.");
237   }
238
239 }
```

Listing A.44: shared.util.ConfigUtil

```
1 package edu.illinois.ece.ftstorage.shared.util;
2
3 /**
4  * Class used to create a string of data of a certain size in bytes using
5  * {@link LoremIpsum}. The assumption is that one character is
6  * {@value #characterSizeBytes} bytes long.
7  */
8 public class CreateData {
9
10   private static LoremIpsum loremIpsum = new LoremIpsum();
11   // we don't want to create a string that's larger than 100MB
12   private final static int maxSizekBytes = 100 * 1024;
13   public final static int characterSizeBytes = 2;
14
15   public static String createString(int kBytes) {
16     if (kBytes > maxSizekBytes) {
17       kBytes = maxSizekBytes;
18     }
19     StringBuilder sb = new StringBuilder();
20     sb.append(loremIpsum.paragraph(true));
21     while (((sb.length() * characterSizeBytes) / 1024) < kBytes) {
22       sb.append(loremIpsum.paragraph());
23     }
24     return sb.toString();
25   }
26 }
```

Listing A.45: shared.util.CreateData

```
1 package edu.illinois.ece.ftstorage.shared.util;
2
3 import java.io.ByteArrayOutputStream;
4 import java.io.Closeable;
5 import java.io.File;
6 import java.io.IOException;
7 import java.io.OutputStreamWriter;
8 import java.io.UnsupportedEncodingException;
9
10 import javax.xml.transform.OutputKeys;
11 import javax.xml.transform.Transformer;
12 import javax.xml.transform.TransformerConfigurationException;
13 import javax.xml.transform.TransformerException;
14 import javax.xml.transform.TransformerFactory;
15 import javax.xml.transform.TransformerFactoryConfigurationError;
16 import javax.xml.transform.dom.DOMSource;
17 import javax.xml.transform.stream.StreamResult;
18
19 import org.apache.logging.log4j.LogManager;
20 import org.apache.logging.log4j.Logger;
21 import org.w3c.dom.Document;
22
23 public class FtStorageUtil {
```

```
24
25   private final static Logger logger = LogManager.getLogger(Thread
26       .currentThread().getStackTrace()[1].getClassName());
27
28   /**
29    * Close an array of {@code Closable} objects silently, i.e. only log an
30    * error if the {@code Closable} object is not null.
31    *
32    * @param closeable
33    *            The array of {@code Closable} objects to close.
34    */
35   public static final void closeSilently(Closeable... closeable) {
36     for (Closeable c : closeable) {
37       if (c != null) {
38         try {
39           c.close();
40         } catch (IOException e) {
41           logger.error("Caught {}. Error was: {}", e.getClass(),
42               e.getMessage());
43         }
44       }
45     }
46   }
47
48   /**
49    * Transform a {@link Document} to a {@link String}.
50    *
51    * @param initialBufferLength
52    *            Initial length of the buffer in bytes. If you know the
      ↪ length
53    *            of the document, use this value. If not, make it as
      ↪ accuarte
54    *            as possible - too low a number might slow things down, too
55    *            high a number might use up too much memory.
56    * @param newDocument
57    *            The document to transform into a string.
58    * @return The transformed document as a string.
59    * @throws TransformerFactoryConfigurationError
60    * @throws TransformerConfigurationException
61    * @throws TransformerException
62    * @throws UnsupportedEncodingException
63    */
64   public static final String documentToString(int initialBufferLength,
65       Document newDocument) throws TransformerFactoryConfigurationError,
66       TransformerConfigurationException, TransformerException,
67       UnsupportedEncodingException {
68     TransformerFactory tf = TransformerFactory.newInstance();
69     Transformer transformer = tf.newTransformer();
70     transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
71     transformer.setOutputProperty(OutputKeys.METHOD, "xml");
72     transformer.setOutputProperty(OutputKeys.INDENT, "yes");
73     transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
74     transformer.setOutputProperty(
75         "{http://xml.apache.org/xslt}indent-amount", "4");
76     ByteArrayOutputStream baos = new ByteArrayOutputStream(
77         initialBufferLength);
78     transformer.transform(new DOMSource(newDocument), new StreamResult(
79         new OutputStreamWriter(baos, "UTF-8")));
80     return baos.toString();
81   }
82
83   /**
```

```
84     * For a File's current folder location, create a new folder in it and
       ↪ make
85     * this the new directory containing the file. The preceding path of the
86     * file is kept and no files are moved.
87     *
88     * @param originalConfigFile
89     *            The file for which to create a new parent folder.
90     * @param newParentFolderName
91     *            The new folder name, which will contain the file.
92     * @return The string containing the file's path with the inserted new
93     *         parent directory of the file.
94     */
95   public static final String getPathWithNewParentFolderName(
96       File originalConfigFile, String newParentFolderName) {
97     if (originalConfigFile.getParent() != null) {
98       if (originalConfigFile.getParent().endsWith(newParentFolderName)
99           || originalConfigFile.getParent().endsWith(
100              newParentFolderName + File.separator)) {
101        return originalConfigFile.toString();
102      }
103      File f = new File(originalConfigFile.getParent(),
104          newParentFolderName);
105      boolean success;
106      if (success = f.exists()) {
107
108      } else {
109        success = (f.mkdirs());
110      }
111      if (!success) {
112        logger.error(
113            "Failed to create parent directory(ies) on path {}.",
114            f.getPath());
115      }
116      f = new File(f, originalConfigFile.getName());
117      logger.debug("Returning path: {}", f.getPath());
118      return f.getPath();
119
120    } else {
121      return newParentFolderName + File.separator
122          + originalConfigFile.getName();
123    }
124  }
125 }
```

Listing A.46: shared.util.FtStorageUtil

```
1 package edu.illinois.ece.ftstorage.shared.util;
2
3 import java.security.MessageDigest;
4 import java.security.NoSuchAlgorithmException;
5 import java.util.Arrays;
6
7 import org.apache.logging.log4j.LogManager;
8 import org.apache.logging.log4j.Logger;
9
10 import edu.illinois.ece.ftstorage.shared.storagePackage.HashingSchemeType;
11
12 public class HashingUtil {
13
14   protected final static Logger logger = LogManager.getLogger(Thread
15       .currentThread().getStackTrace()[1].getClassName());
16
```

```
17	public static byte[] encode(HashingSchemeType hashingScheme, byte[]
	↪ data) {
18	  MessageDigest md;
19	  byte[] calculatedDigest = new byte[] {};
20	  switch (hashingScheme) {
21	  case NONE:
22	    calculatedDigest = new byte[] {};
23	    break;
24	  case MD5:
25	    try {
26	      md = MessageDigest.getInstance("MD5");
27	      calculatedDigest = md.digest(data);
28	    } catch (NoSuchAlgorithmException e) {
29	      logger.warn("HashingScheme "
30	          + hashingScheme
31	          + " not recognized or not implemented. Using default value.");
32	    }
33	    break;
34	  case SHA1:
35	    try {
36	      md = MessageDigest.getInstance("SHA1");
37	      calculatedDigest = md.digest(data);
38	    } catch (NoSuchAlgorithmException e) {
39	      logger.warn("HashingScheme "
40	          + hashingScheme
41	          + " not recognized or not implemented. Using default value.");
42	    }
43	    break;
44	  default:
45	    logger.warn("HashingScheme "
46	        + hashingScheme
47	        + " not recognized or not implemented. Using default value.");
48	    calculatedDigest = new byte[] {};
49	    break;
50	  }
51	  return calculatedDigest;
52	}
53
54	public static boolean verify(HashingSchemeType hashingScheme, byte[]
	↪ data,
55	    byte[] dataDigest) {
56	  MessageDigest md;
57	  byte[] calculatedDigest = new byte[] {};
58	  logger.debug("Verifying {} for digest {}.", hashingScheme, dataDigest);
59	  switch (hashingScheme) {
60	  case NONE:
61	    calculatedDigest = data;
62	    break;
63	  case MD5:
64	    try {
65	      md = MessageDigest.getInstance("MD5");
66	      calculatedDigest = md.digest(data);
67	    } catch (NoSuchAlgorithmException e) {
68	      logger.warn("HashingScheme "
69	          + hashingScheme
70	          + " not recognized or not implemented. Using default value.");
71	    }
72	    break;
73	  case SHA1:
74	    try {
75	      md = MessageDigest.getInstance("SHA1");
76	      calculatedDigest = md.digest(data);
```

```
77       } catch (NoSuchAlgorithmException e) {
78         logger.warn("HashingScheme "
79             + hashingScheme
80             + " not recognized or not implemented. Using default value.");
81       }
82       break;
83     default:
84       logger.warn(
85           "HashingScheme "
86               + hashingScheme
87               + " not recognized or not implemented. Using default value
               ↪ {} for digest.",
88           calculatedDigest);
89       break;
90     }
91     return (Arrays.equals(calculatedDigest, dataDigest));
92   }
93
94 }
```

Listing A.47: shared.util.HashingUtil

```
1 package edu.illinois.ece.ftstorage.shared.util;
2
3 import java.util.Random;
4
5 /*
6  * Copyright 2010 Oliver C Dodd http://01001111.net
7  * Licensed under the MIT license:
  ↪ http://www.opensource.org/licenses/mit-license.php
8  */
9 public class LoremIpsum {
10
11   /*
12    * The Lorem Ipsum Standard Paragraph
13    */
14   protected final String standard = "Lorem ipsum dolor sit amet,
    ↪ consectetur adipisicing elit, sed do eiusmod tempor incididunt ut
    ↪ labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
    ↪ exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
    ↪ Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
    ↪ dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
    ↪ non proident, sunt in culpa qui officia deserunt mollit anim id est
    ↪ laborum.";
15   private final String[] words = { "a", "ac", "accumsan", "ad",
    ↪ "adipiscing",
16       "aenean", "aliquam", "aliquet", "amet", "ante", "aptent", "arcu",
17       "at", "auctor", "augue", "bibendum", "blandit", "class", "commodo",
18       "condimentum", "congue", "consectetur", "consequat", "conubia",
19       "convallis", "cras", "cubilia", "cum", "curabitur", "curae",
20       "cursus", "dapibus", "diam", "dictum", "dictumst", "dignissim",
21       "dis", "dolor", "donec", "dui", "duis", "egestas", "eget",
22       "eleifend", "elementum", "elit", "enim", "erat", "eros", "est",
23       "et", "etiam", "eu", "euismod", "facilisi", "facilisis", "fames",
24       "faucibus", "felis", "fermentum", "feugiat", "fringilla", "fusce",
25       "gravida", "habitant", "habitasse", "hac", "hendrerit",
26       "himenaeos", "iaculis", "id", "imperdiet", "in", "inceptos",
27       "integer", "interdum", "ipsum", "justo", "lacinia", "lacus",
28       "laoreet", "lectus", "leo", "libero", "ligula", "litora",
29       "lobortis", "lorem", "luctus", "maecenas", "magna", "magnis",
30       "malesuada", "massa", "mattis", "mauris", "metus", "mi",
31       "molestie", "mollis", "montes", "morbi", "mus", "nam", "nascetur",
```

```
32        "natoque", "nec", "neque", "netus", "nibh", "nisi", "nisl", "non",
33        "nostra", "nulla", "nullam", "nunc", "odio", "orci", "ornare",
34        "parturient", "pellentesque", "penatibus", "per", "pharetra",
35        "phasellus", "placerat", "platea", "porta", "porttitor", "posuere",
36        "potenti", "praesent", "pretium", "primis", "proin", "pulvinar",
37        "purus", "quam", "quis", "quisque", "rhoncus", "ridiculus",
38        "risus", "rutrum", "sagittis", "sapien", "scelerisque", "sed",
39        "sem", "semper", "senectus", "sit", "sociis", "sociosqu",
40        "sodales", "sollicitudin", "suscipit", "suspendisse", "taciti",
41        "tellus", "tempor", "tempus", "tincidunt", "torquent", "tortor",
42        "tristique", "turpis", "ullamcorper", "ultrices", "ultricies",
43        "urna", "ut", "varius", "vehicula", "vel", "velit", "venenatis",
44        "vestibulum", "vitae", "vivamus", "viverra", "volutpat",
45        "vulputate" };
46    private final String[] punctuation = { ".", "?" };
47    private final String _n = System.getProperty("line.separator");
48    private Random random = new Random();
49
50    public LoremIpsum() {
51    }
52
53    /**
54     * Get a random word
55     */
56    public String randomWord() {
57      return words[random.nextInt(words.length - 1)];
58    }
59
60    /**
61     * Get a random punctuation mark
62     */
63    public String randomPunctuation() {
64      return punctuation[random.nextInt(punctuation.length - 1)];
65    }
66
67    /**
68     * Get a string of words
69     *
70     * @param count
71     *            - the number of words to fetch
72     */
73    public String words(int count) {
74      StringBuilder s = new StringBuilder();
75      while (count-- > 0)
76        s.append(randomWord()).append(" ");
77      return s.toString().trim();
78    }
79
80    /**
81     * Get a sentence fragment
82     */
83    public String sentenceFragment() {
84      return words(random.nextInt(10) + 3);
85    }
86
87    /**
88     * Get a sentence
89     */
90    public String sentence() {
91      // first word
92      String w = randomWord();
93      StringBuilder s = new StringBuilder(w.substring(0, 1).toUpperCase())
```

```
94          .append(w.substring(1)).append(" ");
95      // commas?
96      if (random.nextBoolean()) {
97        int r = random.nextInt(3) + 1;
98        for (int i = 0; i < r; i++)
99          s.append(sentenceFragment()).append(", ");
100     }
101     // last fragment + punctuation
102     return s.append(sentenceFragment()).append(randomPunctuation())
103         .toString();
104   }
105
106   /**
107    * Get multiple sentences
108    *
109    * @param count
110    *            - the number of sentences
111    */
112   public String sentences(int count) {
113     StringBuilder s = new StringBuilder();
114     while (count-- > 0)
115       s.append(sentence()).append("  ");
116     return s.toString().trim();
117   }
118
119   /**
120    * Get a paragraph
121    *
122    * @useStandard - get the standard Lorem Ipsum paragraph?
123    */
124   public String paragraph(boolean useStandard) {
125     return useStandard ? standard : sentences(random.nextInt(3) + 2);
126   }
127
128   public String paragraph() {
129     return paragraph(false);
130   }
131
132   /**
133    * Get multiple paragraphs
134    *
135    * @param count
136    *            - the number of paragraphs
137    * @useStandard - begin with the standard Lorem Ipsum paragraph?
138    */
139   public String paragraphs(int count, boolean useStandard) {
140     StringBuilder s = new StringBuilder();
141     while (count-- > 0) {
142       s.append(paragraph(useStandard)).append(_n).append(_n);
143       useStandard = false;
144     }
145     return s.toString().trim();
146   }
147
148   public String paragraphs(int count) {
149     return paragraphs(count, false);
150   }
151 }
```

Listing A.48: shared.util.LoremIpsum