FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Practical Framework for Byzantine Fault-tolerant Systems

Marshal Plan Scholarship

Project Report

eingereicht von

Adrian Djokic

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

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 is modular by definition.

Having a modular framework to test distributed algorithms enables the use of novel ideas, 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 has the calls to an abstract coding scheme integrated to encode and decode data. Extending the framework with a coding scheme, such as Reed-Solomon, should be a minor task.

Keywords

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

Contents

List of Figures

List of Listings

List of Algorithms

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[8] 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 model and configuration, it is possible to reconstruct the original message from any given storage cluster, that is, by reading from a subset of 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[22]. 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[25].

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 [21] 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 [5][15][13][18][23] and data coding schemes[6][7][10][12][24], 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[20][27][26][28].

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[11]. 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.

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[16].

A distributed system containing nodes which could exhibit 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 faulty processes may 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. 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 $n \geq 3f + 1$.

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[14], security[1], and error-correction[31].

Lately, network coding has been explored for use in distributed storage systems[4], in order to improve the reliability, efficiency, and robustness of such systems[9]. 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[8].

2.3 State-of-the-Art

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 more recently[19][12]. While the performance of a BFT algorithm which employs coding has been evaluated against a "classical" algorithm[18], this has been done on a small scale with few nodes, and there is currently no known work of benchmarks with larger scale systems. In order to gain impartial results from benchmarking various BFT algorithms, tests need to be run on the same platform. Current existing libraries that provide fault-tolerance for distributed systems[2][29] do not employ or enable the use of coding in their infrastructures, and are thus not suited for achieving our goal. Also, apart

[1] in a distributed system, processes run on individual nodes

from the fact that these systems are not being actively maintained[3][30], 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.

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)[17].

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 type of `StoragePackage`, `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.

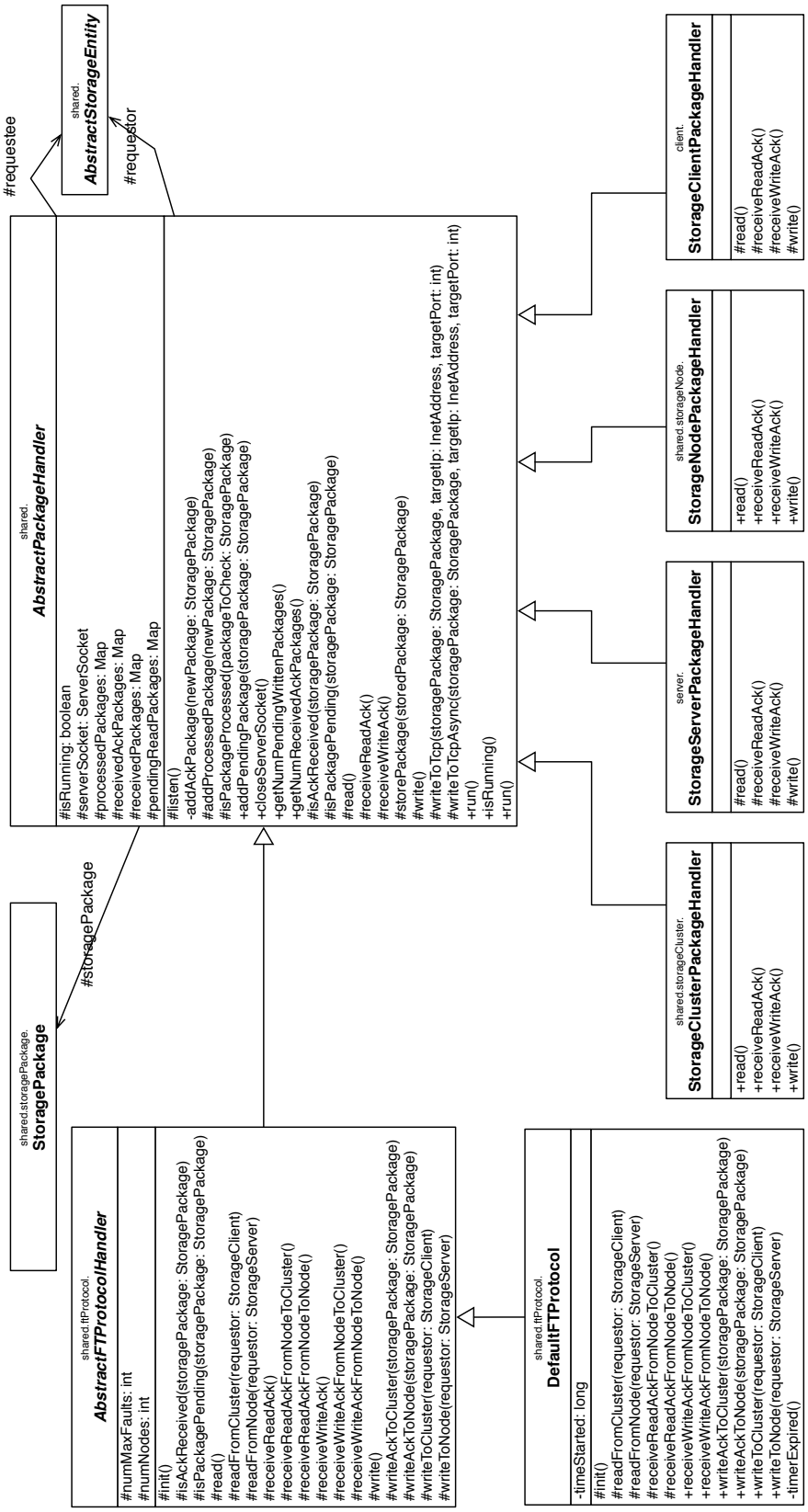

Figure 3.1: Framework class diagram — AbstractPackageHandler

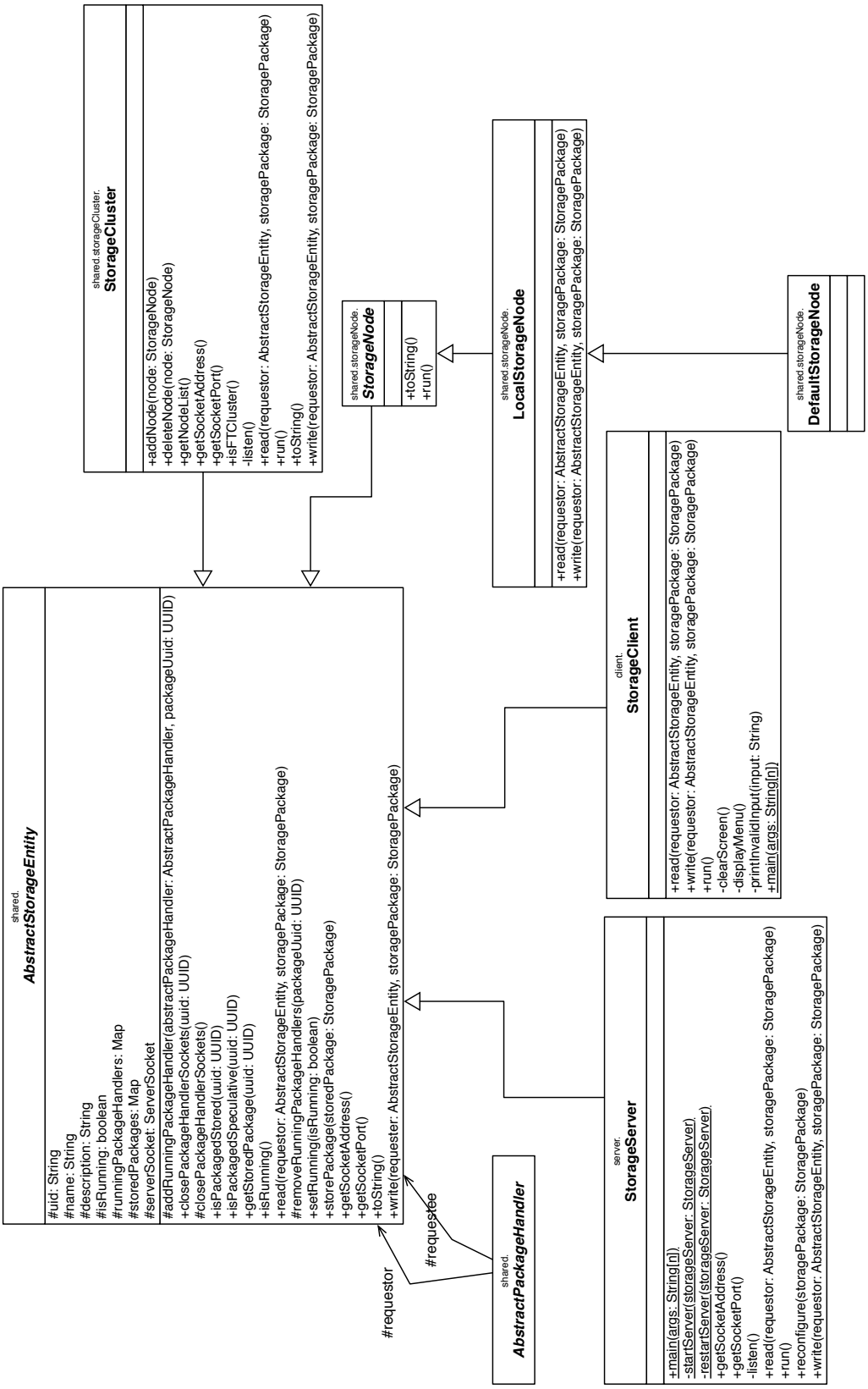

Figure 3.2: Framework class diagram — AbstractStorageEntity

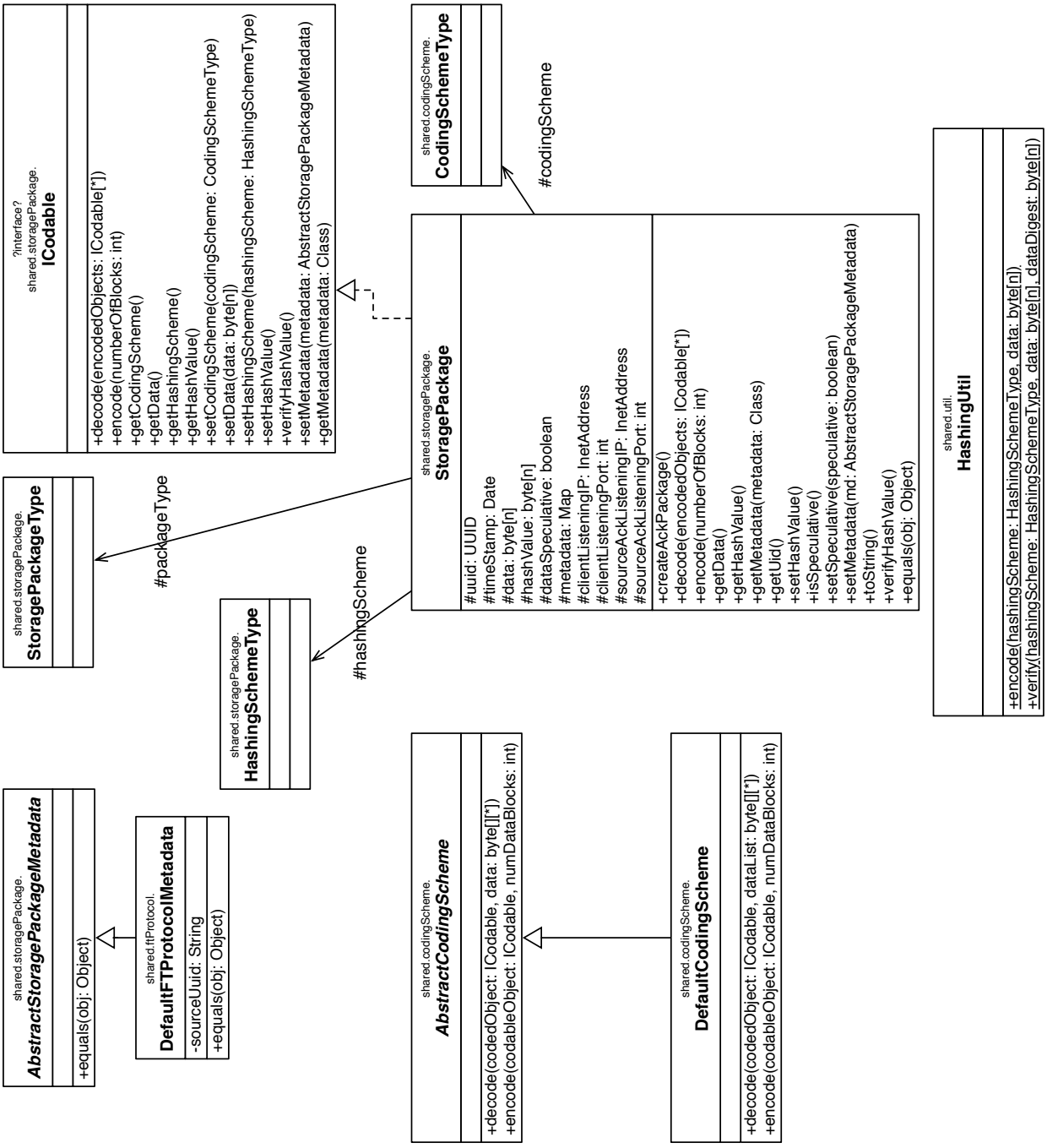

Figure 3.3: Framework class diagram — StoragePackage

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`, since communication between `AbstractStorageEntities` is performed via this 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.

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

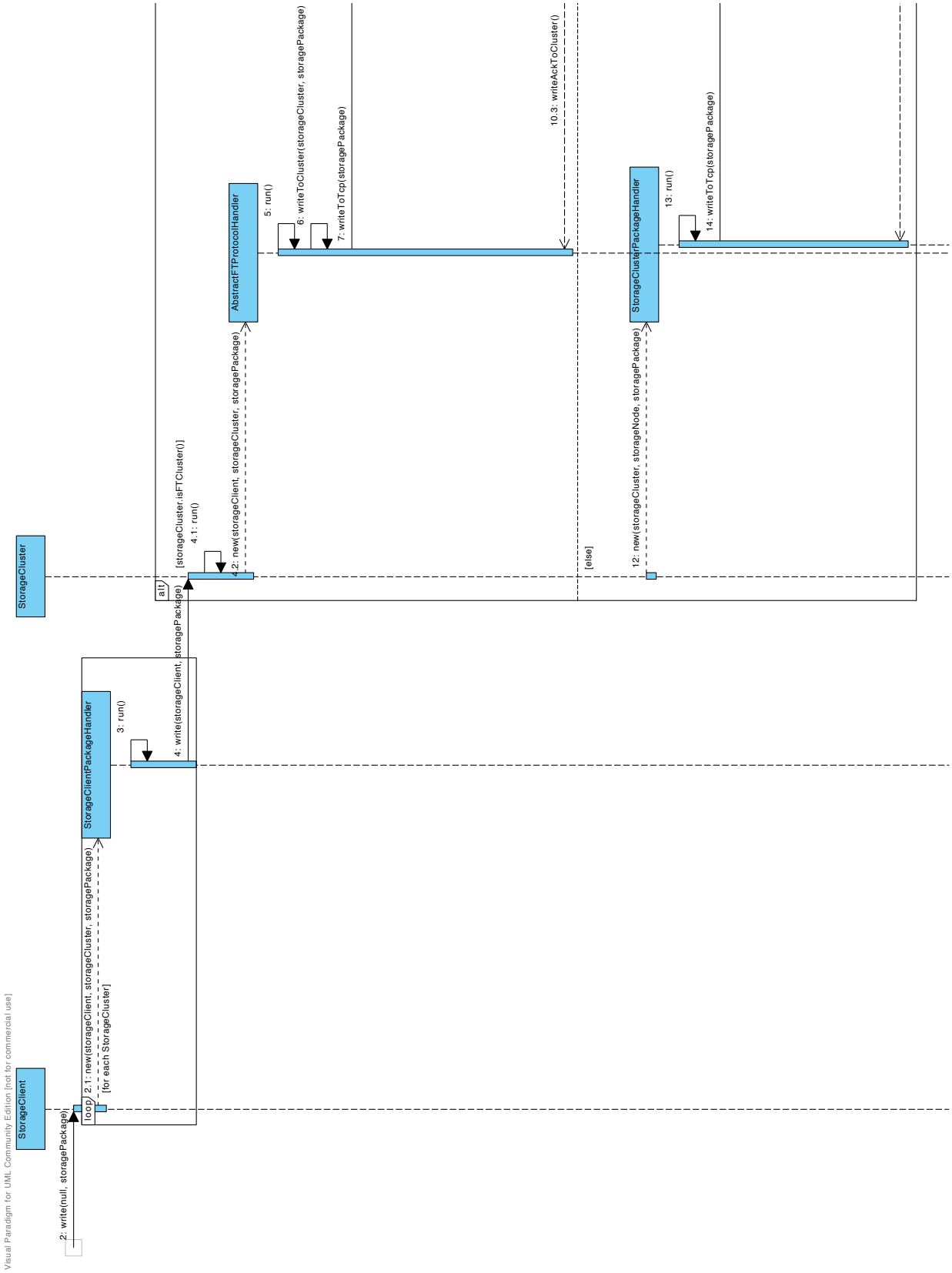

Figure 3.4: Framework sequence diagram — Client-side

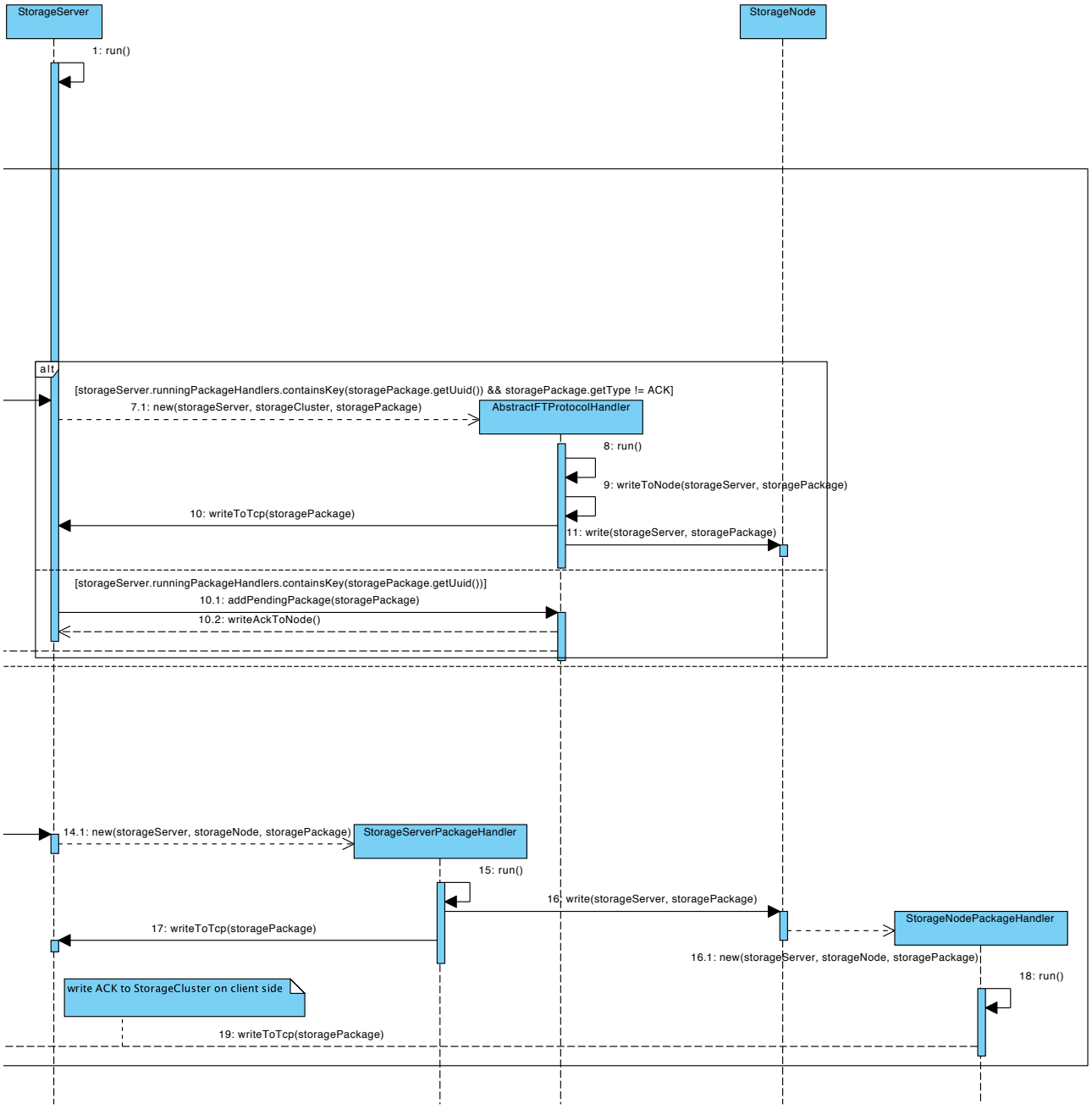

Figure 3.5: Framework sequence diagram — Server-side

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.

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.

4 Algorithms

4.1 DefaultFTProtocol

Following is the default fault-tolerant algorithm implementation, `DefaultFTProtocol`, 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.

$N \geq 3f + 1 = \|\texttt{StorageCluster}\|$, 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.

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`):`
if *new write request received from client* **then**
  code p into $p_0 \dots p_{N-1}$ so that the original data can be recovered by reading $f+1$ different error-free p_i;
  foreach $S_i, i \neq 0$ **do**
    write p_i to S_i;
  end
else write request has already been received from another client, but not yet committed
  respond to new client with previous client's request;
`// each` S_i`:`
foreach $S_j, j \neq i$ **do**
  write p_i to S_j;
end
packages_verified $\leftarrow 0$;
packages_processed $\leftarrow 0$;
verified $\leftarrow false$;
while verified $= false$ **do**
  $p_j \leftarrow$`poll(`*`StoragePackage queue`*`)`;
  if *not* *coding flag set* **then**
    if *hashing flag set* **then**
      if `hash(`p_i`)=hash(`p_j`)` **then**
        packages_verified $\leftarrow$packages_verified $+1$;
    else
      if $p_i = p_j$ **then**
        packages_verified $\leftarrow$packages_verified $+1$;
    packages_processed $\leftarrow$packages_processed $+1$;
    if packages_verified $\geq f+1$ **then**
      verified $\leftarrow true$;
    if packages_processed $\geq f+1$ ***and*** verified $= false$ **then**
      break current loop;
  else
    if $\|$`StoragePackage` *queue*$\| \geq f+1$ ***and*** packages_processed $\leq f$ **then**
      packages_processed $\leftarrow$packages_processed $+1$;
      decode data from queue;
      if *decoding successful* **then**
        verified $\leftarrow true$;
  if (packages_processed $-$ packages_verified) $\geq f+1$ ***and*** verified $= false$ ***or*** `timer_expired()` **then**
    break current loop;
end
repeat
  send ACK to client containing p_i with the full or hash of the (decoded) data;
until *ACK received from client*;wait a defined interval between sending ACKs to client
if *ACK from client matches* p_i **then**
  remove speculative flag from p_i
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`):`
foreach $S_i, i \neq 0$ **do**
  write UUID to S_i;
end
`// each` S_i`:`
if *not* UUID *stored* **then**
  $p_i \leftarrow null$;
  foreach $S_j, j \neq i$ **do**
    request p_j from S_j;
  end
  packages_processed $\leftarrow 0$;
  verified $\leftarrow false$;
  $H \leftarrow$new hash table; `// contains received data (at most` $f+1$ `different keys possible)`
  while verified $= false$ **do**
    $p \leftarrow$`poll(`*StoragePackage queue*`)`;
    if *not* *coding flag set* **then**
      if *hashing flag set* **then**
        if `contains(`H, `hash(`p`))` **then**
          H[`hash(`p`)`] $\leftarrow H$[`hash(`p`)`] $+ 1$;
        else
          H[`hash(`p`)`] $\leftarrow 1$;
        if H[`hash(`p`)`] $\geq f + 1$ **then**
          verified $\leftarrow true$;
      else
        if `contains(`H, p`)` **then**
          $H[p] \leftarrow H[p] + 1$;
        else
          $H[p] \leftarrow 1$;
        if $H[p] \geq f + 1$ **then**
          verified $\leftarrow true$;
      packages_processed $\leftarrow$packages_processed $+1$;
    else
      if $\|$*StoragePackage queue*$\| \geq f + 1$ ***and*** packages_processed $\leq f$ **then**
        packages_processed $\leftarrow$packages_processed $+1$;
        decode data from queue;
        if *decoding successful* **then**
          verified $\leftarrow true$;
    if packages_processed $\geq f + 1$ ***and*** verified $= false$ ***or*** `timer_expired()` **then**
      break current loop;
  end
write p_i to client;

Algorithm 4.4: DefaultFTProtocol server-side read

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[15], 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[18], still needs to be done.

6 Discussion

Even though we have introduced yet[2] another[29] 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.

Bibliography

References

[1] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

[26] 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/.

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

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

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

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