CS347 Project Report

Fault Tolerance in Distributed Systems Using Fused Data Structures

Andreas Petrou, Ben Thompson, Callum Bloomfield and Krishan Mistry
Department of Computer Science
University of Warwick

Abstract

Conventional fault-tolerance for data in distributed architectures requires active replication of information across multiple locations. This is a resource intensive process, especially in terms of memory usage. An alternative approach is to maintain fault-tolerant data within a *fusible data structure* - in theory providing the same levels of reliability whilst minimising memory usage. Using techniques outlined by Balasubramanian and Garg [1], we present a modern fused data structures package written in Python using strongly-typed communication protocols. We further draw comparisons between the two implementations, and perform analysis to prove the efficacy of tolerating crash faults.

1 Introduction

In the digital age, there is exponential growth of information being created, stored, used, shared and archived. There is a wide spectrum of applications from which data is generated, including digital multimedia generated by individuals, to large data lakes generated for business applications. Whilst the two couldn't differ more, data is only useful whilst data availability and data integrity is maintained - hence the requirement for a meticulous data backup policy. Data loss has the potential to have far-reaching consequences - it is hard to predict when such spontaneous events will happen. However, we know that data loss can happen in two forms: crash faults, such as server failure; and Byzantine faults, where incorrect information is returned by the system.

Within the field of data storage, coding theory has been applied to obtain high levels of reliability and even performance improvements from parallelising access. However, in modern applications where the scale of data far exceeds the available size of a single hardware storage device, distributed systems are employed. The conventional approach to maintain fault-tolerant data in distributed systems is through active data replication. In the event of primary instance failure, a failover is performed to use one of f replicas that are made - with the failed structure being regenerated. The likelihood of a fault occurring is an important aspect to consider when making decisions on how many replicas, f, to produce. In real world usage, when there is a low likelihood of failure, active replication is not efficient in terms of memory usage - if each replica is a mirror copy the storage required is $f + 1$ times the size of the data being stored. As a result of more storage usage, more hardware storage devices are needed, resulting in increased power consumption.

A proposed alternative with significantly less storage requirements for the replicas, through the use of coding theory, are the use of *fusible data structures*. The purpose of this report is to provide an overview of how crash faults are tolerated within a distributed system using fused backups. In addition, the paper also discusses our modern Python implementation of fused data structures using strongly-typed gRPC communication streams. The abstract data types implemented are lists, hash maps, and queues.

The following section provides a concise insight into the background knowledge required to understand how *fused data structures* can be implemented. This is followed by design choices and a discussion of our implementation.

In summary, our main contribution is the development of a functional fused data structures package written in Python conforming to modern standards.

2 Research

The concept of 'fusible data structures' was first introduced in a research paper published in 2007 by Garg and Ogale [4]. They exhibited the means to create fused data structures for several abstract data types including arrays, stacks, and queues through the use of the exclusive disjunction (`XOR`) operation. Their research showed that fused backups managed to achieve space optimisation by a factor of more than $k/2$ where k is the number of lock servers.

Further research has shown that while active data replication requires nf backup structures to tolerate f crash faults, fused backups only require f backup structures to tolerate the same number of crash faults. In addition, it has been proven that fused data structures can save memory when dealing with Byzantine faults, these are faults where corrupt information is returned by servers. While replication can deal with f Byzantine faults by having $2nf$ backup structures, fused structures can tolerate f Byzantine faults with an additional $nf + f$ backups [1].

For creating fused backups, Cauchy Reed-Solomon and Vandermonde Reed-Solomon error correcting codes can be used for efficient recovery of data [13]. Alternate techniques for fusion include the use of Luby transform codes [1]. An analysis of this technique showed a decrease in time required to recover from faults; in addition, the codes offer enhanced scalability and reliability [15]. As demonstrated by Luby, the Luby transform code 'quadratic time implementations are faster in practice' than Reed-Solomon codes [9].

Additional studies have shown the space required by fusion is at most 83% of that used in active replication. However, the main drawback comes with recovery times being up to three orders of magnitude more expensive [3].

Fused data structures have also been used for dealing with crash faults in deterministic finite state machines - with the same benefit of a decrease in space requirements. A polynomial-time solution for finding the minimal set of of fusible machines that can tolerate a fixed number of crash faults was found [11].

For creating our own implementation of a fault-tolerant distributed system, we decided to use Python. When Balasubramanian and Garg published their proof-of-concept implementation in 2013 [17], Java was the more popular language. At the time of writing, Python is ranked more popular, and there is no fused data structures implementation on the Python Package Index (PyPI). Thus, we intend to design and develop our solution to be production-ready - ready for release as a package for others to use.

3 Requirement Analysis

Prior to designing and implementing the system, it is crucial to identify measurable requirements. Requirements are categorised as being either functional or non-functional, in addition a priority has been given.

Functional Requirements

- User must be able to create primary structures such as maps and lists. (Must Have)
- User must be able to add or remove elements from primary structures. (Must Have)
- User must be able to retrieve data from primary structures. (Must Have)
- Primary structures must use fused structures for backups of data. (Must Have)
- Primary structures must be able to establish connections with fused structures for communication. (Must Have)
- Fused structures must be updated accordingly when a change occurs in primary structures. (Must Have)
- Fusion must use Reed-Solomon erasure codes for encoding primary data. (Should Have)

Non-Functional Requirements

- Given a system of n structures, we must tolerate f crash faults, where $n > f$. (Must Have)
- Given a system of n structures, we must tolerate f Byzantine faults where $n > f$. (Could Have)
- System must be operational 24/7. (Could Have)
- Recovering from faults should take no longer than 5 hours. (Should Have)
- Create a library of our implementation so that other developers can benefit from our work (Could Have)

4 Design

Several questions were raised when developing a fault-tolerant system consisting of several servers, each hosting some data structures. For the following sections, primary structures refer to the original version of data structure, while fused data structures refer to backup versions of the original data structure.

What data structures to use?

The data structures we think are most important to implement are lists and hash-maps. Other structures to continue would include queues, stacks, and tree-based structures.

What operations can be performed on the data structures?

The operations available on each data structure will be different - whilst values in a list can be added and removed at any index, queues need to only provide operations to *enqueue* and *dequeue*.

What type of data will need to be stored?

Even though Python is a dynamically typed language and does not require the declaration of variable types, all data structures used during testing will be integers. Nevertheless, our solution should attempt to work with other data types.

How can we fuse multiple data structures into one backup structure?

An initial attempt of fusing data structures presented an algorithm that used the `XOR` operation to merge data from multiple structures into a single backup structure [4]. Nevertheless, one of the main drawbacks of data structure fusion is its time to recover from errors. It is not wise to use this technique for performing backup operations when a system is prone to crash faults as the overhead in recovering data can be devastating [4]. A better approach for merging data structures demonstrated that the time it takes to recover data could be much faster when Reed-Solomon erasure codes are used.

How many data structures should be fused into a single backup structure?

This ultimately depends on how many primary structures exist, what the server configuration is, and how many faults are to be tolerated.

How can we recover our data when primary data structures crash?

In the case when primary data structures crash, the client should make requests to the appropriate fused structures - then perform the reverse data encoding operations.

Will we need both primary and fused data structures to recover our data in case of a fault, or will the fused backups suffice?

Assuming a system is designed to tolerate f faults, where none of the fused structures fail, data should be recoverable.

4.1 Reed-Solomon codes

Consider there exists n data structures $\{D_1, D_2, \ldots, D_n\}$, and a set of m checksum data structures $\{C_1, C_2, \ldots, C_m\}$. The checksum data structures contain data calculated from the contents of the primary data structures. The aim is that in case of a failure of m primary structures, the contents of the checksum structures will be used to recover lost data. To make the system fault-tolerant, a function F is required such that the contents of the fused structures will be calculated as follows:

$$c_i = F_i(d_1, d_2, d_3, ..., d_l)$$

Every time a data element is updated in a primary structure D_j from d_j to d'_j, a function G handles the update on the fused structure:

$$c'_i = G_{i,j}(d_j, d'_j, c_i)$$

If data from the primary structures is transformed into a matrix P, and the data from the checksum structures into a matrix C, then if all the functions F_i are mapped as rows of a Vandermonde matrix F of dimensions $m \times n$ then [7]:

$$FP = C$$

When a data element changes from d_j to d'_j, the fused structure can be updated by defining $G_{i,j}$ as follows [13]:

$$c'_i = G_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j)$$

For data recovery, we need two additional metrics:

$$A = \begin{bmatrix} I \\ F \end{bmatrix} \qquad E = \begin{bmatrix} D \\ C \end{bmatrix}$$

When a data structure fails, the rows from both A and E that correspond to the faulty data structure are deleted - forming two new matrices: A' and E'. The missing data can be recovered by solving for D in the equation $A'D = E'$. The faulty fused structures can be recovered by using matrix F.

The standard method of solving a system of linear equations, Gaussian elimination, is not feasible in certain situations [6]. As a result, Galois fields of field length 2^w, where w is the length of the words being used, $GF(2^w)$ are required for addition, and multiplication operations [13].

In the two equations illustrated above, the functions F and G, which are used for adding and updating values in the fused data structures, make use of Vandermonde matrices.

5 System Design & Implementation

The distributed system can be deconstructed into numerous subsystems - each handling different functions. As displayed in Figure 2, the system operates on a client-server model; the client offering an interface with data stored on primary data structures stored on primary servers. The primary structures then communicate with each of the fused structures in order to create backups of the original data. The only operations a client handles with fused structures directly is for the purposes of recovery.

Figure 3 depicts a larger system with *clusters*. A cluster is a wrapper around an instance of a set of primary data structures; it allows each server to handle multiple data structures independently of each other.

5.1 Communication Protocols

To facilitate communication between client-server and server-server, we opted to use gRPC - a modern open source Remote Procedure Call framework. We defined a strongly typed communication protocol using version 3 of the protocol buffers syntax [5]. Due to the differing needs of the primary and fused servers, and their respective structures, we define a separate service for each. Our protocol definitions are detailed in Figure 6.

5.2 Client

To the end user, the client interface provides methods to create a fault-tolerant structure with a defined number of primary structures and number of faults tolerable. This interface provides methods to interact with the primary data structures, whilst also providing logic to recover data in case of primary structure crash failure.

5.3 Primary Servers & Primary Structures

Primary servers are capable of handling requests made via the gRPC service `PrimaryDataStructure`. On creation of a primary structure, independent connections are made with the fused structure servers. Primary structures can handle methods synonymous with the data structure it has been defined with.

Data is stored within `PrimaryNode` elements within an instance of the data structure. To maintain efficient fused data structures, blank spaces in fused data structures must be minimised. Thus, an ordering of elements as they would appear in the fused data structures, a stack of `PrimaryAuxNode` elements, is stored within a doubly linked list. `PrimaryNode` and `PrimaryAuxNode` have bi-directional pointers to each other. This obfuscation allows for efficient storage of elements when it comes to data stored on fused structures.

5.4 Fused Servers & Fused Structures

Fused servers are capable of handling requests made via the gRPC service `FusedDataStructure`. Each fused structure can handle multiple primary structures within a *cluster*.

Fused data is stored within `FusedNode` elements in a doubly linked list structure. For each primary structure that is attached to the fused structure, an ordering of elements is maintained synonymous to the data stored in primary structures. Instead of storing data, a pointer is stored to the `FusedNode` element that contains the data.

The ordering of elements for each primary structure within the `FusedNode` doubly linked list is guaranteed to mirror that of the doubly linked list of `PrimaryAuxNode` in primary structures.

5.5 Clusters

Our system is logically partitioned into groups of structures across a set of nodes, which we refer to as *clusters*. The following principles govern this abstraction:

- A cluster is formed from an arbitrary number of primary and fused servers.
- Each primary or fused server can store data structures originating from any number of clusters.
- A primary or fused data structure belongs to exactly one cluster.
- Fused data structures in a cluster will record recovery codes for primary data structure in the same cluster only.

5.6 Reed-Solomon Codes

The proof-of-concept implementation presented in the original paper by Balasubramanian and Garg opted to use the `Jerasure` [12] C library for Reed-Solomon coding, justified in part by the performance advantages that this would have over an equivalent implementation in Java [10]. In a similar manner, Python is also slower than C for most workloads [19]. Writing custom implementations for algorithms that already have strong open-source alternatives is often considered bad practice within software engineering [2], so we opted to use the same `Jerasure` library in our implementation, allowing us to realize similar performance gains.

To facilitate this, it was necessary for us to write our own Python *extension module* package for `Jerasure`. This was composed of two parts, the first of which being a short Python script to instruct the interpreter on how to load the module. This was paired with a C program, which was responsible for marshalling and unmarshalling data into formats expected by each language, and passing computations to `Jerasure`. To illustrate the function of this code, the header files for the C aspect of the code are replicated below:

```
PyObject* recover_data(PyObject*, PyObject*);
PyObject* calculate_rs_code(PyObject*, PyObject*);
PyObject* calculate_rs_matrix(PyObject*, PyObject*);
```

Figure 1: `python_jerasure.h` header file

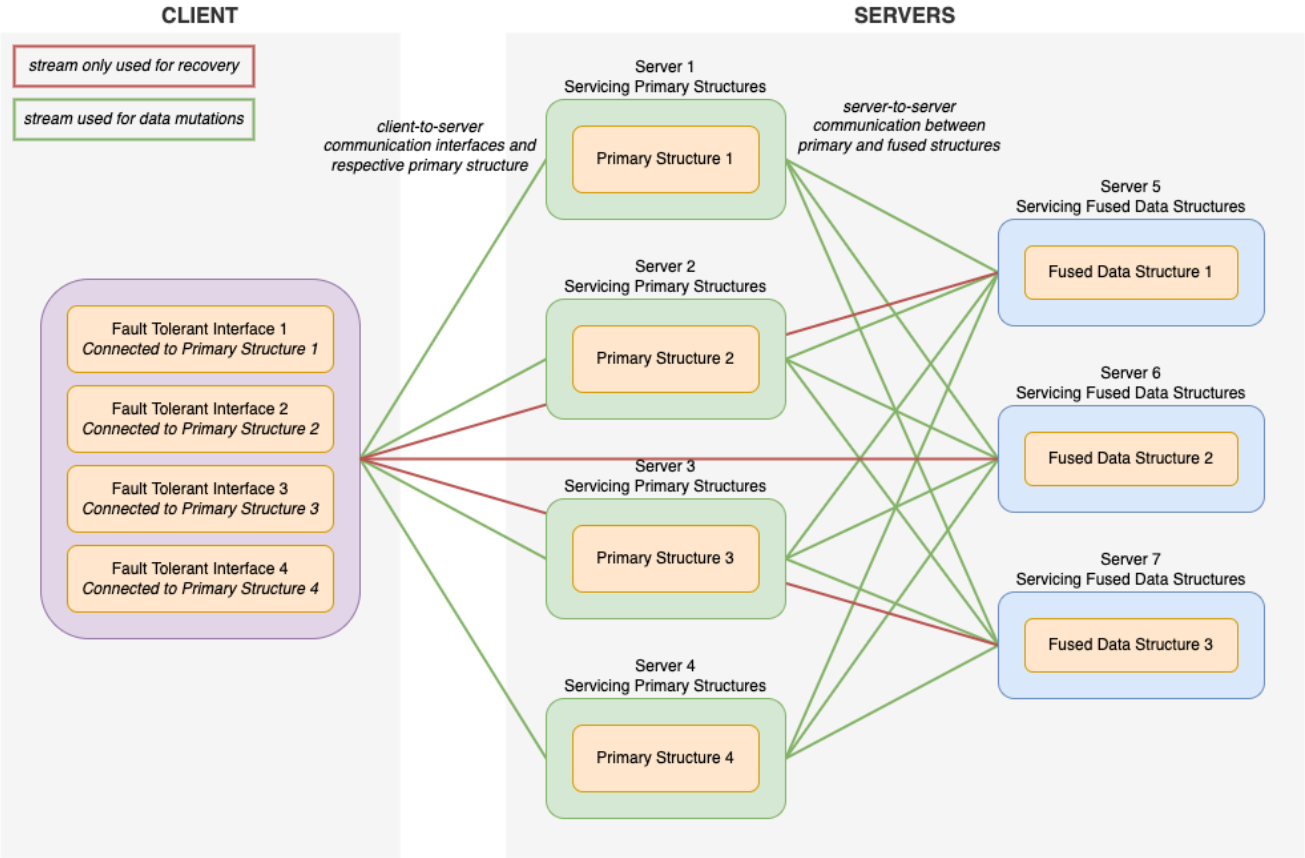

Figure 2: Diagram representing the client-server model implemented with fused data structures. This specific example has 4 primary structures which can tolerate up to 3 crash faults.

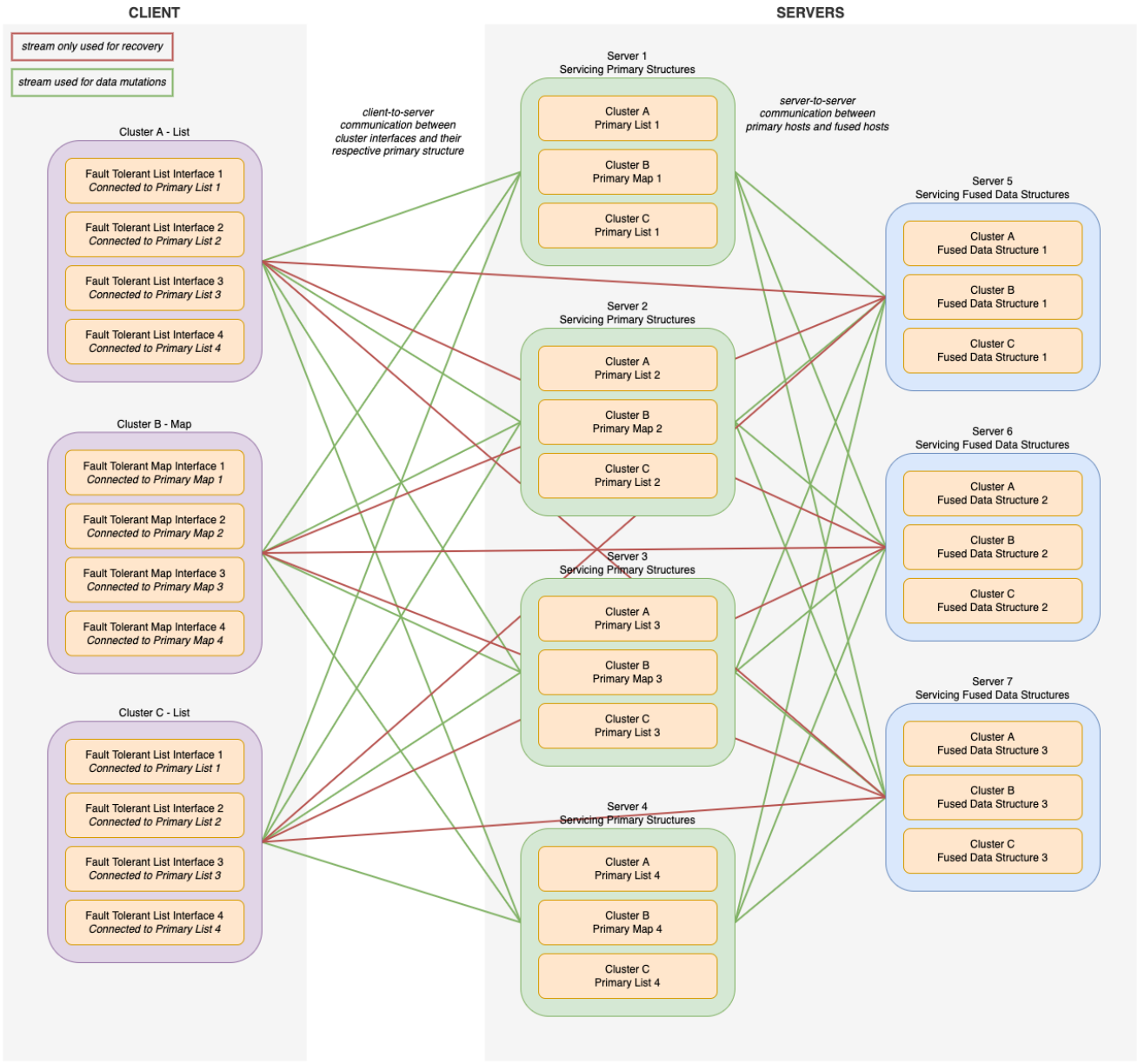

Figure 3: Diagram representing a client-server model with different data structures in use via the form of clusters. This specific example has 4 primary structures in each cluster which can tolerate up to 3 crash faults. Each cluster forms its own fused data structure on each server

6 Testing

In order to validate the performance of our system, we adopted a black-box testing approach. Using the API exposed by our package, we created a system comprising of one logical cluster for each structure type, which in turn was composed of three primary and three backup structures. We then checked that the program exhibited the expected behaviour for a series of test cases. The source data for these test cases are listed below for the `map` and `list` structure types:

Cluster 1		
List 1	List 2	List 3
Add **123** at index **0**	Add **456** at index **0**	Add **789** at index **0**
Add **456** at index **1**	Add **420** at index **0**	Add **126** at index **0**
Add **125** at index **1**	Add **420** at index **0**	Add **127** at index **0**
Add **256** at index **3**	-	-

Table 1: Test data for `list` data structure

Cluster 2		
Map 1	Map 2	Map 3
Add **1234** with key **100**	Add **6** with key **1**	Add **1** with key **5**
Add **9600** with key **200**	Add **42** with key **2**	Add **2** with key **6**
Add **7121** with key **100**	Add **96** with key **57**	Add **4** with key **7**
Add **420** with key **579**	Add **12** with key **59**	Add **3** with key **8**

Table 2: Test data for `map` data structure

The expected state of the data structures in the system after each successive operation is listed below; subsequent test cases will make reference to these to determine pass or failure:

List 1	→	[123, 456]	→	[123, 125, 456]	→	**[123, 125, 456, 256]**
List 2	→	[420, 456]			→	**[420, 420, 456]**
List 3	→	[126, 789]			→	**[126, 127, 789]**
Map 1	→	{100:1234, 200:9600}	→	{100:7121, 200:9600}	→	**{100:7121, 200:9600, 579:420}**
Map 2	→	{1:6, 2:42}	→	{1:6, 2:42, 57:96}	→	**{1:6, 2:42, 57:96, 59:12}**
Map 3	→	{5:1, 6:2}	→	{5:1, 6:2, 7:4}	→	**{5:1, 6:2, 7:4, 8:3}**

After this test data had been communicated to the nodes, we executed the following test cases to ascertain the validity of the insertion and retrieval operations:

Test Number	**Test Case**	**Expected Result**
1	Access index 0 of List 1	123
2	Access index 2 of List 1	125
3	Access index 2 of List 3	789
4	Access index 1 of List 2	420
5	Access index 4 of List 2	*Index Error*
6	Get value with key 100 from map 1	7121
7	Get value with key 57 from map 2	96
8	Get value with key 8 from map 3	3
9	Get value with key 7 from map 1	*Key Error*

Table 3: Tests for retrieving elements from lists and maps

The expected results listed in **Table 3** are based on the data defined in **Table 1** and **Table 2**. After confirming that this first set of test cases passed, we executed a similar series of cases to determine the validity of removal operations, as listed below:

Test Number	**Test Case**
1	Remove index 0 from List 1
2	Remove index 2 from List 2
3	Remove index 1 from List 3
4	Remove index 1 from List 3
5	Remove index 3 from List 1
6	Remove key 100 from Map 1
7	Remove key 579 from Map 1
8	Remove key 7 from Map 3
9	Remove key 100 from Map 1

Table 4: Tests for removing elements from lists and maps

The state of the data structures in the system after each of these test cases was then analysed, and confirmed to match that which was expected, as listed below:

List 1	[123, 125, 456, 256]			→	[125, 456, 256]	→ **Index Error**
List 2	[420, 420, 456]					→ **[420, 420]**
List 3	[126, 127, 789]			→	[126, 789]	→ **[126]**
Map 1	{100:7121, 200:9600, 579:420}	→	{200:9600, 579:420}	→	{200:9600}	→ **Key Error**
Map 2						→ **{1:6, 2:42, 57:96, 59:12}**
Map 3				→	{5:1, 6:2, 7:4, 8:3}	→ **{5:1, 6:2, 8:3}**

Finally, we tested the ability of the system to recover from crash faults at the primary data nodes. To do so, we simulated the maximum number of primary structure faults that the test system supported, and compared the results to the expected test data in **Table 1**. The successful outcome of this test is evidenced through the following snippet from the system's logging output, which shows the recovery and auxiliary codes used to reconstruct the data:

```
Attempting to recreate original data for 3 faults on data points: {0, 1, 2}
Available primary data: {
'370f93eb-dfdd-4a9b-ba08-0178714d6c5d': [],
'f64506ce-3d27-4897-a73e-210bcaa3a27a': [],
'bb9317b2-3ce7-4f11-8129-896a402342e6': []}
Available auxiliary data: [[0, 1, 0], [1, 0, -1], [2, -1, -1]]
Available RS-encoded data: [(678, 995, 789), (41143, 37571, 789), (61441, 61955, 789)]

Lists which were restored:
Primary structure `0` with structure identifier `370f93eb-dfdd-4a9b-ba08-0178714d6c5d` was recovered: [123, 456, 789]
Primary structure `1` with structure identifier `f64506ce-3d27-4897-a73e-210bcaa3a27a` was recovered: [555, 456]
Primary structure `2` with structure identifier `bb9317b2-3ce7-4f11-8129-896a402342e6` was recovered: [789]
Data restoration complete...
```

Figure 4: Logging outputs from the datastructure client during recovery from a crash fault.

6.1 Unit Testing

We used the `unittest` Python testing framework to automate the process of validating results calculated during data fusion and repair. Testability is an important software quality measurement [8], and hence unit testing is in keeping with our goal of producing an implementation that is suitable for production environments.

Testing this aspect of the application in an automated manner was particularly pertinent, as it covered the boundary between our high-level Python code, and lower level C code, which can be considered more error-prone. Without these tests in place, regressions introduced through changes in the C code could go unnoticed, which would ultimately lead to loss or corruption of data stored in the fused structures. Method signatures for these tests are replicated below to illustrate their purpose:

```
test_calculate_rs_matrix_valid();
test_calculate_rs_matrix_invalid();
test_calculate_rs_code_valid();
test_calculate_rs_code_invalid();
test_recover_data_valid();
test_recover_data_invalid();
```

Figure 5: Automated test case signature for our `python_jerasure` package.

7 Evaluation

Unlike its proof-of-concept Java counterpart, our implementation is production-ready, given its adherence to all modern standards and protocols. We expose a simple API to the user, requiring specification of only the desired structure type, number of tolerable faults, and the number of primaries data nodes. All other aspects, such as allocation of ports to nodes, is abstracted away. Moreover, the process of adding, retrieving and removing data from the system is straightforward, through a simple method call on a client object. Although it could be argued that the interpreted nature of Python makes it a sub-optimal candidate for large scalable systems, its increasing use in high-traffic websites such as YouTube [18] largely disprove this, reinforcing the potential applications of our system.

The claim that our system is production-ready is supported by the following features of our system:

- Inclusion of unit tests (which the original Java implementation failed to do).
- Using Poetry, a modern packaging and dependency management system.
- Scanning our code with Bandit, an up to date security linter to find common security issues within our code.
- Following the Python style guide [16] to ensure our code is consistent and readable.
- Producing an example application that consumes our library.

8 Conclusion

Working from the principles described in 'Fault Tolerance in Distributed Systems Using Fused Data Structures', by Bharath Balasubramanian and Vijay K. Garg [1], we have successfully implemented a fault-tolerant distributed system that utilises a mixture of primary and fused data structures. As confirmed in our evaluation, we have produced a system that meets all of our primary requirements, in a way that adheres to modern programming standards. We have also thoroughly tested our implementation, and thus deem it suitable for publication to the Python Package Index [14]. As this is the first known fused data structure implementation in Python, other developers will now have that opportunity to experiment with fusion-based approaches in their systems.

9 Future Work

A future area of research could be to quantitatively investigate the extent to which our implementation scales in a cloud-based environment with many thousands of nodes, in a manner similar to Boyd [3]. Moreover, implementation of ordered data-structures such as tree maps could generate additional real-world applications of our system.

Another improvement could be the implementation of the alternative technique suggested by Rajkumar, K. and Swaminathan, P [15], which is claimed to be more efficient than the RS code technique. This is beyond the current scope of this project however, as doing so would require writing a library similar to `Jerasure`, given that there is currently no high-quality open source implementation of LT Codes in C/C++.

References

[1] Bharath Balasubramanian and Vijay K. Garg.
"Fault Tolerance in Distributed Systems Using Fused Data Structures".
In: *IEEE Transactions on Parallel and Distributed Systems* 24.4 (2013), pp. 701–715.
DOI: 10.1109/TPDS.2012.96.

[2] Jenny Blessing, Michael A Specter, and Daniel J Weitzner. "You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries".
In: *arXiv preprint arXiv:2107.04940* (2021).

[3] Jeremy J Boyd et al. "Active replication vs. fusion as fault tolerance mechanisms". PhD thesis. 2016.

[4] Vijay K Garg and Vinit Ogale. "Fusible Data Structures for Fault-Tolerance".
In: *Proceedings of the 27th International Conference on Distributed Computing Systems*. 2007, p. 20.

[5] Google. "Protocol Buffers Version 3 Language Specification". In: ().

[6] Nicholas J. Higham. "Gaussian elimination". In: *WIREs Computational Statistics* 3.3 (2011), pp. 230–238.
DOI: 10.1002/wics.164.

[7] Dan Kalman. "The Generalized Vandermonde Matrix". In: *Mathematics Magazine* 57.1 (1984), pp. 15–21.
DOI: 10.1080/0025570X.1984.11977069.

[8] Claus Klammer and Albin Kern. "Writing unit tests: It's now or never!" In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.
IEEE. 2015, pp. 1–4.

[9] Michael Luby. "LT codes".
In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*
IEEE Comput. Soc. DOI: 10.1109/sfcs.2002.1181950.

[10] Gennadiy P Nikishkov, Yu G Nikishkov, and Vladimir V Savchenko.
"Comparison of C and Java performance in finite element computations".
In: *Computers & structures* 81.24-25 (2003), pp. 2401–2408.

[11] Vinit Ogale, Bharath Balasubramanian, and Vijay K. Garg.
"A fusion-based approach for tolerating faults in finite state machines".
In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–11.
DOI: 10.1109/IPDPS.2009.5161018.

[12] James S Plank, Scott Simmerman, and Catherine D Schuman.
"Jerasure: A library in C/C++ facilitating erasure coding for storage applications Version 1.2".
In: *University of Tennessee, Tech. Rep. CS-08-627* 23 (2008).

[13] James S. Plank. "A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems".
In: *Software: Practice and Experience* 27.9 (1997), pp. 995–1012.
DOI: 10.1002/(SICI)1097-024X(199709)27:9<995::AID-SPE111>3.0.CO;2-6.

[14] "PyPI: The Python Package Index". In: (). URL: https://pypi.org.

[15] K. Rajkumar and Pooja Swaminathan.
"Fault tolerance in distributed systems using fused data structures with the help of LT codes".
In: *International Journal of Advanced Intelligence Paradigms* 8 (2016-01), p. 183.
DOI: 10.1504/IJAIP.2016.075726.

[16] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *Style Guide for Python Code*. PEP 8. 2001.
URL: https://www.python.org/dev/peps/pep-0008/.

[17] "TIOBE Index for March 2022". In: *TIOBE* (2022). URL: https://www.tiobe.com/tiobe-index/.

[18] "Youtube Architecture - High Scalability". In: ().
URL: http://highscalability.com/blog/2008/3/12/youtube-architecture.html.

[19] Farzeen Zehra et al. "Comparative Analysis of C++ and Python in Terms of Memory and Time".
In: (2020-12). DOI: 10.20944/preprints202012.0516.v1.
URL: https://doi.org/10.20944%2Fpreprints202012.0516.v1.

A Appendices

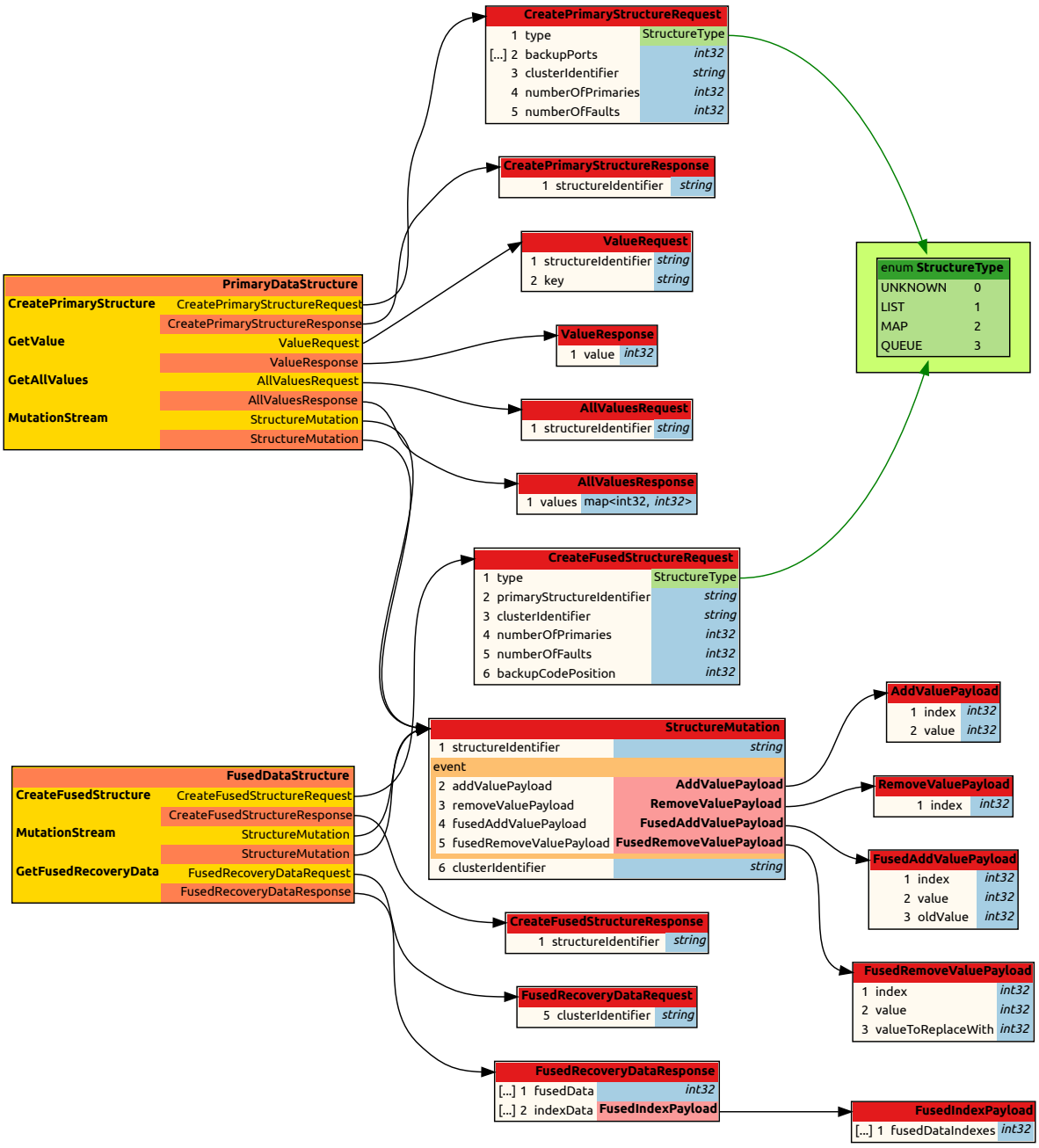

Figure 6: Protocol Definitions