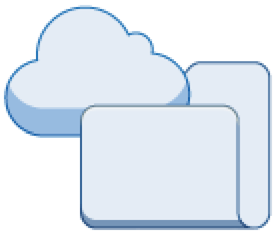


Resilient File Survivability in Peer-to-Peer Networks based on Stochastic Swarm Guidance

Francisco Teixeira de Barros

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Daniel de Matos Silvestre
Prof. Carlos Jorge Ferreira Silvestre

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Daniel de Matos Silvestre
Members of the Committee:
Prof. João Pedro Castilho Pereira Santos Gomes

January 2021

Acknowledgments

First and foremost, I want to express my gratitude to Daniel Silvestre and Carlos Silvestre for accepting me as their apprentice for the writing of my master's dissertation. In particular, I would like to thank Daniel for his constant help and guidance ever since the beginning of my research work. The path has not always been easy, but we completed it. The feedback I got from both coordinators improved my scientific foundations and the way I express my ideas through writing. I am sure that all these traits will be of utmost use in my professional life in the years to come.

Secondly, I would like to pay my special regards to Professor João Pedro Gomes. He was part of the committee that evaluated my midterm. He was courteous, respectful, and provided much needed constructive feedback. Above all, he reminded me to stay on point and that being overzealous can be contra-productive.

The financial sponsorship granted to me by Lisbon's Institute for Systems and Robotics for the participation in the project UID/EEA/50009/2019 - 1801P.00920.1.02 DSOR was also sincerely appreciated. Participating in a non-curricular activity made me more responsible and motivated me to adopt the project's topic as my master's dissertation.

I thank my buddies Diogo Vilela and Rafael Ribeiro, for all the emotional support they gave me concerning my academic career. I also thank them for the many discussions that regard the thesis project; they helped me identify potential errors. I will treasure all these contributions and, of course, the good memories we had together. I acknowledge Ana Figueiredo, an old friend who motivated me to enroll in the course in the first place. Without her motivation, it is possible that, to this day, I would not have even tried.

I wish to express my most profound appreciation to my friend and companion, Sofia Santos. She makes me happy and encourages me to stay motivated for the challenges I face daily. The positive mindset she gifts me with is invaluable. Throughout the thesis, she reminded me I needed to work and pursue my goals, especially when I felt lazy. Thank you.

I praise my parents, Helena and Juvenal Barros, for their patience and loving care and also for providing me the opportunity to enroll in this university degree. Without their continuous assistance, provided in many forms, carrying it to the end would have been way harder, perhaps even impossible.

Abstract

With the growing adoption of technological solutions by governments, companies, and individuals, cloud storage and related services have become desirable alternatives to safeguard important files. Traditional approaches range from: i) centralized architectures, where multiple nodes continuously report to monitoring highly reliable servers; all the way to ii) fully decentralized unstructured Peer-to-Peer (P2P) networks, in which nodes gossip user queries to find and store their items. All paradigms use supplementary techniques that improve the robustness of the system, performance, or resource consumption, including random-walks, periodic acquaintance exchange, proactive and reactive replication, message compression, and others. However, these techniques require complicated algorithms that are hard to validate in order to guarantee that bugs do not compromise the durability of the files.

In this thesis, we propose the use of Probabilistic Swarm Guidance (PSG) algorithms, typically used in robotics to control a formation of robots, to increase the reliability of a system and the durability of the stored files. PSG algorithms are easy-to-implement and offer self-healing properties. Moreover, its convergence rate is theoretically proven through the use of the Perron Frobenius theorem for Markov chains (MCs) and can be optimized. This thesis also reports the creation of a custom simulator where we test MCs generated with different procedures. Through extensive simulation testing, we investigate how PSG behaves in practice in comparison with the ideal scenario. Moreover, we compare the proposed P2P-based Distributed Backup System (DBS) against Hadoop Distributed File System (HDFS) and we outperform it under specific conditions. We conclude that PSG is a viable alternative to existing approaches, but further research is required to determine if it is desirable.

Keywords

File Durability; Markov chains; Peer-to-Peer Storage; Swarm Guidance;

Resumo

Com a crescente adoção de soluções TI por parte de organizações, o armazenamento em nuvem e serviços relacionados tornaram-se alternativas desejáveis para proteger ficheiros críticos. As abordagens tradicionais variam de: i) arquiteturas centralizadas, onde vários nós reportam, continuamente, a servidores de monitorização altamente confiáveis; até ii) redes Peer-to-Peer (P2P) não estruturadas e descentralizadas, nas quais os nós fofocam sobre as consultas dos usuários para encontrar e armazenar os seus itens. Todos os paradigmas usam técnicas complementares que melhoram a robustez do sistema, desempenho ou consumo de recursos. No entanto, essas técnicas requerem algoritmos complicados e difíceis de validar para garantir que os bugs não comprometam a durabilidade dos arquivos.

Nesta tese, propomos o uso de algoritmos Probabilistic Swarm Guidance (PSG), tipicamente usados em robótica para controlar a formação de robôs, de modo a aumentar a confiabilidade de um sistema e a durabilidade dos ficheiros armazenados. Os algoritmos PSG são fáceis de implementar e oferecem propriedades de autocura. Além disso, a taxa de convergência de algoritmos PSG está teoricamente comprovada através do uso do teorema de Perron Frobenius para cadeias de Markov (MCs) e pode ser otimizada. Esta tese também relata a criação de um simulador onde testamos MCs geradas com diferentes procedimentos. Após a realização de várias simulações, investigamos como se comporta PSG na práctica e em comparação com os resultados dos cenários ideais. Além disso, comparamos o nosso sistema de backup distribuído (DBS) baseado em P2P com o Hadoop Distributed File System (HDFS) e em condições específicas, conseguimos superá-lo. Conclui-se que PSG é uma alternativa viável às abordagens existentes, mas é necessária mais pesquisa para determinar se é desejável.

Palavras Chave

Armazenamento *Peer-to-Peer*; Cadeias de Markov; Coordenação de Enxames; Durabilidade de Ficheiros;

Contents

List of Figures

List of Tables

List of Algorithms

List of Theorems

Acronyms

CFS	Ceph File System
CLI	Command-line Interface
DBS	Distributed Backup System
DFS	Distributed File System
DHT	Distributed Hash Tables
DoS	Denial of Service
EC	Erasure Coding
FMMC	Fastest Mixing Markov chain
GFS	Google File System
GlusterFS	Gluster File System
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HybridFS	Hybrid File System
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
MC	Markov Chain
MBS	Minimum Bandwidth Regenerating
MDS	Maximum Distance Separable
MH	Metropolis-Hastings
P2P	Peer-to-Peer
PFK	Peerfact.KOM
PSG	Probabilistic Swarm Guidance
QoS	Quality of Service

SDP	Semi-definite program
SLEM	Second Largest Eigenvalue Modulus
SG	Swarm Guidance
SGDBS	Swarm Guided Distributed Backup System
TCP	Transmission Control Protocol
TSP	Trust Service Provider
UDP	User Datagram Protocol
XML	Extensible Markup Language

Symbolic Notation

Miscellaneous

Epochs or discrete timestamps	$T = \{t_0, t_1, \ldots, t_n\}$, $n \in \mathbb{N}$
Function	$f(x)$
Parameter or variable	x
System States	$S = \{s_0, s_1, \ldots, s_n\}$, $n \in \mathbb{N}$

Probability theory

Event probability	$P(X)$
Random variable	X
Random process	X_t, $t \in T$

Algebra

Vector	$\underline{v}$
Vector at epoch	$\underline{v}^{(t)}$, $t \in T$
Matrix	$\mathbf{M}$
Matrix or vector transpose	x^{T}
Matrix element	$\mathbf{M}_{ij}$, $i, j \in \mathbb{N}$
Matrix row	$\mathbf{M}_{i\star}$, $i, j \in \mathbb{N}$
Matrix column	$\mathbf{M}_{\star j}$, $i, j \in \mathbb{N}$
Matrix Eigenvalue	$\lambda(\mathbf{M})$
Identity Matrix	$\mathbf{I}$
Zeros Matrix, Ones Matrix	$\mathbf{0}$, $\mathbf{1}$
Ones Vector	$\underline{\mathbf{1}}$
Equilibrium, stationary or steady-state distribution vector	$\underline{e}$
Markov chain at epoch	$\mathbf{M}^t$, $t \in T$
Markov chain event chance at epoch	$\mathbf{M}^t_{ij}$, $t \in T \wedge i, j \in \mathbb{N}$

1 Introduction

Contents

1.1 Definitions

Average Path Length Sum-average of the length of all shortest paths between any two nodes i, j.

Churn Collective effect created by the arrival and departure of independent nodes in a network.

Clusters Groups of nodes, where most nodes have direct paths to most of the others.

Connected Network Any node i in a network can effectively communicate with all others.

Degree Number of bidirectional paths at some node i in the network.

- **In-Degree** Number of incoming paths, through which a node only receives data.
- **Out-Degree** Number of outgoing paths, through which a node only sends data.
- **Regular Network** All nodes in a network have the same path degree.

Durability Once accepted by a system, data is not lost.

- **Availability** Data on a system can be promptly accessed.

Error Part of a system that may lead to a fault, e.g., difference in actual and expected outputs.

Fault Condition that may make the system fail to perform its required function.

Failure Occurs when at least one external state of a system, deviates from correct service.

Flash Crowd Spike in user activity concerning number of queries being made to a system.

Network Diameter Number of links that compose the longest of all shortest paths between any two nodes i, j in a network.

Path Stretch Ratio between the number of links a message has to traverse, between any two nodes i, j, in a physical network and its abstracting overlay.

Power-Law Non-uniform distribution of influence in a system, e.g., some nodes have more responsibilities or higher access quotas because they have more resources than the rest.

Small World Combination of high clustering coefficients and low average path length, where most nodes are not necessarily direct neighbors of each other.

1.2 Fundamental Concepts

In general, distributed systems should be *dependable* to the degree that fits their use-case, i.e., they should be able to avoid service failures that are more frequent and more severe than is acceptable. To be dependable is to be ready to provide the correct service (***availability***), in a continuous manner (***reliability***), with the absence of catastrophic consequences to clients (***safety***), e.g., losing a file, with no improper system alterations (***integrity***) and, finally, to be able to undergo modifications and repairs at any time (***maintainability***). Often confused with safety, ***security*** is a set of mechanisms that protect the system against malicious attacks, ensures the absence of unauthorized disclosure of information (***confidentiality***), and possibly remove intruders from it. Dependability does not consider security. As the amount workload of the system increases, it's capacity to remain operational with little or no degradation can be called ***scalability***. Features that contribute to the above characteristics include: ***load-balancing***, which is the process of balancing communication, processing, and storage overhead among all nodes of the system. It helps to avoid bottlenecks, increasing operational performance, and catastrophic failures; ***fault-tolerance*** enables a system to continue operating, possibly at a reduced level, rather than failing as a whole, when a portion fails. Increasing the number of computers implementing the same behavior (***machine redundancy***) and storing multiple copies of a file (***data redundancy***) are two examples of fault-tolerance.

1.3 Motivation

We have entered an era in which devices with computing capabilities are ubiquitous and thus, one can argue that now, more than ever, individuals generate more data that they would like to store in multiple places to avoid its loss, e.g., photos. Governments and Companies have been recognizing IT and the internet in general, not only as a complementary part of their operations but as an integral, sometimes vital, part of their business and governance. The latter type of organization is often required by law to store critical documents for long periods of time and may incur substantial penalties if they fail to comply. For this reason, large quantities of paper are stored in safe-rooms, increasing passive costs, even when digital copies exist on local or remote servers. When it comes to digital storage, among others, one possibility is to use Distributed File Systems (DFSs); these can be advantageous because they facilitate the distribution of documents to multiple clients, which can collaboratively and transparently modify them. Continuous availability and session-guarantees are fundamental properties of DFSs designed for interactive editing. Distributed Backup Systems (DBSs) are a sub-category of DFSs, and their priority is to ensure that uploaded files become durable, even if that means partially sacrificing shareability or editability of the persisted files.

With the evolution of distributed systems, two large-scale computing paradigms have gained popularity, which may be used to implement such systems. On the one hand, we have Peer-to-Peer (P2P) networking, in which equally privileged peers contribute with a portion of their resources to achieve common goals. This approach is popular due to, among others, its self-organized behavior, lack of centralization, and low cost, e.g., HandyBackup [7]. On the other hand, Cloud platforms offer unmatched, on-demand, self-served, availability, and reliability at higher price points. The latest approach is trendy, with companies such as DropBox, Google, and Microsoft [8–10] offering diversified solutions to individuals and organizations. Cloud-based systems are often centralized architectures, in which a large number of computers are clustered and managed by master entities, which may become bottlenecks. While P2P implementations are cheaper for both companies and their clients, the bottom line is that these have a hard time achieving performance levels seen in Cloud implementations, even when acknowledging their unideal inclusion of centralized components. P2P approaches have a higher inherent risk of permanent file loss resulting from the fact that contributors may leave at any time they desire for no particular reason, making them somewhat unappealing for clients who seek to store critical data. We believe a perfect system would be made of entirely independent and equally hard-working peers operating together without any need for network monitors or master entities, hence reducing cost. Capable of deciding how best to make a file durable using a consensus algorithm, which is in itself a complex problem, and never failing at doing so, thus delivering highly dependable service. Creating such a system, however, is not trivial, and we would be naive to assume it could be done with the current state-of-the-art techniques. However, as hardware on devices evolves, in particular, with disks having ever-increasing mean-time-to-failure values and with allocation spaces growing faster than the generality of file-sizes; and having identified a gap in the market regarding DBSs based on P2P technologies, we believe it is important to revisit this topic.

This body of work aims to find out how viable it is to use Probabilistic Swarm Guidance (PSG) in a DBS. The reasoning behind this exploration is due to the widespread adoption of this method in the robotics and control fields. Results demonstrate that it is possible to gift autonomous agents, working independently of each other, with simple probabilistic rules and possibly some limited knowledge of the environment, and have them achieve complex tasks as a group, with minimal setup effort. Furthermore, these swarms of agents can typically deal with unexpected predicaments and recover from them without human intervention. By applying proper adaptations, we investigate if these properties hold in a P2P scenario. Thus, this thesis first develops a simulator and then implements PSG to a DBS environment, something which, to the best of our knowledge, has not been attempted. We hope to give new ideas to the scientific community that may put us closer to the utopic system.

1.4 Brief Literature Review

1.4.1 Swarm Guidance

The control and robotics fields of research are packed with complex problems. One such problem is how to control groups of multiple robotic agents efficiently. Swarm Guidance (SG) is a technique inspired by the behavior regarding groups of animals as seen in nature, e.g., bird flocking [11], shepherding [12], ant-colonies [13], and glowworms [14], which emerges as a solution to these complex problems, by employing dynamic and straightforward techniques at the agent level rather than at the group level. In other words, SG is a form of decentralized control that has each agent applying a local rule that results in a collective behavior achieving the mission goal for the group. In turn, this avoids intricate algorithms, resulting in easier to build and program robotic systems with more robustness, flexibility, and scalability.

Research by B. Açıkmeşe *et al.* shows that PSG [15–17] is also a viable form of SG, and its the approach we choose to adapt to our DBS. Summarizing, SG is a compelling technique that may be applied to a wide array of topics ranging from crowd control [18] and oil spill cleanups [19] to the maneuvering of crewless vehicles in combat [20] and search and rescue operations [21]. We study its effectiveness in the safekeeping of files stored in P2P networks.

1.4.2 Distributed File Systems

As mentioned in the previous section, one way to classify a DFS concerns its level of architectural centralization. P2P approaches are often decentralized, except for some systems, such as BitTorrent [22]. To say that a system is centralized means that at least one node in the architecture plays a central role in its operation, e.g., a metadata server that stores file locations or a monitor server that decides what nodes are operational on the network. Classical DFSs like Google File System (GFS) [4] and Hadoop Distributed File System (HDFS) [23] are examples of centralized architectures. In fact, their modus operandi is the same except for the approach to security and permission handling. In these systems, clients contact master (metadata) servers to know which storage servers they need to contact to read or write files. These same masters are also responsible for receiving heartbeats from storage servers and controlling replication levels within them. Despite the disregard for centralized architectures in the scientific community, these systems have proven to offer unmatched service guarantees, concerning reliability and durability, and operational performance. Ceph File System (CFS) [5,24,25], another state-of-art DFS, uses the CRUSH [26] algorithm to provide fast and precise localization without using an indexing server and leverages P2P behavior within clusters of storage servers, to mask and recover from faults. CFS relies on consensus performed by some of the cluster's dedicated nodes to decide on which storage servers are up. On the other end of the spectrum, we have a novelty system, called Gluster File System (GlusterFS) [27], a completely decentralized solution, in which indexing is done through algorithms similar to those in CFS. GlusterFS does not have any metadata dependency, which makes it exceptionally fast at handling small file operations, albeit slower than GFS and HDFS at handling large file operations. Finally, recognizing that there is no one-system-fits-all solution, Hybrid File System (HybridFS) [28], creates an abstraction layer over HDFS, CFS and, GlusterFS allowing users to use them as if they were one single DFS, ensuring the best storage and access performance by respectively using dynamic file migration mechanisms and artificial intelligence to select which subsystem will receive a certain file.

1.4.3 Overlay Networks

In this master's thesis, we implement a SG algorithm that targets a P2P-based DBS; hence, it is essential to understand P2P overlays. They exist because, in large-scale systems, it is unfeasible and undesirable for each peer to know and interact with every other peer or entity in the network. An overlay is, thus, a logical abstraction of the physical network, typically Internet Protocol (IP) based, and where, single-hop edges represent links between pairs of peers as depicted in Figure 1.1. There are two main categories of overlays, Structured and Unstructured. In the foremost [29], peer placement follows rigid rules to speed up read operations, commonly using Distributed Hash Tabless (DHTs) or tree implementations, e.g., decision trees. Thus, these overlays have higher maintenance costs, since one change in the topology can cause a cascade of modifications concerning peer organization or the data they hold or both, causing them to not be suitable for highly dynamic environments. In the latter [22, 30], information dissemination is likely to occur through gossip or broadcasting communications; this makes reads and writes slower than their structured overlay counterparts, but the maintenance is often straightforward. Since peer placement is arbitrary, a new peer entering or leaving the network requires insignificant rearrangements. Consequently, they tend to provide better scalability and a higher degree of resilience in the advent of failures, however, peer isolation is more likely to occur. In the last decade, some multi-layer overlays emerged. They are but a combination of two or more conventional overlays concurrently abstracting the same physical network. The overlays may work alongside or on top of each other, i.e., in a horizontal or vertical arrangement. When properly combined, these multi-layer types can satisfy a broader range of requirements for their applications and off-set some of the weaknesses of the conventional ones. However, network and computational resources may also deplete faster if attention is not given during design. Furthermore, their complexity can lead to more service failures. Alternatively, bio-inspired over-

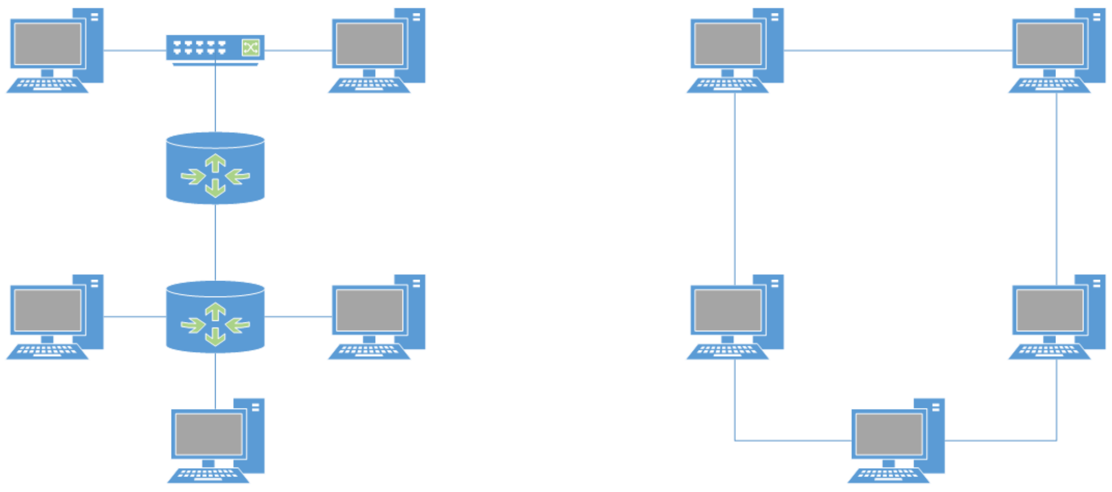


Figure 1.1: P2P Network Diagrams: IP layer *vs.* One possible overlay abstraction

lays [31,32] have shown promising results, and their objective is also to diminish or eliminate some of the inherent disadvantages associated with either structured or unstructured overlays. As the name might imply, the trick is to introduce a collection of simple agents that accomplish complex behaviors (a similar idea to PSG) that would otherwise require time or space consuming algorithms. These agents usually roam the network to aggregate data and rearrange file locations or peer connections. The downside is that the results of agent actions are not always easy to evaluate or predict. We review the overlays we find most interesting in Chapter 2, but for a comprehensive study on the topic we suggest the reader refers to the work of A. Malatras [33].

When designing P2P overlays, one must consider fairness work, fairness of access, robustness, and resistance to churn. We can use the ***degree distribution*** to evaluate the robustness of a configuration since it is possible to detect combinations of weakly connected nodes and individual nodes with many connections (***hubs***), respectively. Hub failures have a significant impact on a system performance or Quality of Service (QoS), severely damaging or even halting it, since weakly connected nodes around them can become isolated. Degree studies provide insights regarding fairness parameters. With the ***network diameter*** and the ***average path length***, we can predict how many links messages will traverse to reach their destinations and how much bandwidth they consume on average. High percentages of neighbors a node i has, who are also neighbors among themselves (***clustering coefficients***) indicate a tendency for network partitioning when there is much churn or when many nodes disconnect sequentially. The latter can also manifest bandwidth overhead when using gossip protocols due to redundant communications within clusters. Unlike high degree nodes, clusters, provide systems with better performance during flash crowd events.

1.4.4 Membership Management

The topology of an overlay is critical to its performance. In dynamic systems, the original topology will eventually be obsolete, e.g., due to changes in the underlying network or changes in traffic patterns, and thus, will require reconfiguration. While state-of-art overlay protocols take this issue into account, we highlight two membership management frameworks that target unstructured gossip-based protocols. Newscast [34] and Cyclon [35] gift peers with autonomous capabilities regarding self-organization and overlay healing, and provide peers with useful up-to-date partial views even in large-scale systems, consuming minimal bandwidth. In both protocols, each peer periodically shuffles their partial view of the overlay with one of their current acquaintances. Through timestamps, the time that a peer's IP address remains unselected for shuffling is limited; this results in more up-to-date partial views and, thus, topologies, as well as in the uniform distribution of each peers' addresses over the network. Most importantly, both frameworks enable fast fault detection and facilitate the prevention of failures. Timely detecting disconnected or faulty peers in DBSs is of great importance to avoid losing documents.

1.4.5 Trust in Peer-to-Peer

In P2P communities, trust is essential for clients who are downloading files; this is because the community participants are unknown entities, which are not always well-intentioned. Torrent-based communities are examples where file-sharing often results in the spread of viruses, trojans, and other malware; hence the need for mechanisms that prevent these and other attacks such has Denial of Service (DoS). In a DBS, trust can also be important to decide on the reliability of peers, e.g., the amount of time they are online, the quantity and quality of resources they can provide, and as a result, how data is spread on the overlay. Measuring trust has been a topic of ongoing research ever since distributed systems came to exist. Early systems such as the one proposed by Y. Arafa [36] used Trust Service Providers (TSPs) as a mean to classify participants of the system. The problem with this approach is the introduction of another centralized service and the increased cost. Similar methods are viable and cost-efficient for platforms such as Amazon and eBay, where trust can be seamlessly integrated with other centralized functionalities but is inadequate for P2P communities, particularly those that are fully decentralized or do not require any form of user authentication. Noticing this issue, Li Xiong *et al.* proposed PeerTrust [37], a feasible trust model for P2P e-commerce applications, introducing three basic trust parameters and two adaptive ones, which mainly concern feedback received from the community, the quantity and the context of transactions. Finally, recent work by Z. Yuhong *et al.* [38] and by S. K. Awasthi *et al.* [39] identified two deficiencies in both old and contemporary P2P trust frameworks. The first being that they consider only local trust as a measure of confidence and the second is the normalization of peer trust values. Combined, these result in a few problems: time inefficiency of algorithms, popularity being mistaken for trust, past behavior not being entirely represented on prevailing trust values, and the decrease in trust of every other peer when the trust of some other increases.

1.4.6 Fault Tolerance and Data Durability

Peer availability in P2P networks is ever-changing and not guaranteed, and thus, the durability of files is at risk. To accomplish file durability, a system must deal with faults. Ideally, the faults will be prevented, forecasted, and recovered from. Apart from practicing good programming and avoiding error-prone code, one example of fault prevention is suggested in the work by K. Gupta *et al.* [40]. In this work, only nodes who prove to have minimum machine specification requirements, are allowed to provide storage space to the DFS. We do not believe that this metric alone is a sufficiently good filter, but if combined with other tools, it can be a positive contribution. Another way of accomplishing the desired durability is by employing file-level or block-level replication or Erasure Coding (EC). Replication has the advantage of having a straightforward, less resource-intensive recovery model, faster recovery and faster read operations. On the other hand, EC has a better expansion factor (higher durability for the same amount of utilized resources, e.g., disk space), and because of possible write parallelization, these operations are usually faster [41]. However, one downside of EC is that recovering files is CPU intensive and a time-consuming process, increasing the window of time where additional faults may occur. Moreover, research suggests that peaks in bandwidth during recovery might further degrade the quality of the systems, which can lead to more faults due to congested network links [42]. Another key question, often referred to in the literature as the replica-control problem, is: *at what time should a system replicate or recover data?* Reactive approaches are the least complex and consist of recovering data whenever a failure is detected; they suffer the most from bandwidth bursts, and burst mitigation includes the use of predictive models and thresholds to avoid delay recovery [43, 44]. Proactive approaches seek to smooth bandwidth consumption by injecting new redundancy data into the network over time at a fixed or variable rate [45]. Some proactive approaches can be considered fault forecasting mechanisms. One such example is a system that monitors peer patterns and tweaks the injection of new redundancy blocks according to the system current state in an attempt to decrease the loss probability [46]. Another example would be a system that uploads data to a highly reliable cloud server [47, 48] when it detects that overlays are in critical condition.

1.4.7 File Replication with Erasure Coding

EC algorithms are defined by the number of data symbols k to be encoded, and the number of coded symbols n to be produced, i.e., by the pair $(n,\ k)$ ECs. The fundamental idea, is that a file of size S is divided into k equally sized fragments then turned into $n > k$ encoded fragments, allowing the original file to be recovered from any set of encoded-fragments with count k. Replication can be seen as a $(n,$ $1)$ code, without an encoding step. Systems that use (n, k) or more sophisticated ECs are evaluated under some of the following metrics: repair-bandwidth, disk I/O cost and repair locality. Since coding is a slow process, CPU cycles, XOR counts and cache misses are often evaluated as well. The first metric is of particular importance, in our opinion, because whenever a peer fails in our implementation, all files it held would need to be promptly recovered by some peer who would need to gather enough fragments from possibly many different nodes. Such burst may lead to delays or even failure of the repair process due to physical link saturation between peers who are involved in the repair process, or even throughput constraints enforced at individual nodes.

Concerning algorithm classification, the most widely known belongs to the Maximum Distance Separable (MDS) class, which likely offers the best reliability-redundancy trade-off. However, research [49] suggests that for coding to be competitive in DFS, we need to minimize costs inherent to the MDS class, while keeping redundancy levels to only the amount necessary to ensure specific service guarantees. The same research identifies a Minimum Bandwidth Regenerating (MBS) class and refers to codes in between these two as *regenerating codes*. There is no standard for classification EC algorithms, and often, it depends on the obtained trade-offs as a consequence of chosen n, k values, possibly some extra parameters, and actual algorithmic implementation. Shokrollahi *et al.* introduces raptor codes [50], which are a linear-time coding and decoding extension of fountain codes, belonging to the rateless code class. Rateless codes are characterized by the property that a potentially limitless sequence of encoded fragments can be generated from a given source, and the original data is recoverable from any subset of the encoded fragments of size equal to or slightly larger than original data size. Rateless codes differ from MDS codes because they are scalable with respect to both machine and network resources, and fault-tolerant concerning dynamic and heterogeneous network environments. Zhou *et al.* [51] studies state-of-the-art techniques that improve coding efficiency through bitmatrix optimization, vectorization, and scheduling, proving that, when used in conjunction, fragment generation and data reconstruction can be up to 552.27% faster.

1.5 Contributions

This thesis main contributions are the design, implementation, and evaluation of a novel algorithm in P2P-based DBSs. The algorithm aims to ensure the durability of uploaded files, i.e. file survivability, using PSG. The main features can be summarized as follows:

- Creation of an open-source simulator dedicated to the problem at hand, named Hives.
  - Well-documented; Easy to use, modify and extend; Supporting up to 10^4 nodes.
- Novel proposal of using a PSG algorithm in DBS environment with the following steps:
  1. Given a P2P overlay configuration, $\mathbf{A}$, and the desired equilibrium distribution, $\underline{e}$.
  2. Synthesize a Markov chain, $\mathbf{M}$, that converges to $\underline{e}$, using heuristic methods.
  3. Optimize $\mathbf{M}$ mixing rate, using mathematical optimization techniques.
- Evaluation of the system and comparison with HDFS.
- Enumeration of possible topics for future research work.

1.6 Document Structure

The remainder of this document has the following structure. In Chapter 2, we explain why P2P and network simulators are important, and we review two of the most well-known. We identify key problems with available simulators. We then explain how we implemented a custom made simulator specifically built to tackle our research and discuss our main results regarding this part of our work. Chapter 3 focuses on our primary contribution. We review the architectures of important contemporary DFSs and some P2P overlays. Still in this chapter, we review work-related with Markov Chains (MCs) and PSG, discuss strategies to minimize a MC mixing rate and the expected performance of such optimizations, resulting from our experimentation. Lastly, we define our problem, propose a solution, and give insights into its implementation before analyzing the simulation results obtained from the code package presented in Chapter 2. Chapter 4 offers some conclusions and discusses limitations that serve as lines of future work.

2

Hives: An Open-source Simulator

Contents

2.1 Context

Due to the dynamic nature of P2P-based applications and their decentralized nature, it is often difficult to study the properties of the system in real environments or deployed states. We could conclude if the application effectively achieves its main goals, e.g., manages to ensure the durability of files over a given period, but comprehending all of its intricate details would be a more difficult task. However, the main problem with live testing applications of this type is the cost associated with development and deployment, especially if the number of machines in the system is large. It would also be harder to hypothesize and validate different system configurations. Researchers often resort to the use of simulators to solve these issues and develop P2P applications, algorithms, or overlays, at little to no cost, in terms of both time and money. There are two categories of simulators, packet-based simulators, and application-level simulators. The former includes simulators such as NS-3 [52] and are good for calculating delay, bandwidth expenditure, and routing for each packet sent by the nodes in the environment. Unfortunately, these simulators are complex and very low-level. They are designed mainly for the simulation of physical IP networks and routing protocols. The latter type abstracts the network stack and is better suited to study P2P schemes while providing reasonably accurate means of evaluation. Among other parameters mentioned in Section 1.4.3, these simulators can approximate bandwidth expenditure, help determine clustering coefficients and measuring degree distributions under various conditions of churn reported in public data-sets that were composed using data from actual implementations.

2.2 Related Work

2.2.1 PeerSim

PeerSim [2, 53] is likely to be the most well-known P2P application-level simulator at the date of this thesis. The paper describing the framework counts 288 paper citations in the IEEE[1] database alone. The framework offers two types of operation engines, cycle-based and event-driven. The cycle-based engine supports simulations with up to 10^7 peers by introducing relaxed assumptions in the system, whereas the event-based engine scales to 10^5 peers and tries to make simulations more realistic. At the implementation level, the fundamental difference between cycle-based simulators and event-driven simulators is that the simulation messages are necessarily synchronous and sequential in the first type. However, in event-driven simulators, there are no discrete-time steps, i.e., simulations have asynchronous characteristics. As a result, the latter type of engine makes timing or omission failures straightforward to implement by, for example, introducing random waiting times in message responses. PeerSim comes with a handful of predefined overlay protocols for both structured and unstructured overlays, support-

[1] Institute of Electrical and Electronics Engineers

ing various churn patterns. A few more protocols on top of the default ones can be found as extras developed by independent researchers who used PeerSim in their work [53]. The framework design has modularity, ease-of-use, and scalability in mind. There is a perception that PeerSim modules are easy to understand, reuse, and extend. Documentation is provided, at least for the cycle-based version, but the event-driven engine lacks information sources. Scalability is achieved by disregarding much of the transport layer properties and ignoring everything below and including the network layer in the OSI model (Figure 2.1). The provided source code comments (JavaDocs) are not thorough and there seems to be a lack of good programming practices. Many hacks exist in the code to improve the engine performance, which does not help readability either.

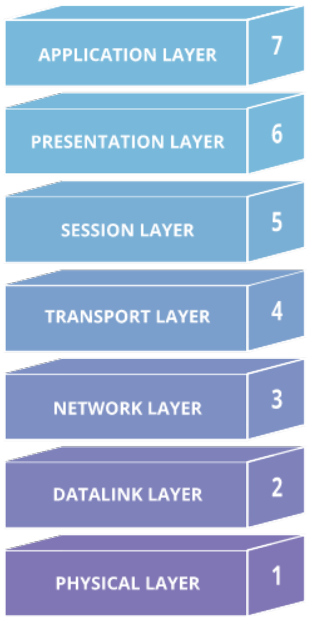
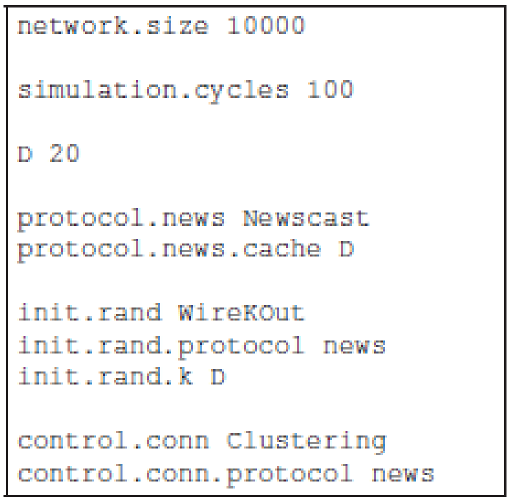


Figure 2.1: OSI Model Stack [1]

PeerSim architecture defines a network as a list of nodes, with each node having a list of protocols it can run at any time. The simulation uses initializers and controls, respectively, executing before and during the simulation. Both monitor and modify the simulation, e.g., bring up or shut down some nodes of the network or modify their protocol parameters. Controls further aggregate data about the simulation state using observer modules annotated in the configuration files. An example configuration file is given in Figure 2.2. In it, we can see that once executed, the simulation will contain 10^3 network nodes and will last 10^2 cycles. The variable D is assigned the value of 20 for later use. Each node runs one single protocol called news implemented by the class Newscast, which receives one single parameter with value referenced by variable D. The subsequent three lines use the class WireKOut to initialize randomized overlay links to be managed by the news protocol, thus creating a random overlay topology with constant out-degree k equal to D. Finally, an observer of type Clustering periodically reports and logs the clustering coefficient of the overlay network managed by the news protocol for later processing by the researcher.



```
network.size 10000

simulation.cycles 100

D 20

protocol.news Newscast
protocol.news.cache D

init.rand WireKOut
init.rand.protocol news
init.rand.k D

control.conn Clustering
control.conn.protocol news
```

Figure 2.2: PeerSim configuration file [2]

2.2.2 Peerfact.KOM

Peerfact.KOM (PFK) [54] is another P2P simulator. It is both more complex and powerful than PeerSim but has less recognition. PFK offers support for structured and unstructured overlays. However, rather than the usual approach taken by generic simulators, which focus on simulating the layers above the transport level, PFK offers some functionality that gets it closer to packet-level simulators, such as NS-3. In fact, this simulator targets large-scale distributed systems that require an evaluation concerning the dependencies in multi-layer P2P environments. PFK only offers an event-driven engine that assures sequential processing of events by use of an event queue. Events reach the queue via entity insertion or external action files, i.e., simulated nodes or predefined events with inputted configurations.

The architectural design of the simulation framework is depicted in Figure 2.3. The user layer defines user strategies on top of the application layer, such as the API calls they could perform. The application layer defines the application logic, e.g., a DBS. The service layer allows researchers to define intermediate protocols, for example membership management, aggregation, and trust protocols, like Cyclon [35], Newscast [34], and AbsoluteTrust [39]. Finally, the P2P Overlay layer defines the organizational structure of the simulated nodes, with some pre-implemented overlay protocols, but researchers can write their own and plug them in, as long as they adhere to the framework specification. The remaining layers are self-explanatory: the transport layer allows the choice between Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) messaging between nodes, and the network layer simulates different

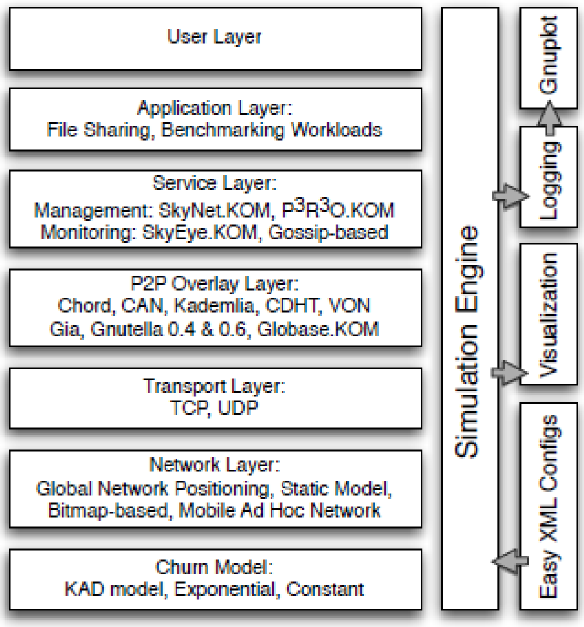

User Layer
Application Layer: File Sharing, Benchmarking Workloads
Service Layer: Management: SkyNet.KOM, P3R3O.KOM Monitoring: SkyEye.KOM, Gossip-based
P2P Overlay Layer: Chord, CAN, Kademlia, CDHT, VON Gia, Gnutella 0.4 & 0.6, Globase.KOM
Transport Layer: TCP, UDP
Network Layer: Global Network Positioning, Static Model, Bitmap-based, Mobile Ad Hoc Network
Churn Model: KAD model, Exponential, Constant
Simulation Engine
Gnuplot
Logging
Visualization
Easy XML Configs

Figure 2.3: Peerfact.KOM layered architecture [3]

physical network properties. When combined, these accurately map throughput, delay, jitter, and other values. Finally, the churn layer allows the choice between the use of well-known data sets (similar to PeerSim) or the use of mathematical functions or distributions. PFK recommends no more than 10^5 simulated peers for timely simulation but claims to support up to 10^6 without issue. This simulator surpasses PeerSim by providing a visualization Graphical User Interface (GUI), a logged history of relevant events (specified by the user), and files containing statistics that can be directly fed into gnuplot software without further processing.

While PFK is a powerful simulator, that is simultaneously extensively well-documented, more so than any other simulator we have reviewed, its configuration requires that a user understands the Extensible Markup Language (XML) schema. Furthermore, the same level of configuration is not as straightforward to accomplish as the PeerSim, even when considering that many options can be left vacant, resulting in the use of default values. This complexity is illustrated by the 110-page document [55] on how to use PFK when compared to a 19-page document [56] on PeerSim.

2.3 Contributions

Although the aforementioned simulators are great tools for researchers released under the GPLv3 license, most simulators tend to lack documentation or practical usage examples. They are also usually written with Java or C# programming languages and expect the tested protocols or algorithms to use those languages as well. Some protocols and algorithms, such as PSG, are very mathematics-oriented, based on random weighted choices (probabilities), matrix and vector operations (linear algebra), data-set manipulations, convex optimization problems, to name a few. Neither Java nor C# are designed or particularly good for this type of programming. There are barely any open-source packages or community support for mathematical or data science purposes. In contrast, Python is full of rich libraries such as NumPy [57], SciPy [58], and Pandas [59], which provides all the tools to test the proposed DBS. While some P2P simulators exist for the Python language, most have few options and lack documentation and had not had any updates in a few years. Unlike the previously mentioned simulators, none of them has seen adoption in scientific research, making them even more unappealing. For these reasons, this thesis details the implementation of a custom and dedicated simulator that is made freely available at GitHub Repository[2].

2.4 Problem Statement

In the thesis, we propose to study the properties PSG in a DBS environment with application to file durability. Given the objectives of the research, from a *technical* perspective, the envisaged simulator should support:

- Multiple P2P groups in simultaneous, each ensuring the durability of a different file.
- P2P groups may have different membership sizes up to at least a few thousand nodes.
- Network nodes may belong to an arbitrary amount of P2P groups.
- Network nodes may have different uptimes and their faults can be permanent or transient.
- Provide mechanisms to detect faults, e.g., disconnected network nodes or corrupted files.
- P2P groups may evict faulty nodes and replace them with new nodes.
- Provide mechanisms to log P2P group states and properties as the simulation progresses.
- Provide mechanisms that simulate time to recover faults, e.g., file replica re-replication.

From a *usability* perspective, the simulator should:

- Be straightforward to configure.
- Be easy to modify and extend, concerning modules and classes.
- Provide mechanisms to create different simulations rapidly.
- Create human-readable output files that can be post-processed with any programming language.

[2] https://github.com/FranciscoKloganB/hivessimulator

2.5 Implementation

This section describes how we went about fulfilling the requirements specified in Section 2.4. At first, we present our assumptions and design choices (2.5.1). We then explain how to get started with the simulator, which demonstrates our approach to usability (2.5.2). Next, we present the existing modules, their purposes and relationships, including diagrams (2.5.3). Lastly, in we make a brief overview of the simulator's execution flow (2.5.4).

2.5.1 Assumptions and Design Choices

The proposed **simulator uses a cycle-based** approach, although it would be worth the possibility to have asynchronous implementations[3]. The cycle-based offers the advantage of being easy to monitor and log the behavior of network nodes and their states as individuals. Consequently, it facilitates network group resource expenditure monitoring, capturing the overlay health, and determining if files are being persisted in the DBS using the algorithms presented in Chapter 3.

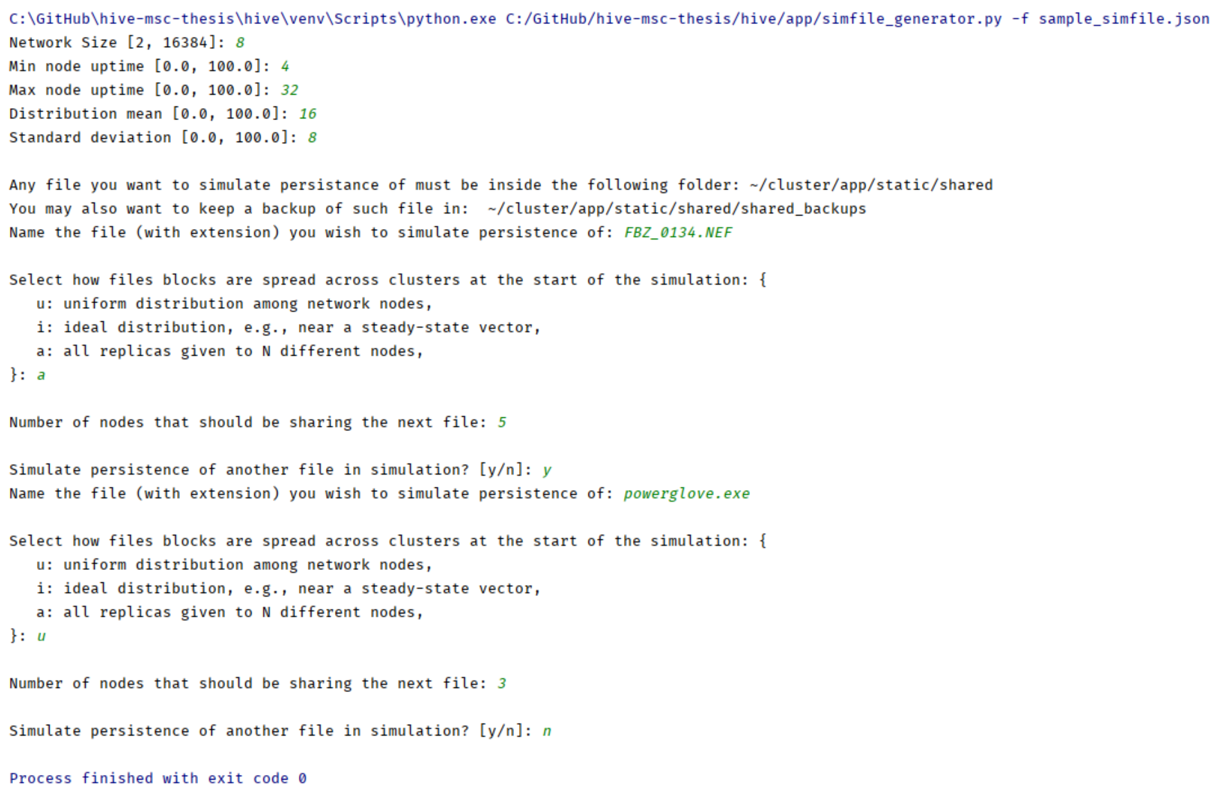
Simplicity led to a second choice: **support fail-stop, fail-noisy and, fail-silent**, rather than fail-arbitrary (byzantine) models. By ignoring byzantine faults, we are free to focus on the algorithm behavior and results. The first three models allow for comprehensive testing with arbitrarily difficult simulations while avoiding topics that are outside of the scope of this thesis, such as security and any attack done to or by nodes, and avoids the use of complex and unnecessary failure detection algorithms when implementing prototypes in the simulator.

There are a variety of aspects that influence a simulator's usability, some being source code readability and good documentation as well as performance and features, e.g., channel loss, disk corruption, message corruption, time omissions, churn simulation based on real-word traces, and other types of simulated faults. With ease-of-use in mind, **divide network, hardware and engine properties, and simulation properties** into two **different configurations**. This approach is uncommon because it makes result reproducibility harder, but allows researchers to set up the simulation engine quicker and without information overload. Furthermore, our engine cannot fully replicate experiments even if one single configuration file is used, and as will be explained in the next section, having this separation opens possibilities regarding network properties during simulations.

2.5.2 Configuration and Initialization

The first step to use the Hives simulator is to generate a text file in JSON format, called a simfile, which defines simulation specific properties. Simfiles include settings such as the number of nodes available

[3] Event-based simulators are not necessarily assynchronous, e.g., PeerSim events are placed in a synchronous queue avoiding $\mathcal{O}(n)$ queries to simulated nodes, because those events are only consumed by interested entities.

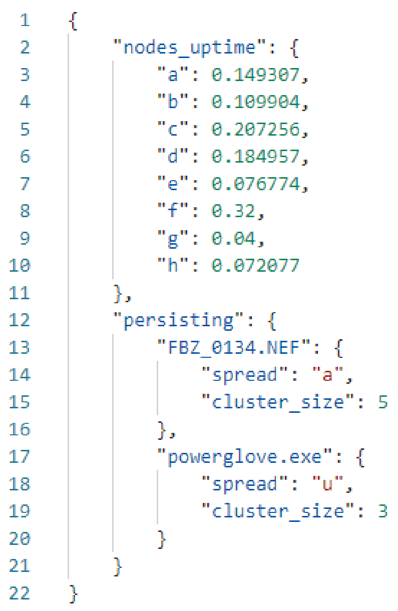


```
C:\GitHub\hive-msc-thesis\hive\venv\Scripts\python.exe C:/GitHub/hive-msc-thesis/hive/app/simfile_generator.py -f sample_simfile.json
Network Size [2, 16384]: 8
Min node uptime [0.0, 100.0]: 4
Max node uptime [0.0, 100.0]: 32
Distribution mean [0.0, 100.0]: 16
Standard deviation [0.0, 100.0]: 8

Any file you want to simulate persistance of must be inside the following folder: ~/cluster/app/static/shared
You may also want to keep a backup of such file in:  ~/cluster/app/static/shared/shared_backups
Name the file (with extension) you wish to simulate persistence of: FBZ_0134.NEF

Select how files blocks are spread across clusters at the start of the simulation: {
   u: uniform distribution among network nodes,
   i: ideal distribution, e.g., near a steady-state vector,
   a: all replicas given to N different nodes,
}: a

Number of nodes that should be sharing the next file: 5

Simulate persistence of another file in simulation? [y/n]: y
Name the file (with extension) you wish to simulate persistence of: powerglove.exe

Select how files blocks are spread across clusters at the start of the simulation: {
   u: uniform distribution among network nodes,
   i: ideal distribution, e.g., near a steady-state vector,
   a: all replicas given to N different nodes,
}: u

Number of nodes that should be sharing the next file: 3

Simulate persistence of another file in simulation? [y/n]: n

Process finished with exit code 0
```

Figure 2.4: Command-line Interface after executing *simfile_generator.py* script

in the network, the name of the files that will be stored in the DBS, initial cluster group sizes, and how the file replicas are initially spread inside the clusters. To simplify the process, Hives comes bundled with a python script, *simfile_generator.py*, that expects one single argument for execution, which is the name to be assigned to the simulation file being generated. As the script executes, a series of questions are made in the user's Command-line Interface (CLI) as depicted in Figure 2.4, which will be stored in the format expected by the Hives simulator (Figure 2.5). Lines two through eleven define the storage machines available in the DBS, referred herein as network nodes. The dictionary keys are the node identifiers, which in this case are sequential labels (but could be IP Addresses), and the values are the time those nodes spend online after first joining a cluster group. Lines twelve through twenty-one indicate the simulation will try to preserve two files simultaneously, named *FBZ_0123.NEF* and *powerglove.exe*, respectively. The former file will begin the simulation with a cluster of five network nodes selected at random from the dictionary in line two, and only three of those will receive an entire copy of the file. The latter file will only have three nodes in the cluster group, and file replicas will be uniformly distributed among them. File replica spread strategies u and i, are equivalent for the second file (*powerglove.exe*), but if the cluster definition had more than three nodes, they would no longer be.

Using the generated simulation file within the Hives engine is simple. The user only needs to insert the command *python hive_simulation.py -f samplesimfile.json* in his CLI. Doing so results in the execution



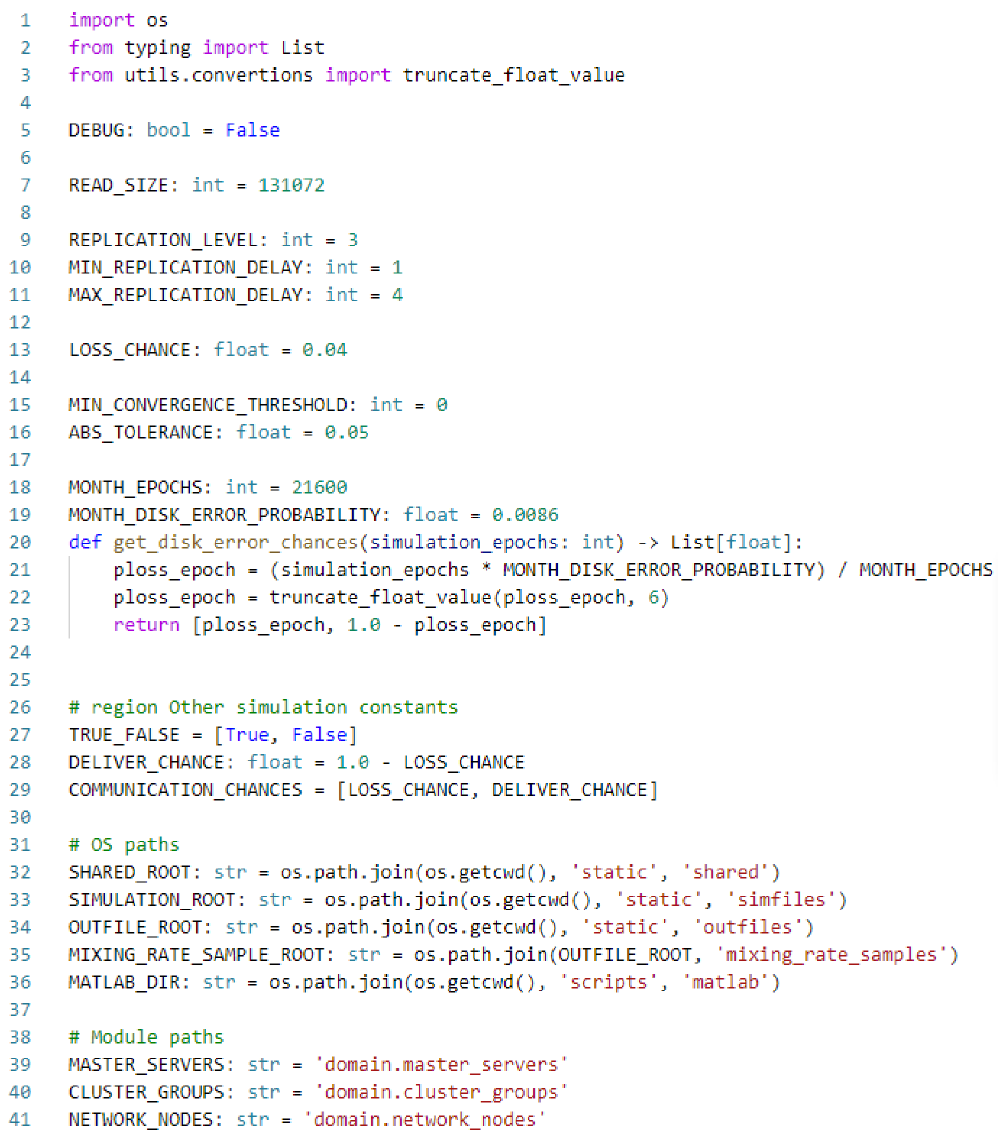
```
{
    "nodes_uptime": {
        "a": 0.149307,
        "b": 0.109904,
        "c": 0.207256,
        "d": 0.184957,
        "e": 0.076774,
        "f": 0.32,
        "g": 0.04,
        "h": 0.072077
    },
    "persisting": {
        "FBZ_0134.NEF": {
            "spread": "a",
            "cluster_size": 5
        },
        "powerglove.exe": {
            "spread": "u",
            "cluster_size": 3
        }
    }
}
```

Figure 2.5: Output of *simfile_generator.py* with choices in Figure 2.4

of a simulation with the default arguments, environment settings, modules, and classes. For more Hives customization information, we recommend reading the simulator online documentation[4].

While further customization is not mandatory, we recommend the user to open the Python module $environment\_settings$ and adapt available constants to suit his purposes. Looking at Figure 2.6, we notice that the module contains important constants for the simulator operation, such as operating system and module paths. Most importantly, it contains data that defines the probability of failing a message transmission, the replication level that the clusters groups strive to achieve during experiments, and the min-max time it takes for file replicas to be recovered. At first glance, there seems to be no advantage to having a configuration module such as this one, other than perhaps simulation files becoming smaller and easier to configure. However, having some configurations as part of a global python file opens the possibility for more flexible and dynamic environments, without obscure functions spread across multiple classes. One example of flexibility is present in line number twenty, where disk error chance returns as a function of the simulation duration based on real-world analysis [60] concerning disk error probabilities. Another example, not depicted in the Figure 2.6, is exemplified in Figure 2.7, where the goal replication level can be globally changed during run time. The provided example is a simple illustration, but, with a similar methodology, a user can extend Hives such that the network nodes update their connectivity status using a different ruleset at some point in the simulation or implement specific churn patterns.

[4] https://www.hivessimulator.tech/

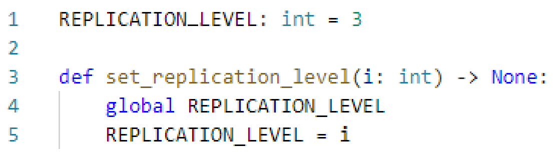


```
1   import os
2   from typing import List
3   from utils.convertions import truncate_float_value
4
5   DEBUG: bool = False
6
7   READ_SIZE: int = 131072
8
9   REPLICATION_LEVEL: int = 3
10  MIN_REPLICATION_DELAY: int = 1
11  MAX_REPLICATION_DELAY: int = 4
12
13  LOSS_CHANCE: float = 0.04
14
15  MIN_CONVERGENCE_THRESHOLD: int = 0
16  ABS_TOLERANCE: float = 0.05
17
18  MONTH_EPOCHS: int = 21600
19  MONTH_DISK_ERROR_PROBABILITY: float = 0.0086
20  def get_disk_error_chances(simulation_epochs: int) -> List[float]:
21      ploss_epoch = (simulation_epochs * MONTH_DISK_ERROR_PROBABILITY) / MONTH_EPOCHS
22      ploss_epoch = truncate_float_value(ploss_epoch, 6)
23      return [ploss_epoch, 1.0 - ploss_epoch]
24
25
26  # region Other simulation constants
27  TRUE_FALSE = [True, False]
28  DELIVER_CHANCE: float = 1.0 - LOSS_CHANCE
29  COMMUNICATION_CHANCES = [LOSS_CHANCE, DELIVER_CHANCE]
30
31  # OS paths
32  SHARED_ROOT: str = os.path.join(os.getcwd(), 'static', 'shared')
33  SIMULATION_ROOT: str = os.path.join(os.getcwd(), 'static', 'simfiles')
34  OUTFILE_ROOT: str = os.path.join(os.getcwd(), 'static', 'outfiles')
35  MIXING_RATE_SAMPLE_ROOT: str = os.path.join(OUTFILE_ROOT, 'mixing_rate_samples')
36  MATLAB_DIR: str = os.path.join(os.getcwd(), 'scripts', 'matlab')
37
38  # Module paths
39  MASTER_SERVERS: str = 'domain.master_servers'
40  CLUSTER_GROUPS: str = 'domain.cluster_groups'
41  NETWORK_NODES: str = 'domain.network_nodes'
```

Figure 2.6: Actual screenshot of *environment_settings.py* configuration file without docstrings.

2.5.3 Main Simulation Modules

Apart from the scripts and modules mentioned in the previous section, Hives has three modules to effectively simulate a DFS. The first module, and perhaps the simplest, is $master\_servers$, which contains classes whose responsibility is mostly coordinating groups of clusters. Any class in this module inherits from the base class Master, which reads and interprets the simulation file. It instantiates all the network nodes (Figure 2.5 - line 2) and the adequate size clusters that will hold the file block replicas with a maximum predefined byte size specified in the environment settings module (Figure 2.6 - line 7). Other than setting up the DFS, Master type classes play the role of centralized servers for the entire network. For example, they could be metadata servers that map which clusters hold blocks of a certain file or



```python
REPLICATION_LEVEL: int = 3

def set_replication_level(i: int) -> None:
    global REPLICATION_LEVEL
    REPLICATION_LEVEL = i
```

Figure 2.7: Example of how to make simulation environments more dynamic using Python.

authentication servers that allow users to log into the system. The *cluster_groups* module is the second of those three modules. Like the previous one, this module contains classes of type Cluster. Clusters are abstractions that make sense mostly in a simulation setting, meaning they do not necessarily have direct real-world counterparts. Clusters are groups of network nodes. In the simulation, a cluster main responsibility is to keep a collection of network node instances in a single object. This design choice slightly reduces the simulator memory footprint. More importantly, it gives us the ability to easily log and capture the group state by making each message in the system pass through entities of this type and making the network nodes execute their epochs only when ordered by the cluster abstraction. The third and last module is *network_nodes*, which contains type Node classes. Nodes implement the algorithms concerning file replica storage and routing, network sampling, sibling complaints, among others, as long as each node can independently execute them. There are cases in which it is more suitable to implement an algorithm in a cluster module class, e.g., consensus, leader election, and other distributed algorithms. Hence, users should implement group algorithms in Clusters types. There are a few more modules and packages bundled within the Hives simulator, which are documented in the simulator official website in the helpers[5] section. In this dissertation document, we also provide one sequence diagram and several class diagrams in Appendix A.

2.5.4 An Overview of the Simulator's Lifeline

We now summarize the code-flow of our simulator, also shown in Figure A.1. A user first runs the command *$ python hive_simulation.py -f simfile.json*, with possibly other flags[6]. What follows is a sequence of object instantiation events (messages 2 through 4) and the distribution of possibly many file blocks of the file whose durability will be tested (messages 5 and 6) to its respective cluster's nodes. Once all nodes and clusters are set up, a loop that lasts the user-defined epochs triggers the same code flow repeatedly, which should always be the same regardless of the user's modification or extension to classes in the modules explained in the current section. First, a Master type orders each of the clusters it manages to execute, one at a time, then verifies if they terminated. If they did, the logs collected throughout the simulation are written to the user's disk in JSON format for later processing. When a Cluster receives an instruction to execute, it orders each Node to independently execute their part of user implemented

[5] https://www.hivessimulator.tech/app.domain.helpers.html
[6] https://www.hivessimulator.tech/scriptdocs.html

algorithm, in our case, independently realizing a MC that converges to a predefined steady-state vector. The Cluster then evaluates the group's condition and logs parameters such as file replicas that were lost or the number of nodes that disconnected. Finally, the Cluster performs maintenance-related tasks as defined the class implementation of the function, before giving control back to the Master type.

2.6 Discussion

From the usability standpoint, all our requisites were achieved. The users need only to configure two files to use the Hives simulator. One of those files does not necessarily need to be modified, and using default settings is, in most cases, satisfactory. The other one can be generated automatically with the help of a script that creates the simulation file by posing a series of questions to the user. All simulations output to JSON files containing the results of the properties monitored during the execution of Hives. JSON files are easier to read by humans in comparison with XML files. They are also widely used in web services to transmit data and object representations in text format over Hypertext Transfer Protocol (HTTP) because the format is language-independent. Concerning Hives extensions, most users will only need to create at most three classes, which likely inherit from the bundled-classes depicted in Figure 2.8. As a result, users only need to override some of the methods specified in the message sequence number seven of the diagram given in Figure A.1 and, in some cases write a few more methods to fit their requisites, particularly those that concern logging.

On the technical side, multiple cluster groups with arbitrary sizes are supported through specification in the simulation file. The network nodes can also have different uptimes, taken from a normal distribution with n samples (n is the number of nodes the user-specified during the execution of *simfile_generator.py*). To define nodes uptimes with other methods, the user can replace the script *normal_distr_sampler.py* with some other as long as it returns a mutable iterable, such as a List or NumPy

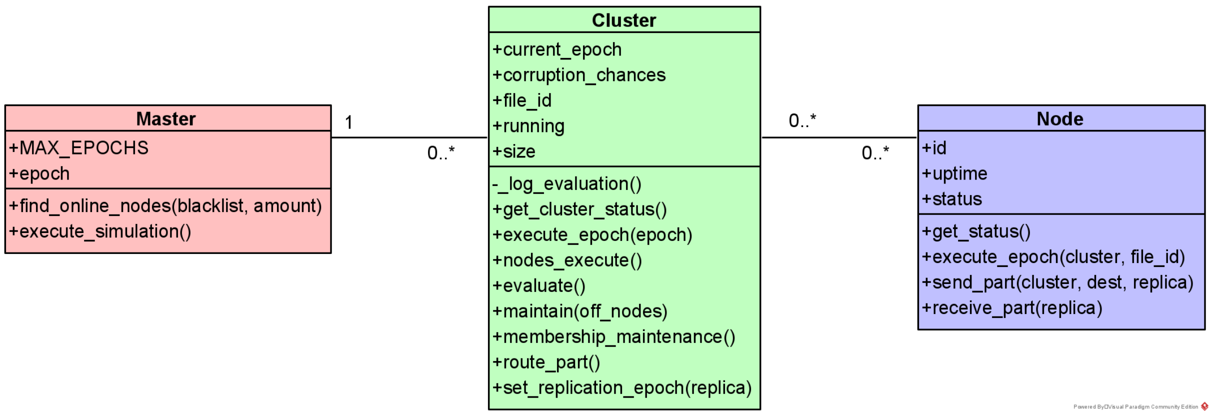


Figure 2.8: Simplified class diagrams for the Hives' main classes.

array, but not a tuple. To detect faults in the system, we query nodes' statuses during an epoch. The Status is an enumerator class that contains values such as Online, Offline, or Suspect. The interpretation of the status depends on the implementation of Node, Cluster, and Master type classes. In our use cases, a suspect node is considered offline by logger classes, e.g., Cluster and its LoggingData, but is considered as possibly online by the remaining network nodes. Suspects are only evicted from a cluster when a quorum of complaints occurs or when they miss too many heartbeats. Dynamic connectivity statuses are supported at the system level by not removing nodes from the Masters *network_nodes* attributes, but their support at a more specific level is implementation-dependent. Cluster groups also use statuses to infer the number of lost replicas during an epoch. With a gateway-like function implemented at cluster classes, messages drop with a 4% chance [34] to simulate channel loss. We create corruption of files in a disk similarly or depending on the simulated algorithm, by having each node passively corrupting their file replicas over time. We recognize that neither approach is the most accurate for simulating corruption on the disk. Still, it is the most efficient, as it avoids an extra step in the simulation with time complexity of $\mathcal{O}(r)$, where r is the number of file block replicas in the system. With current implementations, each file replica can become corrupt more than once per epoch, yet, the probability of such event is negligible. When it comes to replication and recovery, we keep one *FileBlockData* object in the simulation for each file block resulting from the split done at the Master during the initial setup process. This object contains an attribute that stores the number of network nodes that reference that file block. Hence, when a node or corruption fault occurs, the number of references is diminished by an all-knowing entity. When the remaining nodes detect the fault, a random bounded time is selected to account for the recovery. At the marked recovery time, healthy nodes with references to that file block are responsible for restoring the replication level.

Throughout our simulator's production, the priorities were usability, clarity, and the users' development speed. Conversely, simulation speed was never considered until later stages. Hence, we consider the latter to be the only requisite listed in Section 2.4, that Hives does not fulfill. Depending on the implemented algorithm, the simulator could run really slow, taking up to a few minutes to complete. To have a good point of comparison respecting this metric, we tested our engine against the one offered by the cycle-based PeerSim. For a fair comparison, we launched PeerSim 1.0.5[7] with configuration file *config-example2.txt*, included in the base distribution, only changing the number of nodes in the network. Then, we implemented and launched Hives with the same algorithms used in PeerSim's configuration file — Newscast (Section 3.2.3.B) for nodes' network views shuffling and average network out-degree aggregation at each epoch, printing it to the user's CLI. We recorded execution times in seconds, for both engines, and show them in Figure 2.9. By inspection, Hives supports, for average-case implementations, such as Newscast, networks with at most 10^4 nodes. However, we note that for our PSG

[7] https://sourceforge.net/projects/peersim/

algorithm, introduced in the next chapter, the engine struggles with as much as 10^2 nodes. Our simulator's bad performance concerning execution time derives from three aspects. The lack of centralized common states, widely used by PeerSim. Python's natural slowness with for-loop statements, which we use widely for logging purposes and to keep the code-base understandable. Perhaps, most importantly, the fact that we made no efforts concerning velocity optimizations. The engine's main loop as a time complexity of $\mathcal{O}(e \times c \times n)$, which in case of swarm guidance, becomes $\mathcal{O}(e \times c \times n \times r)$[8].

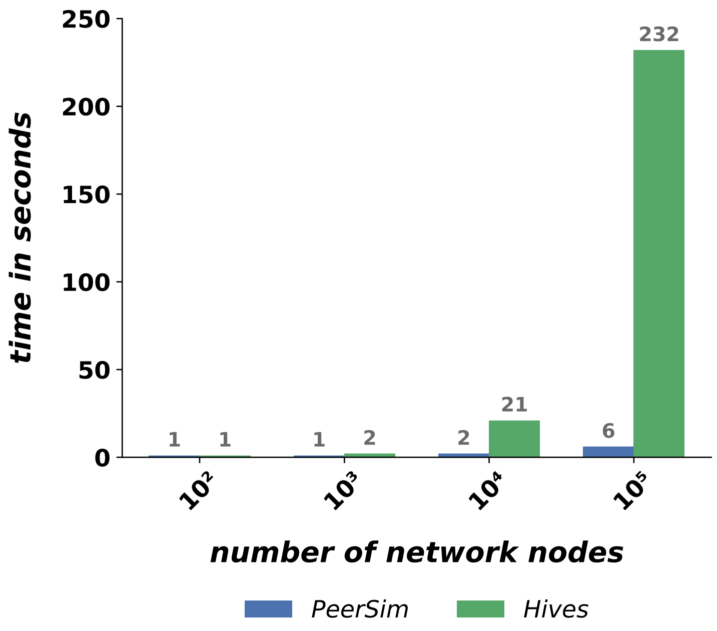


Figure 2.9: Time taken to complete 30 cycles of Newscast Shuffling and Average Degree aggregation.

2.7 Conclusion

In this chapter, we motivated the need for P2P simulators when conducting research work. We surveyed the existing simulators, presented two examples existing in the literature, and, overall, found that powerful simulators exist for Java and C# languages. Both languages lack open-source community-backed, data science, optimization, and mathematical tools. Python, on the other hand, is the scripting language that provides the best tools for these purposes, along with Matlab (an expensive alternative), but lacks simulators. NS-3 is the best option available for Python; however, being a packet-based simulator, this program's target is geared towards telecommunications and low-level protocol studies, effectively hardening the implementation of application-layer protocols, especially if multiple distributed systems paradigms need to be developed in parallel. Given these issues, we have developed a list of essential

[8] e: epochs; c: number of clusters; n: number of nodes per cluster; r: number of file blocks per node

requirements to test the algorithms proposed in this thesis. We then implemented a custom simulator suitable for fast development in a programming language that offers powerful open-source packages, named Hives. Being particularly useful for implementing DFSs of any kind, including those built on-top of P2P networks, it favors quick prototyping of application-layer programs by making use of object-oriented inheritance and polymorphism mechanisms, augmented with functional programming utility modules. Hives disregards the details of anything below and including the transport layer of the OSI Model, but provides means to account for its effects by, among other features, providing a probabilistic interface that simulates message loss and data corruption.

The entire engine source code is made available under the GPLv3 license at our GitHub repository[9]. The entire project follows Google Style Guides[10] coding and docstring conventions. Along with the repository, we ship a live documentation website[11] powered by Sphinx[12] autodocs tool.

For future work, we recognize that the environment settings module can be improved, making its responsibility even clearer. Another possibility is to optimize the code-base in order to have faster simulations, such that Hives can offer a performance on par with the more hard-to-set-up simulators without compromising its usability and clarity.

[9] https://github.com/FranciscoKloganB/hivessimulator
[10] https://google.github.io/styleguide/pyguide.html
[11] https://www.hivessimulator.tech/
[12] https://www.sphinx-doc.org/

3

File Durability with Swarm Guidance

Contents

3.1 Context

As we explained in the motivation Section 1.3, we can argue that there are two big types of architectures for distributed systems, centralized such as cloud and grid computing, and decentralized such as P2P networks. In that same section, we mentioned that in the general case, P2P systems provide cheaper implementations gifted with self-healing capabilities, and the cloud brings a better quality of service guarantees at the cost of introducing potential bottlenecks and single points of failure.

The considered application assumes the existence of a marketplace where people offer space in their disk in exchange for monetary compensation. For the shareholders of the marketplace, we could hypothesize that a full P2P system, without any central servers, other than ones that serve transactions, would be most desirable because the company would incur less infrastructural costs. On the other hand, users might want the reassurance of having their endangered files uploaded from the P2P networks to resilient cloud servers until the P2P clusters stabilize, or that metadata servers help speed up the lookup of their files. More importantly, a system's cost and performance is almost entirely dependent on underlying algorithms, resource expenditure, and use and not so much on it's centralized or decentralized nature. Recognizing the importance of underlying algorithms, we survey the literature to know how to approach the PSG problem in our DBS setting.

3.2 Related Work

3.2.1 Distributed File Systems

3.2.1.A Google File System

GFS [4] wants to provide efficient and reliable access to data using clusters located in geographically spread data-centers. In particular, GFS aims to provide high data throughput, low latency, and reliability in the face of individual server failures. GFS serves as inspiration for a widely adopted open-source project called HDFS [23]. Based on the authors' observation that once written, files on google services, are only read, often sequentially, with random writes being practically non-existent, GFS chooses to optimize throughput over latency, appends over random writes and deliberately skips data block caching. GFS clusters consist of a single master entity and multiple chunk-servers. The master knows for each file, how many chunks exist, and their current location. The master server is responsible for detecting faults over the chunk-servers and chunk-migration between them when they occur. For reads and writes, clients communicate with the master for metadata operations and directly with the chunk-servers for data-bearing operations – Figure 3.1 depicts these interactions. A read operation involves obtaining chunk handles and locations from the master and then retrieving the chunks from one of the many

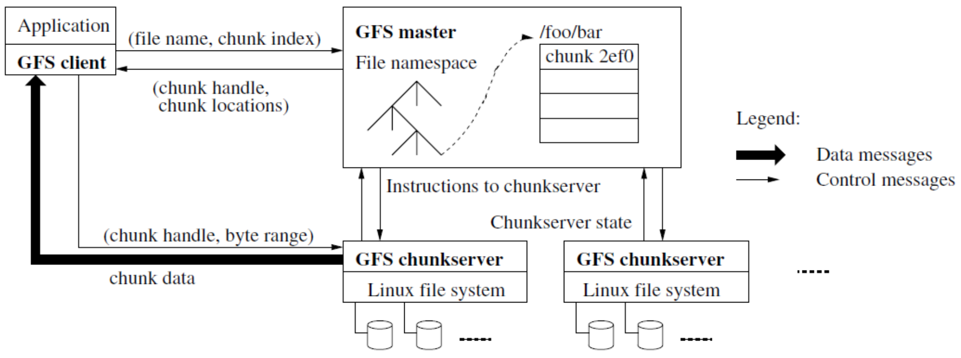


Figure 3.1: GFS Architecture [4]

available chunk-severs. For writes, the client splits the file into 64MB chunks and asks the master to assign them an identifier, he uploads his chunks to the chunk-servers indicated by the master.

GFS is a proven system, with particularly good performance concerning large-scale data processing. However, small files tend to overburden chunk-servers holding them because chunks must have a fixed large value size of 64MB. Admittedly, this might not be the case nowadays, but it certainly was so back in 2003 when the paper was published. However, the reasoning behind the design choice, which gives a substantial size value to chunks, is not senseless. The idea is to reduce network overhead, the number of accesses to the master, and memory used at the master due to mappings.

Concerning our work, convergence to a desired equilibrium using PSG occurs faster, and with more accuracy, the more parts exist on the network; thus, both EC and block-level replication are well suited for us. However, the latter option must have a significantly smaller default chunk size than the one used by GFS/HDFS. Not only because we do not want to waste storage space of peers, but also because it reduces the likelihood of having hundreds of file parts in the system. It will be clear why it is desirable to have many file blocks in Section 3.2.4.

3.2.1.B Ceph File System

CFS [24] is an open-source project that aims to deliver reliable, autonomous, distributed object, block, and file storage that is self-healing, self-managing, and has no single point of failure. The base architectural component of Ceph is the RADOS object store. All clients' reads and writes to the system are represented as objects, irrespective of the objects' original data types. RADOS is a single logical entity composed of many storage clusters. Ceph architecture is sophisticated, so, we summarize the main components. Inside a storage cluster, we find four types of logical machines, storage devices, abstracted with a layer called *object software daemon* (OSD), *monitors*, *managers*, and *metadata servers*. Metadata servers contain information used for POSIX semantics, e.g., file owner or last accessed timestamps. A small number of monitors maintain a master copy of the storage cluster map with its current state. Monitors use Paxos [61] to decide on the state of the cluster. Managers maintain detailed information about placement groups, process and host metadata. Finally, OSD store data on behalf of clients and utilize their nodes' resources to perform replication, erasure coding, load-balancing, recovery, monitoring, and reporting functions. Each Storage Cluster as anywhere from dozens to thousands of these OSDs.

Ceph's innovation is that OSD within the same *placement group*, within a cluster, are organized in a P2P fashion and are capable of autonomously carrying out recovery tasks. Placement groups are fragments of a logical *pool*, and in turn, pools are logical, dynamic partitions that define among other things, how much and which type of replication is done to objects that belong to it. When one OSD in a placement group suspects another has left or joined the cluster, it consults a Monitor to obtain an up-to-date cluster map. After running the CRUSH algorithm [26] over the cluster map, it is possible that the OSD will offload some of its content to another OSD. Clients also use CRUSH to perform reads and writes to the system. To perform a write operation, for example, a client connects to a pool in the storage cluster and runs CRUSH with a cluster map retrieved from a Monitor. The result of the operation gives the client information on the placement group of the data he wishes to upload as well as the OSD he should contact to complete his request. When the target OSD receives the request, it takes the object identifier, pool name, and the cluster map and uses CRUSH, to discover the number of replicas that it should store and to which secondary OSDs he must replicate the write. The operation is completed when the primary OSD, receives acknowledgments from the remaining secondary OSD and acknowledges back to the client. Because operations runs independently on clients and OSDs, there is no need to have the monitor constantly updating cluster to object mappings.

In Figure 3.2 we see how clients can interact with Ceph. LIBRADOS is a library that provides a convenient way of gaining access to RADOS with multiple programming languages. This library also grants access to the CFS as if it was a local partition in the users' machines or through RESTful API calls using RADOS Gateway. Additionally, clients can interact with RADOS Block Devices to perform management operations over the Ceph storage cluster (depicted below the RADOS abstraction), e.g., OSD provisioning or cluster resizing.

Ceph is proof that DFSs can reliably be implemented using P2P-like behaviors. Ceph OSDs perform a lot of self-monitoring and management operations and in a decentralized fashion while being highly dependable [25]. Like GFS, Ceph is meant to be deployed on commodity machines, managed by the same entity. In our approach, presented in Section 3.4.1, we also seek that the network nodes belonging to a cluster group perform their neighbors' monitoring and fault detection, but the machines will not necessarily belong to the same entity, meaning there is a chance for the overlay network to be less stable.

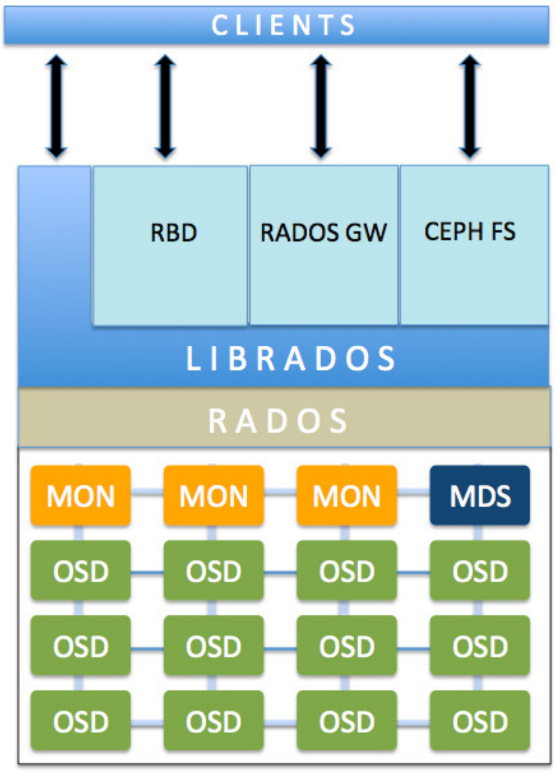


Figure 3.2: CFS Architecture [5]

3.2.2 P2P Overlay Networks

3.2.2.A Structured Overlays

Kademlia [29] is fully decentralized and designed for key-value storage systems, and according to the authors, combines provable consistency, performance, latency-minimizing routing, and symmetric topology. Kademlia peer-identifiers and data-keys are built as in Chord [62], using consistent hashing. Inserts and lookups on the overlay are unidirectional, according to the distance between two identifiers, calculated using XOR. The unidirectionality allows caching to be done along lookup paths, alleviating hot spots. Each peer in the overlay maintains a binary tree, routing table, whose leaves are lists of peers, called k-buckets, at distances 2^x to 2^{x+1}, $x \in [0, 160]$ from itself, sorted according to the least recently seen policy. As a peer receives messages, if the sender is a new acquaintance, that sender is placed on the proper k-bucket. If a replacement is required, then the least seen peer will be evicted, but only if that peer fails to respond to a ping. This policy helps in the defense against attacks where malicious entities flood the network with new dummy peers. By preferring old acquaintances, Kademlia seeks to enhance the stability of the overlay based on the observation that the longer a peer remains connected to the network, the higher the probability that it will remain connected at least another hour [63]. The XOR metric gives Kademlia an advantage over other DHT-based implementations by making routing tables less rigid and by allowing peers to learn useful routing information from messages they receive. K-buckets allow routing to be done around failed nodes, providing increased churn resistance.

We note that Kademlia does not implement an effective mechanism to keep dated entries out of a peer's k-buckets, meaning that the longer a peer stays offline, the worse is k-buckets are when he returns, because the standard implementation removes dated entries only when a peer contacts another and obtains no response.

Self-Chord [31] is a bio-inspired overlay designed for cloud or grid computing architectures. Its foundations are that of the standard Chord, both in terms of identifier generation and the way peers connect, i.e., in ring-like shapes. Self-Chord a few key differences, however, which result from having *ant* agents independently roaming the overlay but collaboratively re-arranging resource location, i.e., data placement is not based solely on the numerical-space relation between data and placement peers. The rearrangement improves lookup operation speed by clustering similar items around neighboring nodes and provides some churn resistance. Other advantages include less overlay maintenance overhead since when a peer joins or leaves the network, ants will autonomously and eventually distribute data appropriately. While redistributing data, Ants also ensure load-balancing, thus improving scalability and robustness since individual peer failures are less likely to lead to massive file loss. According to the authors, the behavior of the ants ensures that even under highly dynamic, unfavorable conditions, rearrangement, and discovery of data items takes only logarithmic time. Bio-inspired systems like Self-Chord, diminish or eliminate some of the inherent disadvantages associated with structured or unstructured overlays. In these, simple agents often accomplish complex behavior that would otherwise require time or space consuming algorithms, often with high code complexity.

We, too, seek to create a system that employs autonomous, self-healing, and self-organizing behaviors using PSG.

3.2.2.B Unstructured Overlays

Gia [30] seeks to overcome the drawbacks of Gnutella [64], which was the first decentralized, unstructured P2P protocol. Gnutella's main issues were the lack of scalability, as resource discovery was made with flooding, and how easily some peers became overloaded when faced with high rates of aggregate queries. Gia proposes the use of super-nodes, which are nowadays also part of the Gnutella protocol. The super-nodes receive and route queries to peers holding data. Super-node selection and the construction of the topology around them is dynamic. Gia uses random walks, both to alter the topology as well as perform lookup operations. Random-walks alone are less likely to find appropriate lookup responses unless they are biased to high-degree peers, which in turn can make peer overloading frequent. To solve the issue, Gia implements a topology adaptation that puts most peers within reach of high capacity peers, while simultaneously ensuring that they can handle the incoming requests. Lastly, all peers keep pointers to data items kept by their immediate one-hop neighbors to help guiding queries in the right direction. The algorithm Gia uses to ensure high-capacity-peers are also high-degree ones depends on a satisfaction function autonomously executed by each peer, which returns a value $s \in [0, 1]$. As long as $s < 1$, the peer takes the initiative of calling another random peer, preferring those whose capacity is better than his. During the handshake, any of the intervenients may abort, based on their capacity and the degrees of their neighbors. The callee accepts the caller whenever he does not know enough peers or the caller has more capacity than at least one of his neighbors; to avoid disconnecting poorly connected peers, the callee will replace the highest capacity peer whose capacity is smaller than that of the caller. In order to avoid hot-spots or overloading of peers, a peer is only allowed to send requests to a neighbor if that neighbor has explicitly given him one Token representing one request that neighbor is willing to accept. Each peer allocates tokens at the rate at which they can answer requests. The results show that Gia manages to reduce network overhead, as expected since it does not use flooding, but is also up to five orders of magnitude faster than Gnutella performing lookups.

Gia successfully promotes optimal performance and longevity of the system through continuous, peer carried optimizations. We identify one fundamental problem with Gia, related to the systems attempt to load-balance super-peers and implicit free-rider avoidance[1]: according to their capacity, peers receive tokens from their neighbors, hence decreasing the number of tokens received by low-capacity peers and, thus, system usability. Still, using a satisfaction metric can be a useful addition to our PSG algorithm to alter the rate at which each network node independently realize their MCs.

BlatAnt [32] is a bio-inspired overlay designed for grid computing architectures where resource discovery needs to be efficient; BlatAnt focuses solely on network-bounded-diameter optimization to minimize network overhead and lookup latency, using swarm intelligence to support flood-like discovery protocols. Like in Self-Chord, ant agents optimize and maintain the overlay; conversely, the ants do not change the data items' location. Instead, they reorganize peer connections so that the global overlay uses as little of them as possible. The idea is that by lowering the average path length required to obtain a data item, lookup times will reduce. BlatAnt also uses local index caching to improve efficiency further. Authors argue that local caches, as an alternative to semantic clustering such as the ones used in Self-Chord, are better for self-organized overlays where a stable topology can not be guaranteed.

The overlay works with three major components. The first is the *peers* themselves. Each peer keeps two structures, a fixed-size α-table, i.e., a partial view of the overlay retaining neighborhood information used to evaluate the redundant connections or the need for new ones, and, a neighborhood-set *N*, containing peer identifiers. Peers contribute to the optimization by rearranging local connections ac-

[1] Peers who consume more community resources than they contribute with.

cording to the connection rule, which reduces the diameter of the overlay, and disconnection rule, which discards redundant connections. The second component are the ants, which have multiple species; *discovery ants* are randomly spawned by peers and live for a limited time. They wander across the overlay, collecting information about its topology and update the α-tables of peers they meet along the way. *Construction ants* act as bootstrappers to the system. When a new peer *i* wants to connect to a peer *j*, he sends one of these ants, if *j* cannot establish the connection because he would violate a degree constraint, he forwards the ant to his lowest-degree neighbor. When *j* accepts the ant, he sends it back to *i*, and the procedure is completed, i.e., they both belong to each other neighborhood-sets. *Optimization ants* establish connections according to the connection rule. In this case, a peer *i* wanting to connect to peer *j* sends him this ant, and *j* only accepts or refuses the ant. If he accepts, he sends the ant back to *i*. *Unlink ants* remove existing connections between peers because the disconnection rule applies or due to a peer leaving the overlay. When this ant arrives at its destination, it removes all sender information from the α-table and the neighborhood-set. *Update Neighbors Ants* are spawned by a peer whenever he establishes or tears-down a connection with some other. These ants visit all of that peer's neighbors and update his information on their respective α-table. Finally, *Ping Ants* are periodically exchanged between peers to keep their connections alive in low traffic situations. The last component are *pheromone trails*, which evaporate overtime. A trail is a value assigned to a connection between peers, whenever an ant or query, walks over that connection, they increase the pheromone concentration on both ends. Discovery ants will tend to follow paths with less pheromone concentration, thus incentivizing full network coverage during the exploration. When a peer detects that for a given neighbor, its pheromone concentration has wholly evaporated, that neighbor is assumed to have left the network without warning. Eventually, each peer who neighbored the disconnected peer will independently initiate a recovery protocol by sending construction ants to the neighbors of the disconnected peer, reorganizing the overlay, and avoiding network partitioning. Thus pheromone trails provide seamless error resolution.

BlatAnt quickly converges to stable overlays with bounded-diameter, even in very dynamic conditions, resulting in faster lookups, resulting from a decrease in flood messages and caching. The protocol could further decrease network overhead by using random walks like Gia. We can use ant-like behavior to improve the discovery of faulty network nodes in our DBS.

3.2.3 Membership Management

Gossip-based protocols are known for their usability to solve problems like database replication, failure detection, and resource-monitoring. We review two frameworks that ensure nodes keep up-to-date partial views of dynamic networks. Another fundamental issue in unstructured P2P networks is how to avoid partitioning. The following alternatives are both scalable, robust, and decentralized solutions.

3.2.3.A Cyclon

Cyclon [35] is usable even in highly dynamic environments, providing node degree symmetry, low diameter, low clustering, resilience to churn, and massive node failures while being a lightweight and simple protocol. Cyclon consists of having each node keeping an ever-changing, partial view, with fixed-size *c*, of the network in a structure called *neighbors*. Periodically, independently and asynchronously, a node contacts a random neighbor to exchange acquaintances. This process is called *shuffling*. During a shuffle, a node *i*, first increments the age of all his known neighbors by one. Then, selects the oldest of all his neighbors, *j*, and $l-1$ other random neighbors, forming *s*. Then *i* replaces *j*'s network address by his own and sets his age to zero, in *s*. After that, *i* sends *s* to *j* and waits for *j* to reply with *s'*. Unlike *s*, *s'* is composed of a purely random subset of *j*'s neighbors, without any age modifications. Upon reception of subsets *s* and *s'*, *j* and *i*, respectively, update their *neighbors* by first using any empty slots in their partial views and then by replacing entries which they sent to the other, i.e., if *i* has no more slots, he replaces, in its *neighbors* structure, the entries that he sent in *s* to *j*, with the ones he received in *s'* from *j*. Note that, after *i* initiates a shuffle with *j*, *i* becomes *j*'s neighbor, but *j* is no longer a neighbor to *i*, effectively reversing neighboring relation between both nodes[2]. The age is essential in the shuffling algorithm, for two reasons. First, it limits the lifetime of each node in neighboring structures, which globally bounds the number of existing pointers to them. Secondly, it also limits the time each nodes' addresses are passed around until they are selected as shuffling targets, resulting in a more up-to-date overlay as well as uniform distribution of each nodes' addresses over the network. When a node leaves the overlay for any reason, the remaining nodes may have to remove him from their *neighbors* structure. Many authors argue that timely removal is fundamental for the robustness of the overlay, Cyclon uses a reactive pessimistic policy that works as follows: Whenever a node *i* contacts *j* for shuffling and obtains no response it assumes *j* is disconnected and removes it from his *neighbors*. Since the time for *i* to contact *j* is bounded by the age property, detection is accelerated, which is a property Gia does not have. Additionally, if *j* was not disconnected, it will remain in other nodes' views because he also initiates shuffle procedures periodically.

Cyclon simulations show that the average path length and clustering coefficient, for various configurations, converged to values similar to those found in random graphs, i.e., converged to small-diameter topologies with low clustering. Gossip-based protocols often have high clustering, which is undesirable both in terms of robustness and overhead. Furthermore, the average path length increased logarithmically, and clustering decreased exponentially, as the number of nodes in the network increased. Noting the former property, the authors propose a *join* method that does not disrupt the randomness of the obtained overlays. Whenever *i* wishes to *join* the overlay, he contacts one node *j* already in the overlay who initiates *c* random-walks with time-to-live (TTL) equal to the average path length of the overlay. The

[2] Cyclon links are not bidirectional, conversely to the majority of protocols.

node *k* where the random-walk ends, replaces one of his *neighbors* entries with *i* network address, setting *i* age set to zero and sends the replaced entry to *i*. Note that even if some of the random walks do not complete due to byzantine failures, *i* will remain connected to the overlay and will eventually find new acquaintances. Cyclon as no apparent downsides other than the one where too few nodes in the system may cause it to be underperforming due to clustering, but other solutions have this issue in common. To some extent, this framework relates to BlatAnt since it implicitly bounds the diameter of the network.

3.2.3.B Newscast

In Newscast [65], all nodes proactively exchange, timestamped, information with each other, including not only neighboring information but also app-specific information, allowing for simplified implementation of aggregation algorithms, i.e., finding statistics regarding for example network size. Newscast is thus responsible for both membership management as well as information dissemination. Within each Newscast node, there are two types of entities. Those who run the actual newscast protocol, are called *correspondents*. Those who run the distributed application are called *agents*. *Correspondents* all run the same instance of the newscast protocol. *Agents* are not required to run the same distributed application. Each *agent* has one or more *correspondents*; this responsibility separation does not explicitly exist in Cyclon. Each *agent* implements an interface with two functions *getNews()* and *newsUpdate(news[])*. The *correspondent* uses the former to request news from their *agent*, and the latter to deliver app-specific news collected from other nodes' *correspondents* in the network. Each *correspondent* as a fixed-size news cache of size *c* and periodically exchange news with each other has followed: request news to its *agent*, timestamp them along with the local time, and IP-address of the node, then store the news in a cache entry. Afterward, it selects another node's IP address running the same type of *correspondent*, as available in his cache. Both *correspondents* exchange the entirety of their caches and pass the merged cache with $2c + 1$ entries to their respective *agents* before discarding the oldest entries, after timestamp normalization, down to *c* and storing the results themselves. The timestamp normalization is not a clock-synchronization process and has some inaccuracy, but as long as transmission times between two nodes are not too big, there should be no problem. During the discarding process, the algorithm ensures that there is at most one news item per *agent* in the cache.

Newscast has a slight edge over Cyclon, due to graciously accounting for aggregation functions in its native protocol. Similarly, inactive nodes are eventually forgotten due to the timestamp property, but we believe that Cyclon has several significant advantages. These include: the resulting overlays of Cyclon have better randomness due to its shuffling-based algorithm, whereas Newscast and other gossip-based management frameworks, seem to have small-world properties according to previous research [66]. Also, Cyclon uses logical timestamps; hence, no clock synchronization is required, not even a relaxed one. Newly joined nodes' cache initialization is based on a regular news exchange with one or more nodes who are already members of the overlay. In contrast, Cyclon uses *c* random-walks exchanging exactly one cache entry with possibly different nodes, which in turn helps to maintain overlay randomness. Finally, while Newscast managed to achieve their goals of providing a robust, scalable, adaptive and lightweight solution, Cyclon exhibits less network overhead and can easily be extended to perform aggregation.

3.2.4 Swarm Guidance in Robotics using Markov Chains

We now review B. Açıkmeşe *et al.* work [15–17], which proposes PSG using MCs[3], where nodes take random actions that lead to an emerging behavior of having a formation based on a discrete probability distribution. The random decisions are governed by a row (or column depending on the implementation) of the Markov matrix associated with the current position. This algorithm does not pose in communication assumptions and exhibits self-healing properties. In the first paper, agents do not communicate and, restricted zones, i.e., areas that agents can not traverse, are enforced by creating transient states. By introducing a Gaussian method to the proposal matrix used by the Metropolis-Hastings (MH) algorithm, the transient states, are handled by creating links around the restricted zones to ensure the region graph remains connected. On the second proposal, agents can communicate. However, the communication is not to improve the speed at which the swarm converges to one goal but to allow sub-swarms of agents with different goals to collaborate with their direct neighbors using consensus procedures. The result is that the final agent formation is a weighted average of the initial goals, where the size of each sub-swarm decides the weight. Restricted zones are expressed explicitly in the adjacency matrix used in the MH algorithm. The third and last paper utilizes the previous results to gift limited mobility agents, to perform similar tasks. The key is that, instead of making each agent follow the MC to decide to where they move next, the agents use the chain to decide if they should accept or reject an environmentally induced movement.

The first paper is the most relevant to our work given that the *space* in the context of P2P is logical rather than physical, and there are only mild motion constraints that can be incorporated directly in the adjacency matrix, e.g., no transient states. We want to avoid complex behaviors, e.g., consensus, within our swarms and logical networks do not have environmental forces, e.g., wind. Hence, we adapt Algorithm 1 taken from [15] to our proposal in Section 3.4.1. The referred implementation of MH algorithm takes two arguments: a proposal matrix, $\mathbf{K}$ and a density distribution vector, $\underline{e}$. The proposal matrix can be, for example, an adjacency matrix that represents the connections between spaces in the actuation space, e.g., a warehouse with 4 zones and what zones grant access to the remainder. The density vector defines, for example, how 12 robotic agents should be distributed in the warehouse after some

[3]In Appendix B we give an introduction to MCs and how to use the theory to predict the steady-state distribution of the overall system.

time. Thus, having $\underline{e} = [0.25, 0.25, 0.25, 0.25]$ would mean that in each zone there should be 3 agents. The algorithm uses the inputs to create a random walk, $\mathbf{R}$, and an acceptance matrix, $\mathbf{F}$, to sample the elements of the resulting Markov matrix, $\mathbf{M}$, with equilibrium $\underline{e}$, to be distributed among the agents. Finally, each agent requires a copy of the Markov matrix, such that they can select a random action based on their current state, i.e., they independently chose the next location based on the current one using the probabilities found inside the row vector or column of $\mathbf{M}$ corresponding to the current region. Given the properties of the algorithm, the distribution will approach the desired one as the execution time approaches infinity. The error between the actual and desired distribution $\underline{e}$ depends on the number of agents, with a larger population representing a better sampling of the distribution.

Algorithm 1 Creating a Markov chain for PSG using Metropolis-Hastings by B. Açıkmeşe *et al.*

```
function METROPOLIS-HASTINGS($\mathbf{K}$, $\underline{e}$)
    $\mathbf{R} \leftarrow \mathbf{0}$
    $\mathbf{F} \leftarrow \mathbf{0}$
    $\mathbf{M} \leftarrow \mathbf{0}$
    $\mathtt{n} \leftarrow$ Length($\underline{e}$)
    for $\mathtt{i} \leftarrow 0$ to $\mathtt{n}$ do
        for $\mathtt{j} \leftarrow 0$ to $\mathtt{n}$ do
            $\mathbf{R}_{ij} \leftarrow \frac{\underline{v}_i \mathbf{K}_{ji}}{\underline{v}_j \mathbf{K}_{ij}}$
            $\mathbf{F}_{ij} \leftarrow$ min(0, $\mathbf{R}_{ij}$)
            if $\mathtt{i} \neq \mathtt{j}$ then
                $\mathbf{M}_{ij} \leftarrow \mathbf{K}_{ij}\mathbf{F}_{ij}$
            end if
        end for
    end for
    for $\mathtt{i} \leftarrow 0$ to $\mathtt{n}$ do
        $\mathbf{M}_{ii} \leftarrow \mathbf{K}_{jj}$ + $\sum_{k \neq j}$(1 - $\mathbf{F}_{ij}$)$\mathbf{K}_{kj}$
    end for
    trigger event BROADCASTMC $\langle \mathbf{M} \rangle$
    return $\mathbf{M}$
end function

upon event RECEIVEMC $\langle \mathbf{M} \rangle$ do at AGENT : $\mathtt{a}$
    $\mathtt{t} \leftarrow 0$
    loop $\mathtt{t} < \infty$
        $\underline{s}_i \leftarrow$ a.GetCurrentState($\mathtt{t}$)
        $\underline{s}_j \leftarrow$ a.SelectNextState($\underline{s}_i$)
        a.GotoState($\underline{s}_j$)
        $\mathtt{t} \leftarrow \mathtt{t} + 1$
    end loop
```

3.2.4.A Fastest Mixing Markov Chains

In the work of Boyd *et al.* [6], the problem of finding a fast Markov Chain is viewed as a Semi-definite program (SDP). Using an optimization algorithm to solve the SDP rather than heuristic methods, e.g., MH (Algorithm 1) or maximum-degree chain algorithms, results in theoretical substantial speed improvements. The convergence speed, i.e., the mixing rate, is determined by the Second Largest Eigenvalue Modulus (SLEM) of the transition probability matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$, given by:

$$\mu(\mathbf{M}) = \max_{i=2,\dots,n} (\mid \lambda_i(\mathbf{M}) \mid). \tag{3.1}$$

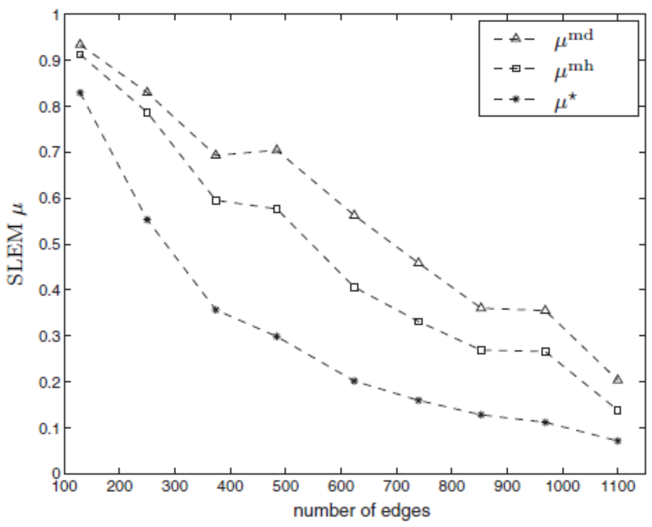


Figure 3.3: Boyd *et al.* results comparing heuristic against SDP creation of Markov chains. [6]

According to MC theory, if $\mu(\mathbf{M}) < 1$ the $\mathbf{M}$ converges to equilibrium $\underline{e}$ as time approaches infinity. Thus, the Fastest Mixing Markov chain (FMMC) problem can be formulated as an SDP, where $\mathcal{E}$ are the edges connecting pairs of vertices (i, j) on a graph $\mathcal{G}$. Denoting by $\mu^\star$ to the SLEM metric $\mu(\mathbf{M})$ of the matrix resulting from the optimization in (3.2), Figure 3.3 illustrates the difference in the convergence rate for the SDP formulation in comparison with MH (μ^{mh}) and the maximum degree (μ^{md}).

$$\begin{aligned} \underset{\mathbf{M}}{\text{minimize}} \quad & \mu(\mathbf{M}) \\ \text{subject to} \quad & \mathbf{M} \geq 0, \\ & \mathbf{M} \cdot \underline{\mathbf{1}} = \underline{\mathbf{1}}, \\ & \mathbf{M} = \mathbf{M}^{\intercal}, \\ & \mathbf{M}_{ij} = 0, \qquad (i,j) \notin \mathcal{E} \end{aligned} \tag{3.2}$$

The main issue is that the optimization targets the uniform equilibrium distribution given that matrix $\mathbf{M}$ is symmetric, which in the case of a DBS, would equate to giving the same approximate number of file blocks to each of the machines belonging to a cluster group, regardless of the quality of the underlying machine and its current resource usage. The use of non-uniform vectors as equilibrium for a MC results in a non-convex problem in general. Boyd *et al.* provides a workaround for this issue, although other techniques could be employed.

3.3 Problem Statement

Let us consider a set of storage nodes, S, participating in a P2P network to hold B file parts (blocks or encoded fragments) belonging to a file F. Their objective is to achieve the durability of F while maintaining the file accessible. Assume that message integrity, confidentiality, and authenticity are not at risk, that there are no malicious attacks on the system and that nodes do not deviate from the defined algorithms intentionally or otherwise. Admitting that message loss may occur when nodes communicate with one another, that they may suffer from disk errors, potentially corrupting file replicas, and they may disconnect at any given time. We must create a distributed algorithm for a P2P-based DBS. The system must be based on PSG algorithms used in Robotics. The problem involves generating an overlay topology, represented by a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, that defines edge-connections between pairs in S or alternately by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{S \times S}$. Topology rearrangements are only allowed when nodes in S leave or join the system, and they must be connected, i.e., network partitions are not allowed except due to failures. There are no constraints with whom peer nodes may communicate as long as any first message, in a sequence of exchanges, must result from a probabilistic event. Our algorithm must also use a policy that diminishes or delays the likelihood of losing parts from B. While the objective is to create a reliable DBS, there is limited space for centralized components and, these must-have minimal functionalities, since the cost benefits of P2P approaches lies in maximizing the utility of participating machines, S. In other words, centralized components can not: i) store data files; ii) directly monitor participating peers; iii) perform computationally intensive operations.

3.4 Proposed Solution

3.4.1 Probablistic Swarm Guidance from Robotics to Distributed Systems

As mentioned, we seek to study PSG in a distributed storage environment and see if files can survive in the system, i.e., we want to know if files uploaded to a remote set of peers working with a PSG algorithm become durable. From a swarm guidance perspective, our proposal is a mixture of the problems studied in B. Açıkmeşe *et al.* [15, 17] and the FMMC problem in Boyd*et al.* [6]. We need to create a MC that guides agents in the system to a desirable formation in the actuation space. The equilibrium vector $\underline{e}$ defines that formation. In our case, the actuation spaces are overlay networks, our regions' bins are network nodes belonging to those overlays, and our agents are files and their replicas. Because files are motionless, and the network links do not exert wind-like forces on them, we must account for this difference and adapt B. Açıkmeşe *et al.* ideas. Another quirk we need to be aware of is that overlay topologies frequently change due to churn or other machine malfunctions. Consequently, we want to select the fastest possible MC to increase the odds that the goal $\underline{e}$ (desired file distribution among the storage nodes) is reached before such an event occurs, thus increasing the system's reliability, the files' availability, and fair load-balancing among clusters' nodes. Boyd *et al.* SDPs are not directly applicable because our desired $\underline{e}$ is not likely to be uniform and even if this was the case, heuristically created MCs are not always slower than SDP ones. Because of the non-uniformity of $\underline{e}$, we may even encounter non-convex problems.

3.4.2 Markov Chain Optimization

In order to attain a good performing DBS service that ensures the durability of uploaded files, when using PSG, two key aspects are tightly related to the algorithm itself. The first is the selection of a suitable equilibrium; this means appropriately filling the entries of $\underline{e}$, taking into account the past behavior of the storage nodes, their hardware capabilities, and perhaps usual working hours in order to minimize the probability of losing file block replicas or encoded fragments. We do not explore this subject in our dissertation. The second key aspect is to utilize a MC that converges quickly to the selected equilibrium,

Algorithm 2 Creation and selection of the fastest Markov chain, from a pool of different choices.

```
function CREATEMC($c$)
    ▷ Random uniform, symmetric and connected matrix with self-loops.
    $\mathbf{K} \leftarrow$ NewTopology($c$)
    $\underline{e} \leftarrow$ NewEquilibrium($c$)

    ▷ Create pool of options
    $\mathbf{K}_{opt} \leftarrow$ SDP.Solve($\mathbf{K}$)                                  ▷ Equation (3.3)

    choices $\leftarrow \emptyset$
    choices $\cup$ METROPOLISHASTINGS $\langle \mathbf{K}, \underline{e} \rangle$                   ▷ Algorithm 1.
    choices $\cup$ METROPOLISHASTINGS $\langle \mathbf{K}_{opt}, \underline{e} \rangle$
    choices $\cup$ GO.Solve($\mathbf{K}$, $\underline{e}$)                              ▷ Equation (3.4)

    ▷ Select the fastest option
    $\mu^\star = \infty$
    $\mathbf{M}^\star = \varnothing$
    for all $\mathbf{M}$ in choices do
        if $\mu(M) < \mu^\star$ then
            $\mu^\star \leftarrow \mu(M)$
            $\mathbf{M}^\star \leftarrow \mathbf{M}$
        end if
    end for

    return $\mathbf{M}^\star$
end function
```

to minimize the time the file-hosting cluster spends in unideal states. Since optimal solutions are hard to produce, we use multiple techniques, including relaxed constraint global optimization, to deal with non-convex cases, to produce sets of feasible MCs, and then choose the one with the fastest SLEM as depicted in Algorithm 2.

$$\begin{aligned} \underset{\mathbf{K}_{opt},\, t}{\text{minimize}} \quad & t \\ \text{subject to} \quad & \mathbf{K}_{opt} = \mathbf{K}_{opt}^{\mathsf{T}}, \\ & \mathbf{K}_{opt} \geq 0, \\ & \mathbf{K}_{opt} \cdot \underline{\mathbf{1}} = \underline{\mathbf{1}}, \\ & \mathbf{K}_{opt} \odot (\mathbf{1} - \mathbf{K}) = \mathbf{0}, \\ & -t \cdot \mathbf{I} \preceq \mathbf{K}_{opt} - \frac{1}{n} \cdot \mathbf{1} \preceq t \cdot \mathbf{I}, \quad t \in \mathbb{R} \end{aligned} \tag{3.3}$$

The first technique we use to approach this problem is comparable to Algorithm 1 (MH function), using symmetric, connected adjacency matrices, $\mathbf{K}$, with obligatory self-loops as a proposal matrix and targeting $\underline{e}$. This enforcement is done for all methods, which also work with possible self-loops and no self-loops, but from our initial experiences, having enforced self-loops revealed a tendency to create faster chains; this also reduces the algorithm's network bandwidth footprint. Our second technique leverages the fact that uniform distributions, $\underline{u}$, are likely to create clean SDPs, so we first optimize $\mathbf{A}$ to $\underline{u}$ by minimizing the eigenvalues of the matrix; then, we use the output, Equation (3.3), as a proposal matrix targeting $\underline{e}$, again using the heuristic method, MH. Our final technique, Equation (3.4), resolves around global optimization, where our constraints force the output matrix to converge to $\underline{e}$. The optimization directly targets $\underline{e}$ because we allow the optimization variable to be asymmetric. Note that neither of our two mathematical optimization models guarantee a globally optimal SLEM. The former technique is local-optimization based, and, conversely, the latter is global-based. However, the objective function minimizes the norm of the output matrix, which is only a relaxed approximation of eigenvalue minimization.

$$\begin{aligned} \underset{\mathbf{M}}{\text{minimize}} \quad & \left\| \mathbf{M} - \frac{1}{n} \cdot \mathbf{1} \right\|_2 \\ \text{subject to} \quad & \mathbf{M} \geq 0, \\ & \mathbf{M} \cdot \underline{\mathbf{1}} = \underline{\mathbf{1}}, \\ & \mathbf{M} \odot (\mathbf{1} - \mathbf{K}) = \mathbf{0}, \\ & \underline{e}^{\mathsf{T}} \cdot \mathbf{M} = \underline{e}^{\mathsf{T}}, \quad \underline{e} \in \mathbb{R}^n \end{aligned} \tag{3.4}$$

Altough neither of the mathematical optimizations guarantees the absolute fastest chain, the resulting speed-ups may be significant, as can be deduced by looking at Figure 3.4. For the presented box plots, we created one thousand $\langle \mathbf{K}, \underline{e} \rangle$ pairs, and for each of those pairs, we created a MC using all the described methods for different-sized clusters. We repeated the experiment in a MATLAB[4] framework, with one thousand more different pairs. The results, were similar, so the conclusion is that optimizations should be, in theory, worth it, regardless of the size of the network, but principally, as they grow, except for our second approach, which brings negligible benefits. We will revisit this topic in Section 3.6 and see if, in practice, the optimizations are as good as in theory.

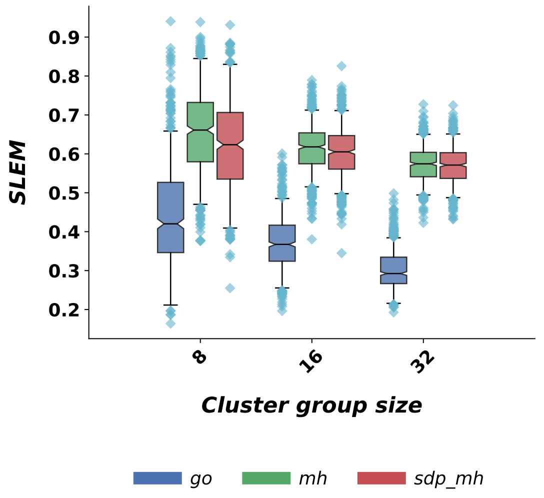


Figure 3.4: Comparing Markov chains' mixing rates with our heuristic and optimization implementations.

3.4.3 Swarm Guided Distributed Backup System (SGDBS)

Our solution is composed of five entities: *Master servers*, *Clusters*, *Monitors*, *Storage Nodes*, and *Clients*. Master servers, H, are responsible for handling client and contributor registration and authentication and mapping clients' files to clusters and keeping a record of each clusters' members. Clusters are groups of storage nodes who contribute with their private storage space to the DBS. A cluster maintains one single file with a predefined replication level, r, on behalf of precisely one client. Consequently, whenever a client, C, wishes to upload a file F, to the system, he first sends the file metadata to H_k, who will reply with a subset S of storage node identifiers with size n, along with the ideal equilibrium vector $\underline{e}$. Upon receiving the response from H_k, C divides F into multiple blocks, B, and creates a random uniform symmetric and connected topology, $\mathbf{A}$, and uses it along with $\underline{e}$, to generate Markov matrix, $\mathbf{M}$, using Algorithm 2, which he then slices into n row vectors, each being sent to the respective S_n storage node, to be used as a routing table for F's blocks. C also sends one replica of each B_i to the r closest storage nodes. From this moment onwards, at every epoch, which does not need to be synchronized, each S_n uses the routing table associated with F to send one or more Bi to other members of the cluster. This process virtually ensures $\mathbf{M}$'s synthesis and the eventual distribution of files according to goal $\underline{e}$.

[4] https://www.mathworks.com/products/new_products/latest_features.html

Monitors, O, are a set of high-reliability servers but could be sets of dedicated commodity network nodes instead, like in CFS [24]. Either way, they do not contribute with storage, but rather, receive complaints from storage nodes about their direct cluster neighbors; When monitors receive a quorum of complaints respecting a node, the complaining cluster's members evict the complainee. Eventually, another storage node will replace the ousted one in the cluster. When a node leaves or joins a cluster group, the Monitors create and broadcast a new MC. Ideally, this replacement would be direct so that the MC could remain the same or only be slightly change to avoid a complete block density distribution restart. Ideally, this replacement would be direct so that the MC could remain the same or only be slightly change to avoid a complete block density distribution restart. For simplicity, we restart the entire algorithm on membership changes and assume that disconnected nodes will never rejoin the system, which degrades the performance of our solution. We note that EC is likely a better pair for PSG, however, we favor the use of block-level replication to provide fair comparisons against HDFS in Section 3.6.

3.5 Implementation

We implement our solution using the Python 3.7[5] programming language. To facilitate package dependency management, we use PIP[6], and to evaluate the solution's performance, we utilize Hives (Chapter 2), a cycle based simulation framework for Python, developed by us. As a consequence of implementing the code in a simulator, the solution implementation is merely an abstraction of a real-world application. Implementing the system in a live-environment would require more time than we have available for the dissertation development. Our Master servers' implementations adhere to the used simulator's specifications. In particular, they act as record-keepers that indicate which storage nodes are currently online, such that clusters may find replacements for their faulty nodes. However, they do not have any registration or authentication mechanism, as we pretend that, during the simulation, any available storage node has gone through these processes at a prior time. Our storage nodes, named *SGNodeExt*, inherit from the default *Node* class offered by the simulator. The same principle applies to our Cluster implementation, called *SGClusterExt*. Our clusters, apart from performing the simulator's described roles, also emulate the initial file's write operation to the r closest storage nodes, done by a client C, as well as the role of a set of Monitors who register the members' complaints, deciding which suspicious members to expel from the complainers' cluster, the resulting member substitution and chain recalculation. In Algorithm 3 we provide the routines carried by these entities. Consult the detailed source in the simulator's website[7] or GitHub page[8].

[5] https://www.python.org/downloads/
[6] https://pip.pypa.io/en/stable/
[7] https://www.hivessimulator.tech/
[8] https://github.com/FranciscoKloganB/hivessimulator

Algorithm 3 Summary of the implemented solution's algorithm

```
▷ Emulate client write request.
upon event ExecuteSimulation ⟨$json$⟩ do at SGMaster : m
    cSize, fid, blocks ← IO.Read($json$)
    cluster ← m.NewClusterGroup(cSize, fid)
    trigger event SpreadFiles ⟨cluster, blocks⟩.
upon event SpreadFiles ⟨$B$⟩ do at SGClusterExt : c
    M ← CreateMC ⟨c⟩                                              ▷ Algorithm 2
    for all s in c.members do
        if s ∈ NEAREST($r$) then
            s.files ∪ {c.fid, $\{b_1, \ldots, b_k\}$}
        end if
        trigger event SynthesizeMC ⟨s, c, $M_{s\star}$⟩
    end for
▷ Storage nodes follow the swarm guidance every epoch time.
upon event SynthesizeMC ⟨$c$, $\underline{s}$⟩ do at SGNodeExt : s
    every $\Delta t$ do
        for all b in s.files[$c$.fid] do
            dest ← SelectNextState($\underline{s}$)
            trigger event ReceivePart ⟨dest, $c$, b⟩
            async wait response then
                if response ∈ {OK} then
                    s.files[$fid$].Delete(b)
                else if response ∈ {BAD_REQUEST} then
                    trigger event Replicate ⟨$c$, b⟩
                    s.files[$fid$].Delete(b)
                else if response ∈ {NOT_FOUND, TIME_OUT} then
                    trigger event Complaint ⟨$c$, s, dest, $\Delta t$⟩
                end if
        end for
▷ Node processes a receive part request.
upon event ReceivePart ⟨$c$, $b$⟩ do at SGNodeExt : s
    if ¬ Sha256($b$.data) then
        return BAD_REQUEST
    else if $b$ ∈ s.files[$c$.fid] then
        return NOT_ACCEPTABLE
    else
        s.files ∪ {c.fid, $b$}
        return OK
    end if

▷ continues on the next page!
```

3.6 Discussion

In the present section, we display and discuss results related to Algorithm 3, implemented on the Hives Chapter 2 simulator. We divide our study into two parts. In the first part, we consider an idealized environment to assess the baseline performance of the algorithm in perfect situations such that we can answer questions that do not depend on the network characteristics or machine quality. In the

```
▷ Monitor processes a cluster's complaint and restarts the swarm if required.
upon event COMPLAINT $\langle complainer, complainee, \Delta t\rangle$ do at SGCLUSTEREXT : c
    complaintId $\leftarrow$ $complainer \mid complainee$
    if complaintId $\notin$ c.epochComplaints($\Delta t$) then
        c.epochComplaints[$\Delta t$] $\cup$ complaintId
        c.complaints[$complainee$] + 1
        if c.complaints[$complainee$] $> \frac{\text{c.size}}{2}$ then
            c.members.Replace(complainee)
            $\mathrm{M} \leftarrow$ CREATEMC $\langle$c$\rangle$
            for all s in c.members do
                trigger event SYNTHESIZEMC $\langle$s, c, $\mathrm{M}_{s\star}\rangle$
            end for
        end if
    end if
```

second part, we simulate conditions close to how real-world problems that may affect the algorithm and make a fair comparison against a simulated HDFS or GFS like architectures. In both parts, every test scenario lasts a day with 480 epochs, i.e., an epoch occurs every three minutes. A file is considered durable if the scenario plays until the end of the 480^{th} epoch without unexpected errors. We detail all of the different played scenarios in Table C.1. The table identifies scenarios by prefixing the tested system acronym[9], followed by the number of storage nodes in the simulated cluster, suffixed by a short, acronym-like, description that summarizes the key property or features that distinguish the scenario from the remainder, e.g., one hundred parts would equate to the suffix of **100P**, a test where messages can be lost in transmission would equate to the suffix of **ML**, likewise a system using optimizations would be tagged with **Opt**. We also use these identifiers in the Figures of the remaining sections. For the remainder of this chapter, any time we verify instantaneous convergence at some epoch, we mean that, during the logging stage of a simulation, Equation (3.5) held, where $\underline{v}_s$ is the current part density within a cluster storage node. We may also imply that a cluster achieved its goal by considering Equation (3.6) instead. Associated with the latter case, we also measure the cluster distance to the desired equilibrium, as in Equation (3.7).

$$c_t \Rightarrow \left|\underline{v}_s^{(t)} - \underline{e}_s\right| \leq a_{tol} + 0.05 \times |\underline{e}_s|\,, \quad \forall s \in S \wedge t \in T \tag{3.5}$$

$$c_{avg} \Rightarrow \left|\frac{\sum_t \underline{v}_s^{(t)}}{t_d} - \underline{e}_s\right| \leq a_{tol} + 0.05 \times |\underline{e}_s|\,, \quad \forall s \in S \wedge t \in T \tag{3.6}$$

$$c_{dm} = \sqrt{\sum_s \left|\frac{\sum_t \underline{v}_s^{(t)}}{t_d} - \underline{e}_s\right|}, \quad \forall s \in S \wedge t \in T \tag{3.7}$$

[9] SG: Swarm Guidance algorithm tested on a close to, or perfect environment;
SGDBS: Swarm Guided Distributed Backup System;
HDFS: Hadoop Distributed File System;

$$a_{tol} = \min\left(\frac{1}{dim(\underline{v})}, 0.05\right), \quad S = \{s_1, \ldots, s_{dim(\underline{v})}\}, \quad T = \{t_1, \ldots, t_d = t_{MarkovChainDuration}\}$$

For the first part, in Section 3.6.1, each test scenario plays 100 times. We selected disk error probability $P(de) = 0.0\%$ and the probability of losing a message in the network links, $P(ml) = 0.0\%$. The replication factor is $r = 3$ in all cases, but we do vary the maximum block size in bytes, $b(F)$, to vary the number of file blocks in the network. For simplicity, we will refer to the total number of file blocks and their replicas as *parts*. We also activate or deactivate the optimizations to MCs to investigate if they produce any practical effects. Using the simulation file, we set the storage nodes uptime to be 100.0%, i.e., they are always on and never disconnect; consequently, we run a Monte Carlo simulation with a predefined pool of $\langle \mathbf{K}, \underline{e} \rangle$ pairs designated as challenges, which can be solved by any of our three methods. This is useful, as it ensures that different scenarios run challenges of equal difficulty at the same respective playthrough $p \in [1, \ldots, 100]$. Ultimately, through the simulation file, we vary cluster sizes.

For the second part (Section 3.6.2), we set disk error and link loss according to the reviewed literature [34, 60], thus creating fault conditions that can either slow down or stop the system functionality. The replication factor is kept at $r = 3$ and we fix $b(F) = 1$MB for both systems. We do this because HDFS documentation discourages the usage of smaller block sizes, and we want our tests to have as many similarities as possible. While the theory argues that using more parts reduces the distance to $\underline{e}$ in the long run, it does not indicate that the equilibrium is reached faster. As a matter of fact, using extra parts might increase the time it takes for the first convergences to occur due to the increased distance to the goal at the start of the simulations. Consequently, using smaller block sizes and thus, having a greater number of parts in Swarm Guided Distributed Backup System (SGDBS) is not necessarily favorable for us. Also, for similarity, more resilient storage nodes will receive a bigger quota of parts to safeguard, hence, we do not use random equilibrium vectors. We base our notion of machine resiliency purely on the time a node remains online throughout a simulation. For both SGDBS and the HDFS system, we test three different scenario that differ with respect to storage node uptimes, $u_k(s), \forall s \in S$, hence we distinguish them by tiers. For the first scenario and tier we have $u_{T1}(s) \in [4, 32]$, correspondingly, we have $u_{T2}(s) \in [32, 64]$ and $u_{T3}(s) \in [64, 100]$. The optimizations proposed in Section 3.4.2 are always executed in these scenarios for the SGDBS simulations, meaning that, we pick the fastest $\mu(\mathbf{M})$ from the pool of available MCs when a membership change occurs. Also, due to membership changes, our Monte Carlo simulations have 500 samples rather than 100, to reduce result variance. Finally, fault detection is not immediate and depends on the running protocol, i.e., complaints *vs.* heartbeats. Once detected, if they concern disconnected storage nodes, t_{snr}, an immediate replacement takes place; otherwise, if they concern lost file block replicas, these will take three to nine minutes, i.e., $t_{brr} \in [1, 3]$ epochs, to be possibly restored to no more than r.

3.6.1 Swarm Guidance on Hives

One issue associated with the use of PSG in a DBS would be the bandwidth consumption, comparing with other approaches that do not regularly change the location of file parts as a feature of the underlying algorithm. Figure 3.5 presents the bandwidth consumption for simulations with different number of parts. Irrespective of this choice, the members exchange ≈ 80MB on average in messages containing file block replicas at every epoch, excluding TCP/UDP headers, message fields like block identifying data, response and complaint messages. A clear disadvantage compared to HDFS, which may be relevant in some scenarios. When implemented with storage nodes (peers) spread worldwide, link saturation is unlikely. However, if all nodes are in the same building and multiple clusters exist, issues may arise not only in the DBS but also in other systems not associated with the backup service.

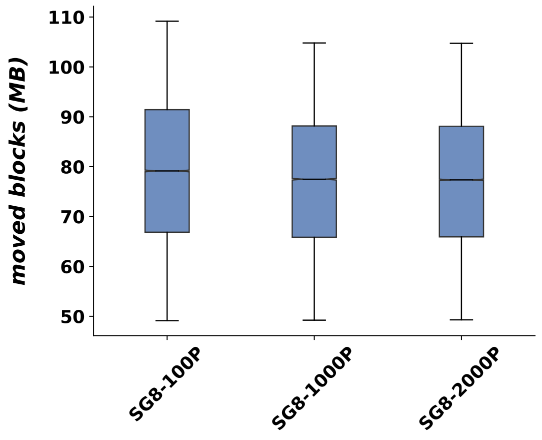


Figure 3.5: Bandwidth consumption in MB, on a epoch basis for clusters persisting one 45MB file.

On the other hand, HDFS and likely GFS, only perform checksum verifications when a client retrieves a block from a remote DataNode. This means that file corruption due to faults in a storage device, the network, or the software is unknown until a client accesses his files, which can be infrequent. By verifying checksums whenever forwarding the parts, the chance of losing files to corruption is minimized. Another advantage is the self-healing property of PSG, which allows clusters to recover from faults transparently. Depending on the used topology, permanent faults are also detected and dealt with quicker (Figure 3.6[10]). Tweaks to reduce bandwidth consumption include: a Gia-like satisfaction metric; adjusting epoch-length statically or dynamically; not allowing parts received at some storage node at some epoch to be considered for routing before the next epoch; using EC instead of block-level replication.

Since the number of parts in the system does not appear to influence bandwidth consumption, the next step is to determine how many parts should exist in the cluster. An increased number of parts implies that somewhere in the system, centrally or otherwise, more metadata will exist. Our solution

[10]Random topologies can result in nodes with low in-degree, increasing time to detect their failures.

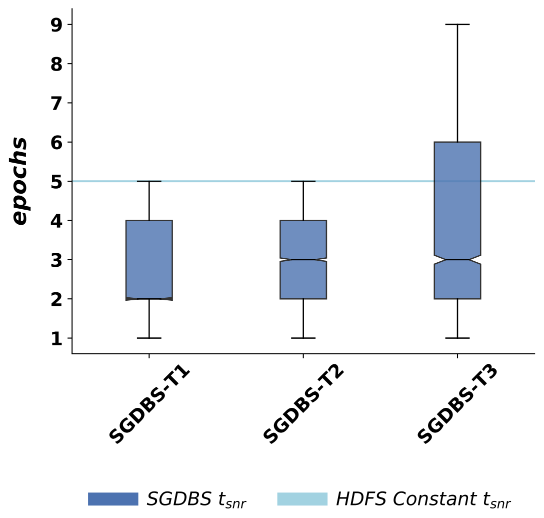
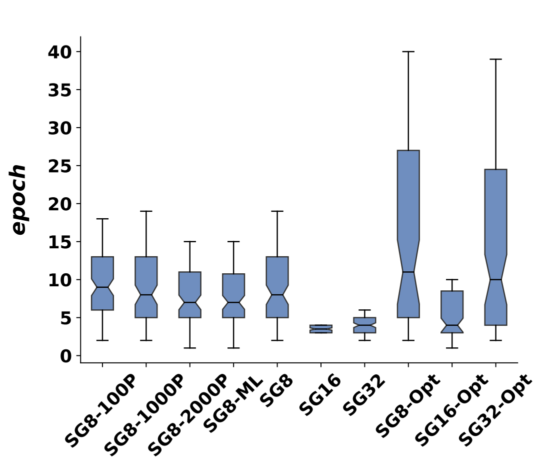


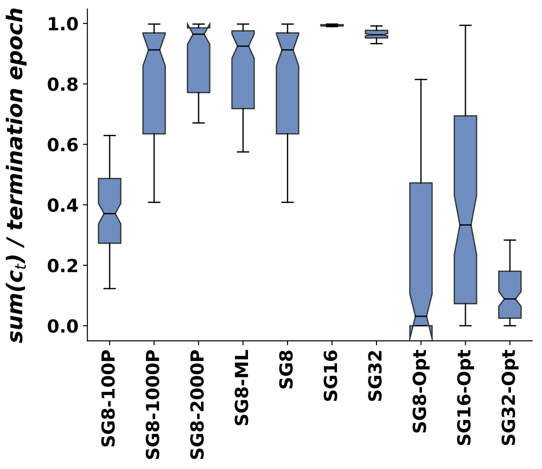
Figure 3.6: Epochs required to detect and replace permanently faulty storage nodes.

has master servers map file identifiers to clusters containing the blocks without tracking which storage nodes have which blocks, thus eliminating a big chunk of queries the masters would otherwise receive. However, the masters or some other entity in the system must maintain for each existing file the hash values associated with its blocks, to check for disk and message corruption. Consequently, increasing the number of parts without bounds is undesirable, as the metadata file would occupy more space. Furthermore, if we were to use EC to minimize bandwidth and disk usage, the reconstruction of lost replicas could become painfully slow (more parts to collect and decode), which endangers durability. Our results indicate that the number of parts does not affect the time it takes to achieve the first convergence (Figure 3.7) in a cluster. Truthfully, all scenarios had very satisfying results taking what equates to something between 30 to 90 minutes to achieve the desired configuration for the first time. Conversely, using approximately ≈ 1000 parts produces the best balance between the number of blocks that require tracking, the witnessed instant convergences, and the number of times the goal equilibrium is achieved, on average, as well as the distance to that goal (Figure 3.8).

As expected, there is a tendency for the first convergences to occur relatively early in a cluster life-cycle and for the number of observed instantaneous convergences to increase as simulations progress. Figure 3.9 shows that the number of occurrences tends to become bounded after a certain point in time. In the aforementioned figures, the inequalities used to declare convergence Equations (3.5) to (3.7) cause clusters with 16 and 32 members to be faster as the entries in the distribution vector are smaller values. If other functions are used results could differ. An example would be using only the relative tolerance in Equation (3.5), i.e., considering only the amount of parts in the cluster, never considering the number of nodes. From these simulations, the use of bigger networks contributes to better load-balance for the same replication factor as well as fewer storage node isolation situations, i.e., they make for more

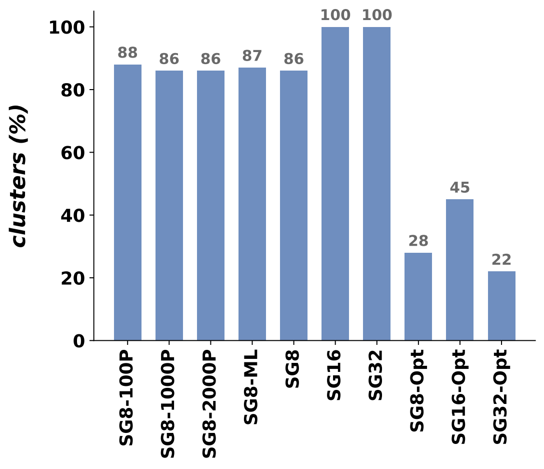


(a) epoch at which c_t boolean condition was first verified.

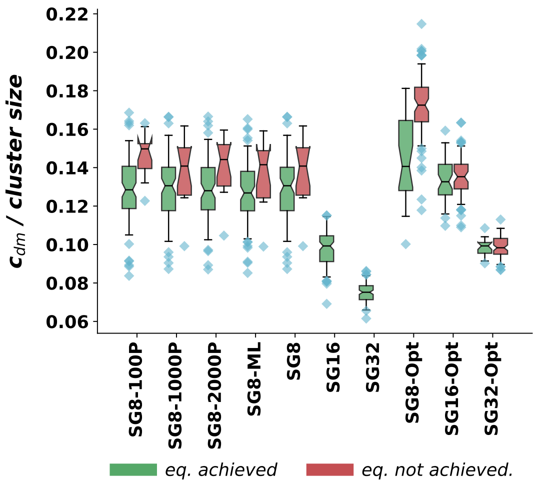


(b) clusters lifetime (%) where c_t boolean condition was verified. In this case termination epoch is fixed for all cases.

Figure 3.7: Overview of instantaneous convergence (c_t) behavior.



(a) percentage of clusters that verified the boolean condition c_{avg} at the end of a playthrough.

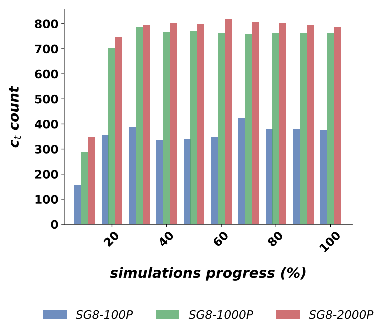


(b) clusters distance to the goal at the end of a playthrough, based on Equation (3.7) (c_{dm}).

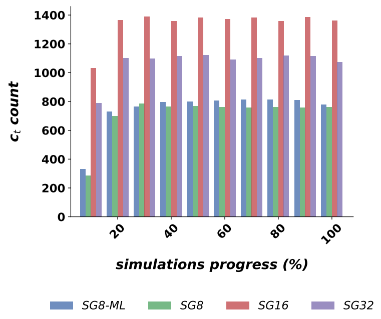
Figure 3.8: Clusters (%) achieving or not achieving the desired equilibrium and registered distance to that goal.

resilient swarms, greatly reducing the number of outliers in Figure 3.6. Another unexpected behavior regards the optimization algorithms used to produce MCs with better SLEM. We expected results to be better than the ones obtained using MH when it comes to reaching and stabilizing around the desired equilibrium. However, they were strictly worse. This is an issue that requires further research.

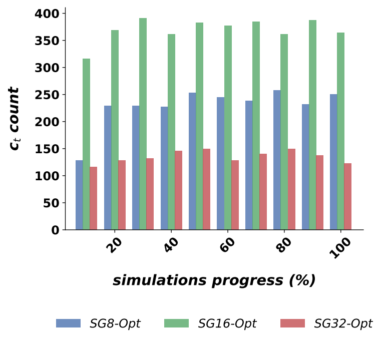
Lastly, message-loss seems irrelevant concerning the studied properties, including time taken to



(a) varying only the number of parts.



(b) varying cluster size, or alternatively allowing for messages to be lost.



(c) varying cluster size with $\mathbf{M}$ optimizations activated.

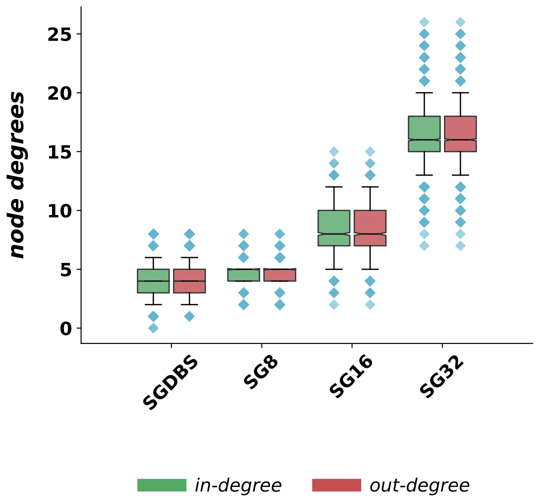
Figure 3.9: Number of instantaneous convergence occurrences as simulations progress, per scenario.

converge, the number of instantaneous convergences, and goal achievement. In a real-world scenario, they may impact system availability and reliability and even more so the durability of the files.

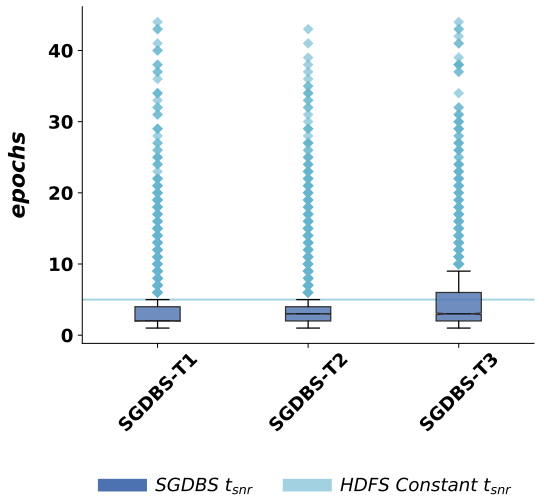
3.6.2 Swarm Guidance DBS vs. Hadoop Distributed File System

In Chapter 1, Section 1.2, we identified properties that distributed systems should support. However, it is often impractical to maximize all such properties. For example, the Eric Brewer's theorem says that in the presence of network partitions, a system either supports availability or read consistency. This does not mean that they are mutually exclusive when not in the presence of network partitions. Consequently, choices depend on the addressed scenario. Given that the considered DBS community is not guaranteed to remain online for long periods, the main critical purpose is to persist user files for arbitrarily long periods in dynamic networks, at a low cost, with few centralization. Consequently, we focus on file survivability metric when in the presence of churn.

From the surveys in Section 1.4.2 and the reviewed work in Section 3.2.2, system robustness is linked to node degrees. In our case, we do not need to concern ourselves with node clusters or hubs because our solution is not gossip-based. However, we still want to avoid weakly connected storage nodes, particularly regarding the number of receive channels they have. Put simply, if no node ever tries to contact a failed node, he shall remain undetected. Note that, while Algorithm (1) and (3.3) result in necessarily symmetric matrices, Algorithm (3.4) does not, hence the importance of in-degree, more so than the out-degree. We explain the outliers in Figure 3.10(b), essentially with the node degrees seen in Figure 3.10(a). Even though our system often outperforms HDFS concerning the time it takes to detect and replace failed nodes, a median in-degree of four, sometimes as low as two, implies that it takes only three node failures for some other to become isolated, a grim scenario when our main objective is to guarantee the durability of uploaded files regardless of peer uptime. Apart from that, new topologies



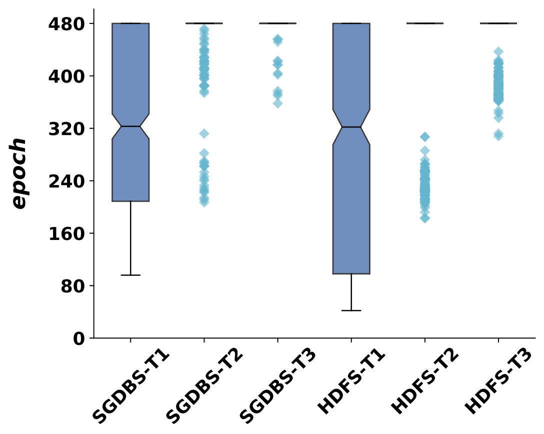
(a) Node degrees with and without optimizations.



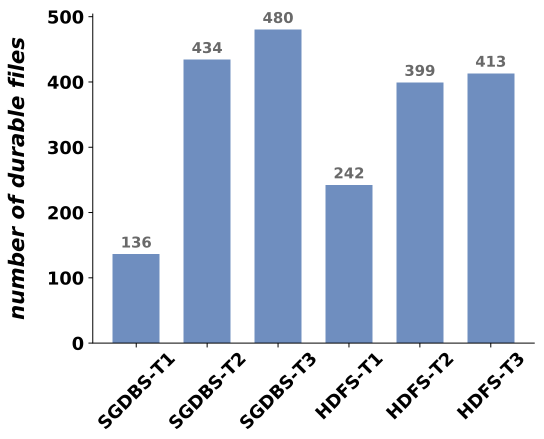
(b) Time to replace faulty nodes, including outliers.

Figure 3.10: Node degrees resulting from different configurations and their impact on t_{snr}.

give no guarantee that a failed, yet undetected, node will have a good in-degree in the next chain, which in turn, might postpone its detection. This situation is particularly relevant when clusters suffer a high churn. We propose that future work takes this problem into account during the topology generation procedure. Additionally, working storage nodes that do not receive messages for extended periods should suspect they are isolated and broadcast their departure from the cluster or request a topology change. Our approach, contrary to the expected, was less successful than HDFS on tier one machines

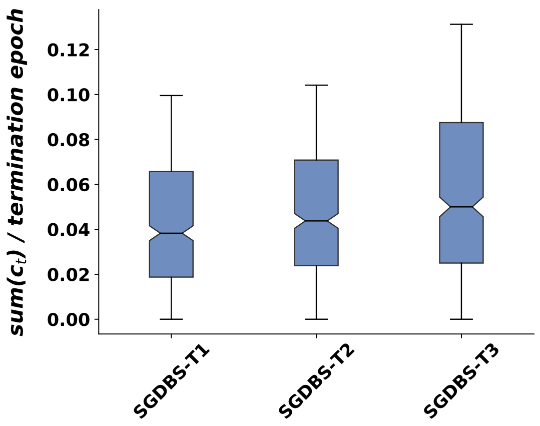


(a) simulations termination epochs distribution.

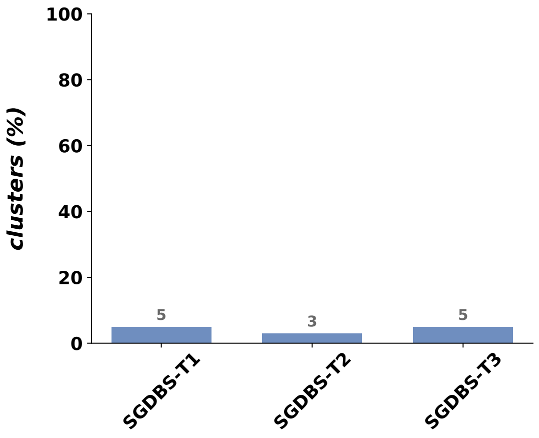


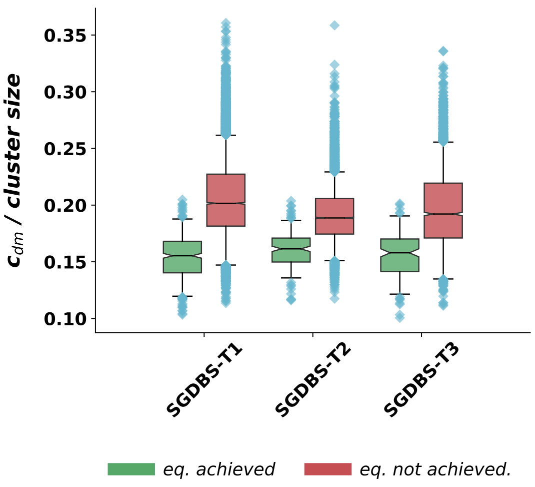
(b) simulations reaching the maximum number of epochs, thus guaranteeing durability.

Figure 3.11: Data respecting termination epochs and playthroughs which successfully ensured file durability.



(a) clusters lifetime (%) where c_t boolean condition was verified.





(b) percentage of clusters that verified the boolean condition c_{avg} at the end of a playthrough or after a membership change.

(c) distance to the equilibrium goal whenever c_{avg} condition was verified, using Equation (3.7)

Figure 3.12: Impact of using volatile Markov chains in clusters with volatile memberships.

($u_{T1}(s) \in [4, 32]$), even considering that HDFS defend against disk errors and file block corruption is reactive and lax. We make this affirmation based on Figure 3.11 showing that HDFS-T1 ensured the durability of clusters' files 1.77 more times than SGDBS-T1. However, the same figure shows that the median termination epoch for both systems is similar, but overall, values are more dispersed for HDFS-T1 than for SGDBS-T1, meaning that SGDBS-T1 is more predictable in terms of results and that, when HDFS fails, it fails harder. Conversely, SGDBS-T2 and SGDBS-T3 achieved better results than their HDFS counterparts, even in the outlier space. Ultimately both systems failed in providing durability for all of the 500 tests.

Finally, we reinforce the value of adaptively changing the topology and the goal, if at all, when cluster membership changes occur, something we did not do as expressed in Section 3.4. The result of such hindsight is visible in Figure 3.12. For the same number of file block replicas in a cluster, the time clusters spent in instantaneous convergence is approximately ten times worse than the clusters that used optimizations and whose members had 100% uptime. The distances to the goal also increased slightly because every member's departure meant restarting the Markov process from a system state with a possibly lousy initial file distribution and finally, due to the reduced time the storage node swarms had to achieve the goal, only $\approx 4.33\%$ of the clusters achieved desired equilibrium, on average.

3.7 Conclusion

In this chapter, we proposed to apply PSG algorithms used in the robotics field to ensure the durability of files hosted on a DBS. We first surveyed general DFS and P2P overlay management literature searching for ideas on how to make this application, improve it, and better understand the predicaments that we should be aware of during the development of the proposed system. We then summarized the papers by B. Açıkmeşe *et al.* [15–17], which proposed the usage PSG as an alternative to other SG methods to control groups of robots in a region space, and, a paper by Boyd *et al.* [6], whose work focused on finding fast mixing MCs by replacing heuristic methods with mathematical SDP optimizations.

We then described our problem as the creation of a P2P-based DBS using the desired PSG algorithm, focusing on file durability and availability on dynamic networks, in which any central components that existed could only perform lightweight operations. From several possibilities respecting system optimizations, we chose to give special attention to mathematical optimization problems. We believed this type of optimization could bring significant benefits to our solution, not only in terms of the time it would take for cluster swarms to converge to the desired equilibrium but also because it opens a path for additional topology constraints that can improve the overlay robustness. Those additional constraints, however, were not considered in our dissertation.

The results were not satisfactory from two perspectives. On the one hand, our classification metrics were not the most appropriate, and the number of tests we ran for our Monte Carlo simulations seemed insufficient. On the other hand, knowing there are no bugs in our implementation, at least regarding MC generation and its realization by agents, we concluded that our mathematical optimizations brought no benefits to our system. On the contrary, it degraded it. We recommend that future researchers search for different improvement methods. As an alternative to MC mixing rate optimizations, selecting better equilibrium vectors that could be either easier to achieve or decrease the probability of loss by accounting machine properties is perhaps a better starting point. Also, sticking to heuristic methods, such as MC, seems ideal since it provided better results in terms of the number of instantaneous convergences that were observed and lifetime goal achievements.

Nevertheless, we still managed to beat a prominent distributed file system, HDFS, in two relevant simulation scenarios. However, the simulated opponent was simplified and not real implementation, based mainly on public documents in their websites and scientific articles. As a consequence, we can not guarantee that one solution is better than the other. Ultimately, the only result truly in line with our expectations was bandwidth consumption being large, making our system unusable for computers with little resources, e.g., old PCs, smartphones, and other small devices not using fiber powered internet connections. We conclude that our proposal is exciting and is worth exploring further. However, it is too early to tell if it is a viable or even desirable solution for a DBS, whether it has a backup or collaborative intent.

4 Conclusions

Contents

4.1 Summarized Contribution Analysis

This dissertation addressed the development of a Python cycle-based simulator that allows testing general-purpose DFSs solutions in Chapter 2. Given the focus of NS-3 [52] (the principal contender for the programming language) on low-level issues regarding the communication network, we developed an alternative simulator. The main features in the design were modularity and usability, focusing on the application layer, which means the project bundles various useful classes that decrease the number of work researchers have to put into programming. The source code[1] is extensively documented[2] and free to use.

A main shortcomming of the current version of the simulator is that it lacks the time-efficiency seen in competitors of other languages [2, 54] due to the shortage of data structures and procedures that favor speed. Other limitations include not emulating the TCP/IP stack and reducing batch-testing flexibility by separating the configurations into multiple files. Concerning the algorithms presented in Section 3.5, we assumed that all network nodes are trustworthy, well-intentioned, and that machines do not deviate from the proposed algorithms.

A second contribution to the state-of-the-art is the proposal to view the organization of the parts of the files in the P2P-based DBS as a SG problem in robotics. We have shown that PSG is suitable for DBSs, despite our solution showing a high tax on network bandwidth. This solution proved better in simulation than HDFS [4, 23], even though it did not achieved 100% durability of the files. A better selection of the steady-state distribution vector is needed.

[1] https://github.com/FranciscoKloganB/hivessimulator
[2] https://www.hivessimulator.tech/

4.2 Future Directions

Several issues related with the addressed problems remain open. In particular, future research endeavors may include:

- Study the effectiveness of the presented solution but using EC as the goto file redundancy model. In principle, this should help achieve better durability and less bandwidth consumption, except during recovery, as mentioned in the literature (see Section 1.4.7). Ideally, the EC algorithm should be fountain-code based to allow new redundancy part injection into the network without the need to collect k of the n existing encoded file fragments.
- Given a set of available storage nodes, form a group that minimizes the probability of file loss in the cluster without following a strategy that simply picks the best from the set, as eventually, this would lead to poor load-balancing across the distributed storage system. The solution would involve computing the reliability metrics of not losing a file in a network where nodes are entering and leaving or computing the probability of losing a file in a P2P network.
- Development of a trust and evaluation system to allow the prediction storage node behavior based on past behavior, hardware specifications, and user profile. Then use these results to assist in the selection of node-weights in the desired equilibrium vector or to anticipate faults and proactively prevent them.
- Adapt the present solution such that the clusters goals and MCs are not altered whenever a membership change occurs. Inevitably, when new members replace old ones, if their profile is significantly different to the point where continuing the chain is worse than restarting it, consider the current configuration and file distribution to minimize the difference between the goals without sacrificing reliability.
- Decentralize the algorithm such that the storage nodes can collectively decide on the MCs to follow after recovering from a fault without contacting the Clients, Masters, or Monitors.

Bibliography

[1] Cloudflare. (2020) What Is The OSI Model? Accessed 14th July 2020. [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/ [ix, 16]

[2] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009, pp. 99–100. [ix, 15, 17, 61]

[3] K. Graffi, "Peerfactsim.kom: A p2p system simulator — experiences and lessons learned," in *2011 IEEE International Conference on Peer-to-Peer Computing*, 2011, pp. 154–155. [ix, 18]

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 29–43, October 2003. [Online]. Available: https://doi.org/10.1145/1165389.945450 [ix, 6, 31, 32, 61]

[5] Red Hat. (2019, February) How to configure red hat ceph storage. Accessed 1st July 2020. [Online]. Available: https://www.redhat.com/en/resources/resources-how-configure-red-hat-ceph-storage-html [ix, 6, 34]

[6] S. Boyd, P. Diaconis, and L. Xiao, "Fastest mixing markov chain on a graph," *SIAM Review*, vol. 46, March 2003. [ix, 41, 42, 43, 57]

[7] Novosoft. (2004) Handy Backup Software for Windows and Linux. Accessed 5th July 2020. [Online]. Available: https://www.handybackup.net/ [4]

[8] DropBox Inc. Work comes together in Dropbox Business. Accessed 5th July 2020. [Online]. Available: https://www.dropbox.com [4]

[9] Google LLC. Cloud Storage for Personal Use - Google Drive. Accessed 5th July 2020. [Online]. Available: https://www.google.com/drive/ [4]

[10] Microsoft Corporation. Personal Cloud Storage – Microsoft OneDrive. Accessed 5th July 2020. [Online]. Available: https://www.microsoft.com/en/microsoft-365/onedrive/online-cloud-storage [4]

[11] R. Olfati-Saber, “Flocking for multi-agent dynamic systems: algorithms and theory,” *IEEE Transactions on Automatic Control*, vol. 51, no. 3, pp. 401–420, 2006. [5]

[12] Jyh-Ming Lien, O. B. Bayazit, R. T. Sowell, S. Rodriguez, and N. M. Amato, “Shepherding behaviors,” in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA ’04. 2004*, vol. 4, 2004, pp. 4159–4164 Vol.4. [5]

[13] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006. [5]

[14] K. N. Krishnanand and D. Ghose, “Detection of multiple source locations using a glowworm metaphor with applications to collective robotics,” in *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.*, 2005, pp. 84–91. [5]

[15] B. Açikmeşe and D. S. Bayard, “A markov chain approach to probabilistic swarm guidance,” in *2012 American Control Conference (ACC)*, June 2012, pp. 6300–6307. [6, 40, 43, 57]

[16] B. Açıkmeşe and D. S. Bayard, “Probabilistic swarm guidance for collaborative autonomous agents,” in *2014 American Control Conference*, June 2014, pp. 477–482. [6, 40, 57]

[17] N. Demir and B. Açıkmeşe, “Probabilistic density control for swarm of decentralized on-off agents with safety constraints,” in *2015 American Control Conference (ACC)*, July 2015, pp. 5238–5244. [6, 40, 43, 57]

[18] J. M. Lien and E. Pratt, “Interactive planning for shepherd motion,” *AAAI Spring Symp. Agents Learn Human Teachers*, March 2009. [6]

[19] E. Masehian and M. Royan, “Cooperative control of a multi robot flocking system for simultaneous object collection and shepherding,” *Studies in Computational Intelligence*, vol. 577, pp. 97–114, January 2014. [6]

[20] L. Chaimowicz and V. Kumar, “Aerial shepherds: Coordination among uavs and swarms of robots,” *Distributed Autonomous Robotic Systems*, vol. 6, pp. 243–252, January 2007. [6]

[21] S. A. Shedied, “Optimal trajectory planning for the herding problem: A continuous time model,” *International Journal of Machine Learning and Cybernetics*, vol. 4, p. 25–30, February 2013. [6]

[22] B. Cohen, “Incentives build robustness in bittorrent,” *Workshop on Economics of Peer-to-Peer systems*, vol. 6, June 2003. [6, 7]

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, June 2010. [6, 31, 61]

[24] Red Hat. (2016) Architecture — ceph documentation. Accessed 1st July 2020. [Online]. Available: https://docs.ceph.com/docs/master/architecture/ [6, 32, 47]

[25] X. Zhang, S. Gaddam, and A. T. Chronopoulos, “Ceph distributed file system benchmarks on an openstack cloud,” *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, November 2015. [6, 33]

[26] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, “Crush: Controlled, scalable, decentralized placement of replicated data,” *SC ’06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, November 2006. [6, 33]

[27] M. Selvaganesan and M. A. Liazudeen, “An insight about glusterfs and its enforcement techniques,” in *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*, 2016, pp. 120–127. [6]

[28] L. Zhang, Y. Wu, R. Xue, T. Hsu, H. Yang, and Y. Chung, “Hybridfs — a high performance and balanced file system framework with multiple distributed file systems,” in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2017, pp. 796–805. [6]

[29] P. Maymounkov and D. Mazières, “Kademlia: A peer-to-peer information system based on the xor metric,” *Proceedings of first international workshop on peer-to-peer systems*, vol. 55, p. 53–65, April 2002. [7, 34]

[30] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, “Making gnutella-like p2p systems scalable,” *Computer Communication Review*, vol. 33, August 2003. [7, 35]

[31] A. Forestiero, C. Mastroianni, and M. Meo, “Self-chord: A bio-inspired algorithm for structured p2p systems,” *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, June 2009. [8, 35]

[32] A. Brocco, A. Malatras, and B. Hirsbrunner, “Enabling efficient information discovery in a self-structured grid,” *Future Gener. Comput. Syst.*, vol. 26, no. 6, pp. 838–846, June 2010. [8, 36]

[33] A. Malatras, “State-of-the-art survey on p2p overlay networks in pervasive computing environments,” *Journal of Network and Computer Applications*, vol. 55, pp. 1–23, September 2015. [8]

[34] N. Tölgyesi and M. Jelasity, “Adaptive peer sampling with newscast,” in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 523–534. [8, 17, 26, 50]

[35] S. Voulgaris, D. Gavidia, and M. van Steen, “Cyclon: Inexpensive membership management for unstructured p2p overlays,” *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, June 2005. [8, 17, 38]

[36] Y. Atif, “Building trust in e-commerce,” *Internet Computing, IEEE*, vol. 6, pp. 18 – 24, February 2002. [9]

[37] Li Xiong and Ling Liu, “Peertrust: supporting reputation-based trust for peer-to-peer electronic communities,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 843–857, 2004. [9]

[38] Z. Yuhong and C. Jie, “A p2p trust model based on trust factor and feedback aggregation,” in *2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE)*, 2019, pp. 214–219. [9]

[39] S. K. Awasthi and Y. Singh, “Absolutetrust: Algorithm for aggregation of trust in peer-to-peer networks,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020. [9, 17]

[40] K. Gupta and J. K. Saini, “Novel approach for distributed file system with multiple layers of fault tolerance,” in *International Conference on Computing, Communication Automation*, 2015, pp. 616–619. [9]

[41] Y. Arafa, A. Barai, M. Zheng, and A. A. Badawy, “Fault tolerance performance evaluation of large-scale distributed storage systems hdfs and ceph case study,” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7. [10]

[42] C. Blake and R. Rodrigues, “High availability, scalable storage, dynamic peer networks: Pick two,” in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1, accessed 16th July 2020. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251054.1251055 [10]

[43] K. Kim, “Lifetime-aware replication for data durability in p2p storage network,” *IEICE Transactions*, vol. 91-B, pp. 4020–4023, December 2008. [10]

[44] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris, “Efficient replica maintenance for distributed storage systems,” vol. 6, January 2006. [10]

[45] E. Sit, A. Haeberlen, F. Dabek, G. Chun, H. Weatherspoon, R. Morris, M. Kaashoek, and J. Kubiatowicz, “Proactive replication for data durability,” January 2006. [10]

[46] L. Pamies-Juarez and P. López, “Maintaining data reliability without availability in p2p storage systems,” January 2010, pp. 684–688. [10]

[47] A. Montresor and L. Abeni, “Cloudy weather for p2p, with a chance of gossip,” *2011 IEEE International Conference on Peer-to-Peer Computing*, pp. 197–217, October 2011. [10]

[48] H. Kavalionak and A. Montresor, “P2p and cloud: A marriage of convenience for replica management,” in *Self-Organizing Systems*, F. A. Kuipers and P. E. Heegaard, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 60–71. [10]

[49] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, September 2010. [10]

[50] A. Shokrollahi, “Raptor codes,” *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, June 2006. [11]

[51] T. Zhou and C. Tian, “Fast erasure coding for data storage: A comprehensive study of the acceleration techniques,” in *FAST*, 2019. [11]

[52] U. of Washington NS-3 Consortium (nsnam). (2011) About — ns-3. Accessed 14th July 2020. [Online]. Available: https://www.nsnam.org/about/ [15, 61]

[53] A. Montresor and M. Jelasity. (2016, November) PeerSim: A Peer-to-Peer Simulator. Accessed 15th July 2020. [Online]. Available: http://peersim.sourceforge.net/ [15, 16]

[54] D. Stingl, C. Gross, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz, “Peerfactsim.kom: A simulation framework for peer-to-peer systems,” in *2011 International Conference on High Performance Computing Simulation*, July 2011, pp. 577–584. [17, 61]

[55] R. Steinmetz, *Documentation for PeerfactSim.KOM*, Technische Universität Darmstadt, Technische Universität Darmstadt, Karolinenpl. 5, 64289 Darmstadt, Germany, August 2011, accessed 16th July 2020. [Online]. Available: http://peerfact.kom.e-technik.tu-darmstadt.de/fileadmin/data/Manuals/2011_08_01_PeerfactSimDocumentation.pdf [18]

[56] G. P. Jesi. (2005, December) PeerSim HOWTO: Build a new protocol for the PeerSim 1.0 simulator. Accessed 16th July 2020. [Online]. Available: http://peersim.sourceforge.net/tutorial1/tutorial1.pdf [18]

[57] NumPy. (2020, June) Numpy reference. Accessed July 2020. [Online]. Available: https://numpy.org/doc/stable/reference/ [19]

[58] SciPy. (2020, July) Scipy v1.5.1 reference guide. Accessed July 2020. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/ [19]

[59] W. McKinney. (2020, June) Pandas documentation. Accessed July 2020. [Online]. Available: https://pandas.pydata.org/docs/ [19]

[60] L. Bairavasundaram, A. Arpaci-Dusseau, R. Arpaci-Dusseau, G. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *TOS*, vol. 4, November 2008. [22, 50]

[61] Wen-Cheng Shi and Jian-Ping Li, "Research on consistency of distributed system based on paxos algorithm," in *2012 International Conference on Wavelet Active Media Technology and Information Processing (ICWAMTIP)*, 2012, pp. 257–259. [33]

[62] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, February 2003. [34]

[63] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," *Proceedings of Multimedia Computing and Networking (MMCN)*, January 2002. [34]

[64] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," *Proceedings First International Conference on Peer-to-Peer Computing*, August 2002. [35]

[65] M. Jelasity, W. Kowalczyk, and M. van Steen, *Newscast Computing*, ser. VU Technical Report. Vrije Universiteit, Faculty of Mathematics and Computer Science, 2003, no. IR-CS-006.03. [39]

[66] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Middleware 2004*, H.-A. Jacobsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 79–98. [39]

[67] P. Gagniuc, *Markov Chains: From Theory to Implementation and Experimentation*. John Wiley & Sons, 2017. [Online]. Available: https://books.google.pt/books?id=2chjtAEACAAJ [76]

A

Hives Simulator Diagrams

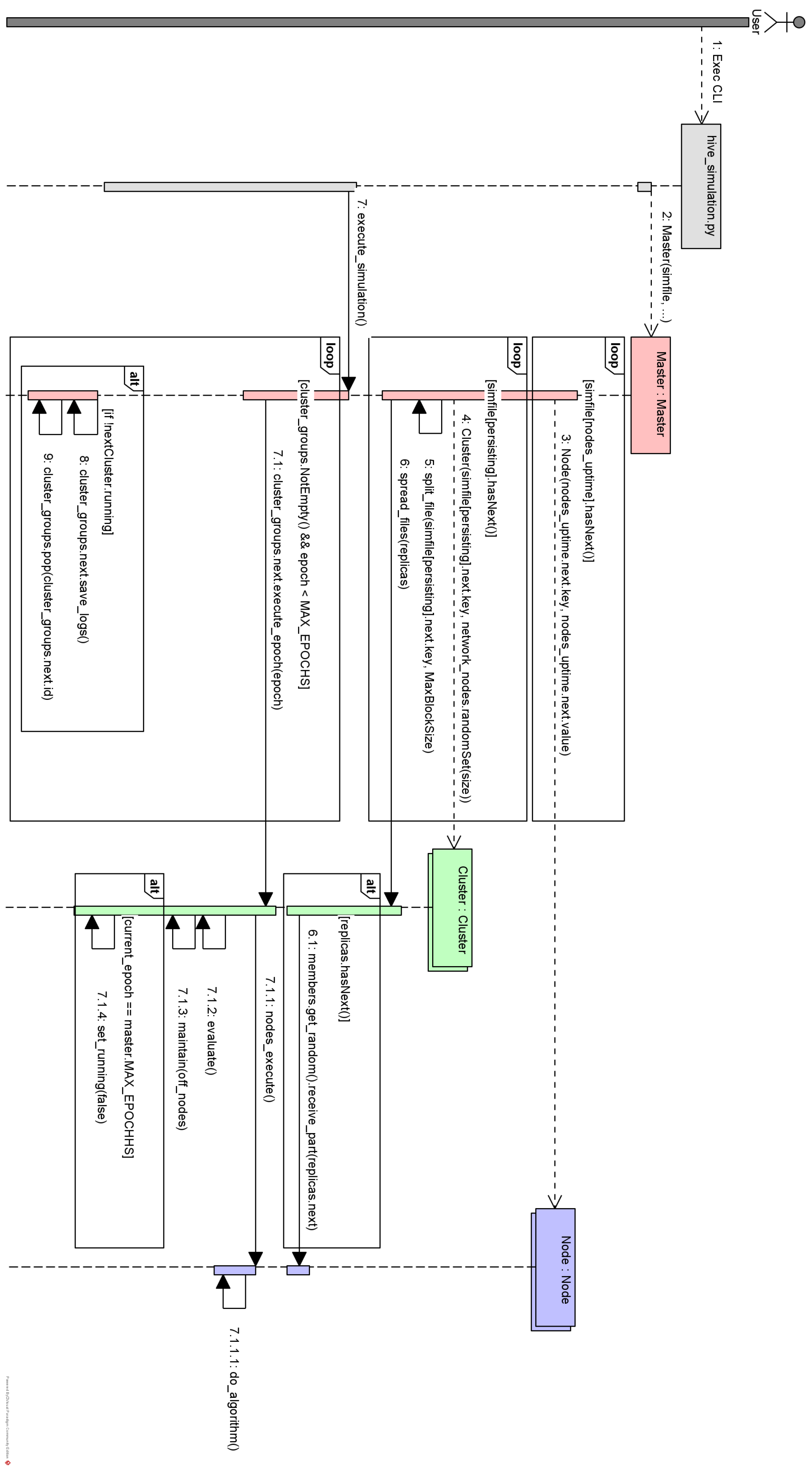

User
1: Exec CLI
hive_simulation.py
2: Master(simfile, ...)
Master : Master
loop
[simfile[nodes_uptime].hasNext()]
3: Node(nodes_uptime.next.key, nodes_uptime.next.value)
loop
[simfile[persisting].hasNext()]
4: Cluster(simfile[persisting].next.key, network_nodes.randomSet(size))
5: split_file(simfile[persisting].next.key, MaxBlockSize)
6: spread_files(replicas)
7: execute_simulation()
loop
[cluster_groups.NotEmpty() && epoch < MAX_EPOCHS]
7.1: cluster_groups.next.execute_epoch(epoch)
alt
[if !nextCluster.running]
8: cluster_groups.next.save_logs()
9: cluster_groups.pop(cluster_groups.next.id)
Cluster : Cluster
alt
[replicas.hasNext()]
6.1: members.get_random().receive_part(replicas.next)
7.1.1: nodes_execute()
7.1.2: evaluate()
7.1.3: maintain(off_nodes)
alt
[current_epoch == master.MAX_EPOCHS]
7.1.4: set_running(false)
Node : Node
7.1.1.1: do_algorithm()

Figure A.1: Sequence diagram with usual code flow for a Hives simulation.

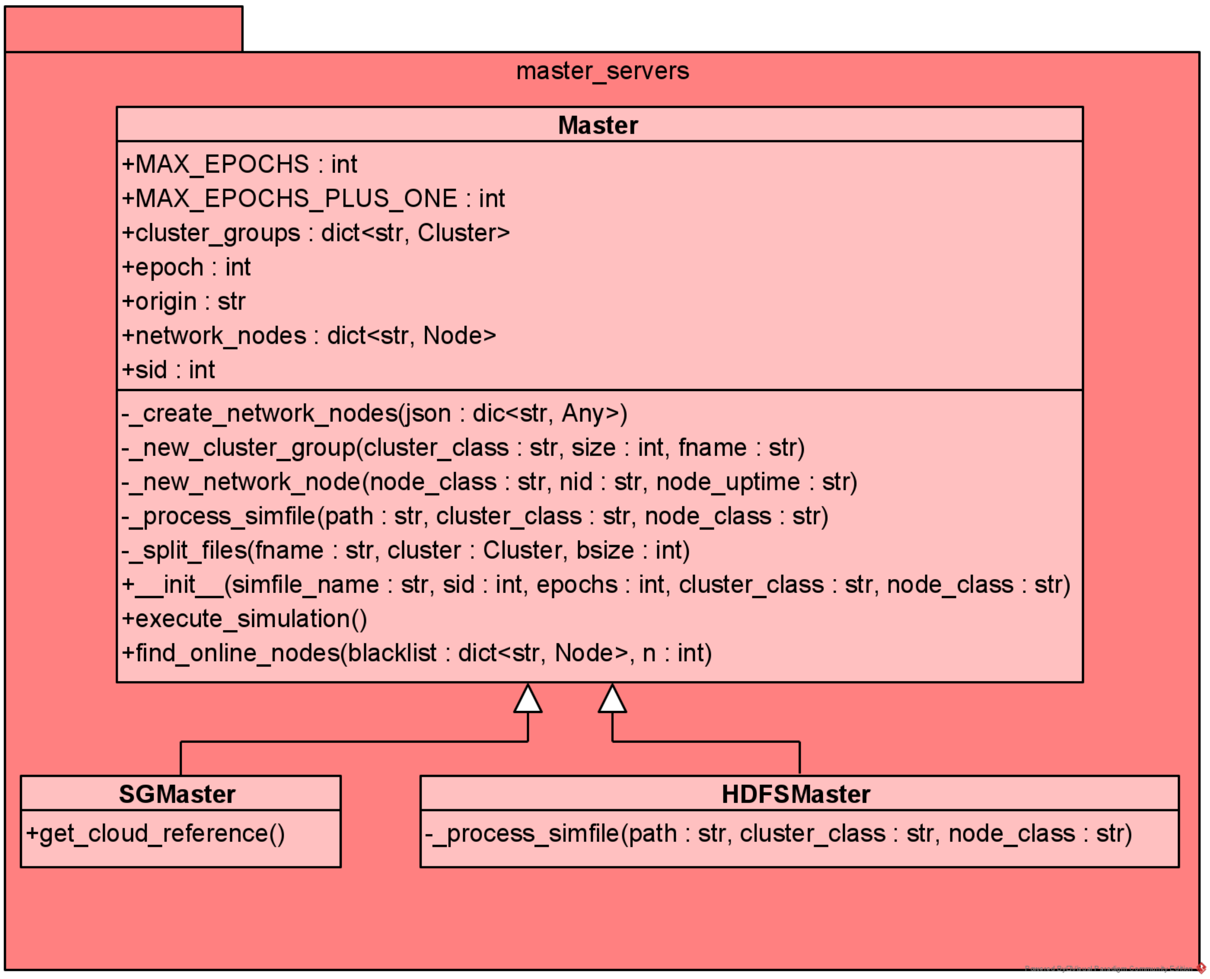

master_servers
Master
+MAX_EPOCHS : int
+MAX_EPOCHS_PLUS_ONE : int
+cluster_groups : dict<str, Cluster>
+epoch : int
+origin : str
+network_nodes : dict<str, Node>
+sid : int
-_create_network_nodes(json : dic<str, Any>)
-_new_cluster_group(cluster_class : str, size : int, fname : str)
-_new_network_node(node_class : str, nid : str, node_uptime : str)
-_process_simfile(path : str, cluster_class : str, node_class : str)
-_split_files(fname : str, cluster : Cluster, bsize : int)
+__init__(simfile_name : str, sid : int, epochs : int, cluster_class : str, node_class : str)
+execute_simulation()
+find_online_nodes(blacklist : dict<str, Node>, n : int)
SGMaster
+get_cloud_reference()
HDFSMaster
-_process_simfile(path : str, cluster_class : str, node_class : str)

Figure A.2: Hives's $master\_servers$ module class diagrams

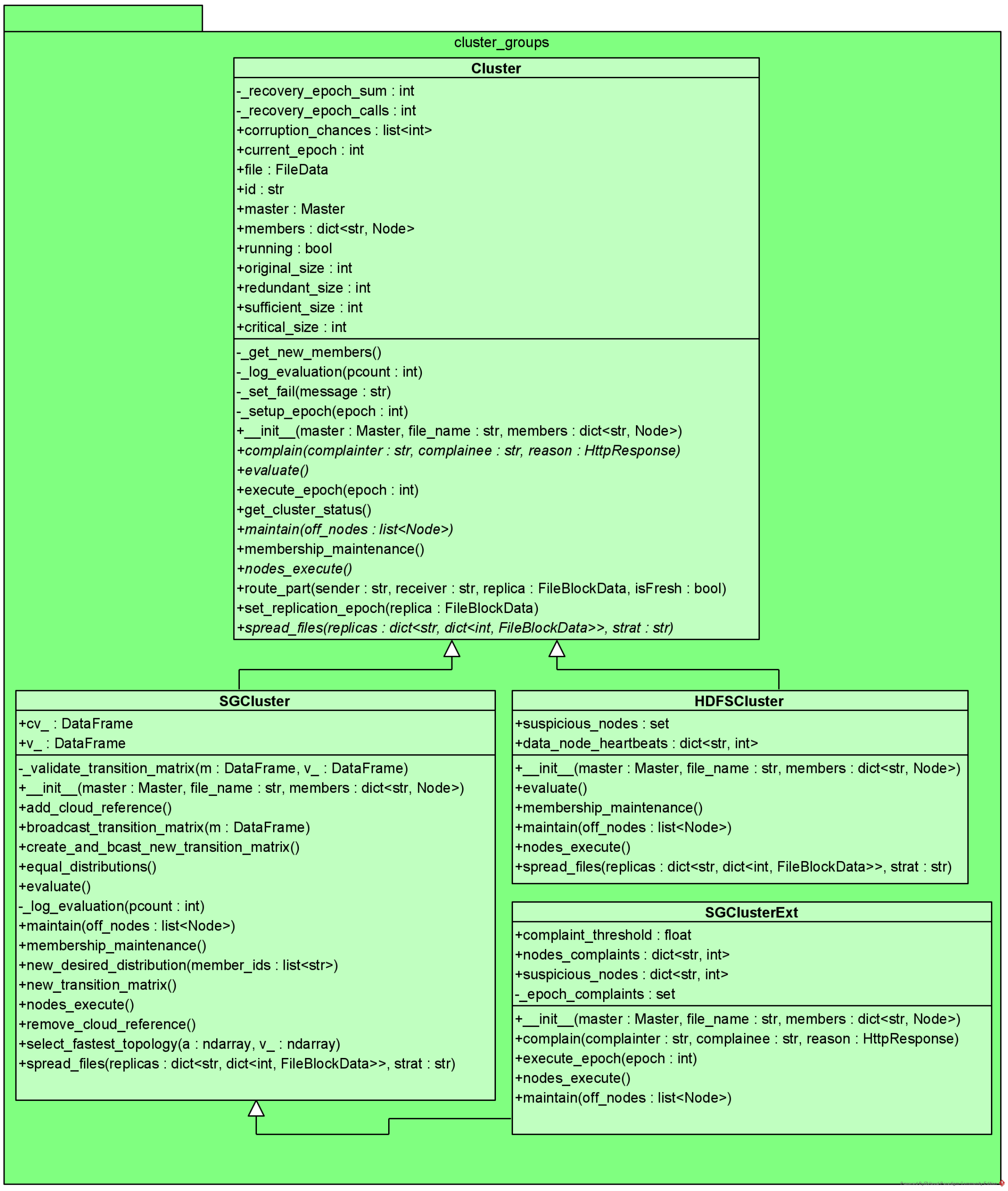
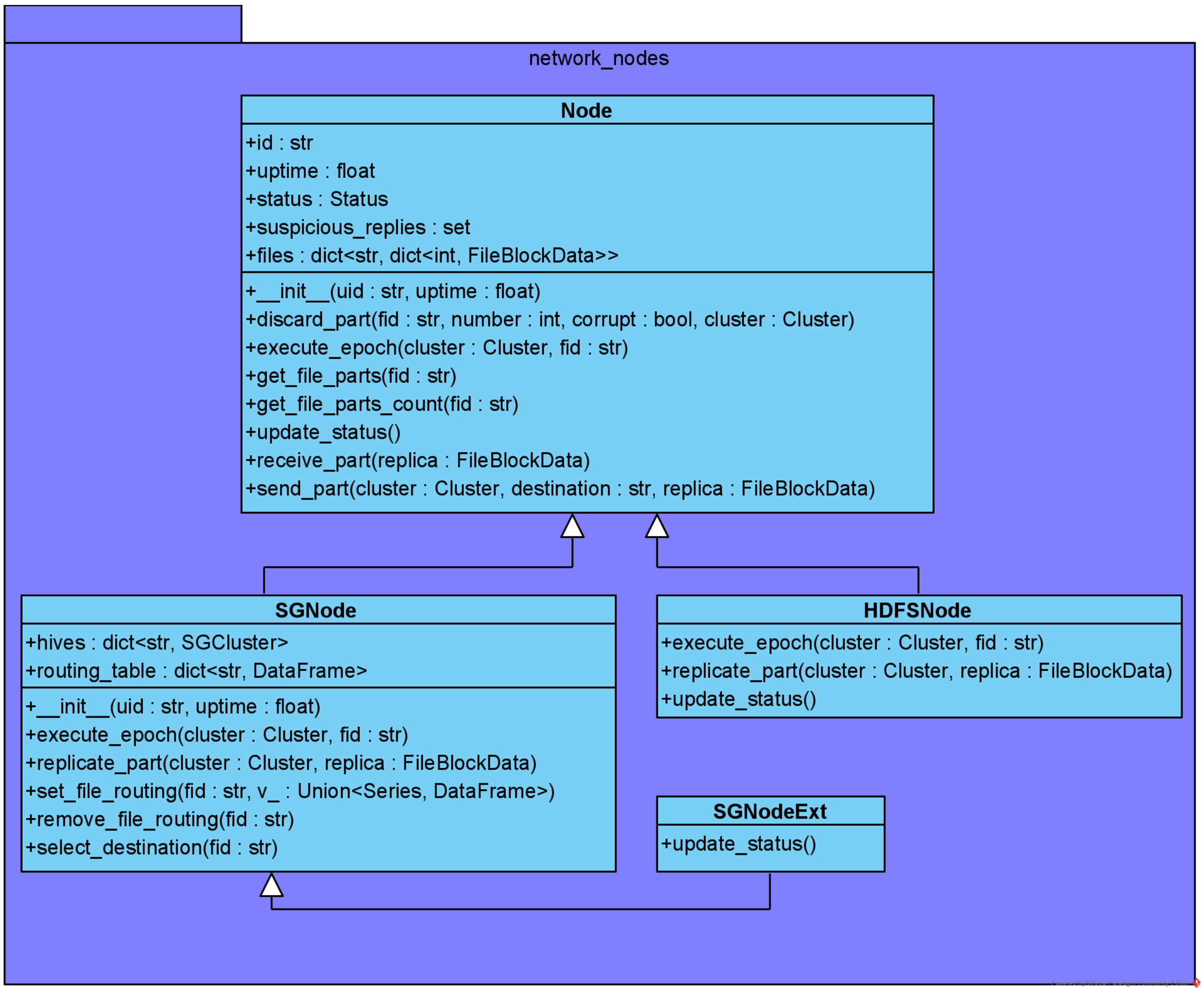

cluster_groups
Cluster
-_recovery_epoch_sum : int
-_recovery_epoch_calls : int
+corruption_chances : list<int>
+current_epoch : int
+file : FileData
+id : str
+master : Master
+members : dict<str, Node>
+running : bool
+original_size : int
+redundant_size : int
+sufficient_size : int
+critical_size : int
-_get_new_members()
-_log_evaluation(pcount : int)
-_set_fail(message : str)
-_setup_epoch(epoch : int)
+__init__(master : Master, file_name : str, members : dict<str, Node>)
+complain(complainter : str, complainee : str, reason : HttpResponse)
+evaluate()
+execute_epoch(epoch : int)
+get_cluster_status()
+maintain(off_nodes : list<Node>)
+membership_maintenance()
+nodes_execute()
+route_part(sender : str, receiver : str, replica : FileBlockData, isFresh : bool)
+set_replication_epoch(replica : FileBlockData)
+spread_files(replicas : dict<str, dict<int, FileBlockData>>, strat : str)
SGCluster
+cv_ : DataFrame
+v_ : DataFrame
-_validate_transition_matrix(m : DataFrame, v_ : DataFrame)
+__init__(master : Master, file_name : str, members : dict<str, Node>)
+add_cloud_reference()
+broadcast_transition_matrix(m : DataFrame)
+create_and_bcast_new_transition_matrix()
+equal_distributions()
+evaluate()
-_log_evaluation(pcount : int)
+maintain(off_nodes : list<Node>)
+membership_maintenance()
+new_desired_distribution(member_ids : list<str>)
+new_transition_matrix()
+nodes_execute()
+remove_cloud_reference()
+select_fastest_topology(a : ndarray, v_ : ndarray)
+spread_files(replicas : dict<str, dict<int, FileBlockData>>, strat : str)
HDFSCluster
+suspicious_nodes : set
+data_node_heartbeats : dict<str, int>
+__init__(master : Master, file_name : str, members : dict<str, Node>)
+evaluate()
+membership_maintenance()
+maintain(off_nodes : list<Node>)
+nodes_execute()
+spread_files(replicas : dict<str, dict<int, FileBlockData>>, strat : str)
SGClusterExt
+complaint_threshold : float
+nodes_complaints : dict<str, int>
+suspicious_nodes : dict<str, int>
-_epoch_complaints : set
+__init__(master : Master, file_name : str, members : dict<str, Node>)
+complain(complainter : str, complainee : str, reason : HttpResponse)
+execute_epoch(epoch : int)
+nodes_execute()
+maintain(off_nodes : list<Node>)

Figure A.3: Hives's $cluster\_group$ module class diagrams



network_nodes
Node
+id : str
+uptime : float
+status : Status
+suspicious_replies : set
+files : dict<str, dict<int, FileBlockData>>
+__init__(uid : str, uptime : float)
+discard_part(fid : str, number : int, corrupt : bool, cluster : Cluster)
+execute_epoch(cluster : Cluster, fid : str)
+get_file_parts(fid : str)
+get_file_parts_count(fid : str)
+update_status()
+receive_part(replica : FileBlockData)
+send_part(cluster : Cluster, destination : str, replica : FileBlockData)
SGNode
+hives : dict<str, SGCluster>
+routing_table : dict<str, DataFrame>
+__init__(uid : str, uptime : float)
+execute_epoch(cluster : Cluster, fid : str)
+replicate_part(cluster : Cluster, replica : FileBlockData)
+set_file_routing(fid : str, v_ : Union<Series, DataFrame>)
+remove_file_routing(fid : str)
+select_destination(fid : str)
HDFSNode
+execute_epoch(cluster : Cluster, fid : str)
+replicate_part(cluster : Cluster, replica : FileBlockData)
+update_status()
SGNodeExt
+update_status()

Figure A.4: Hives's $network\_nodes$ module class diagrams

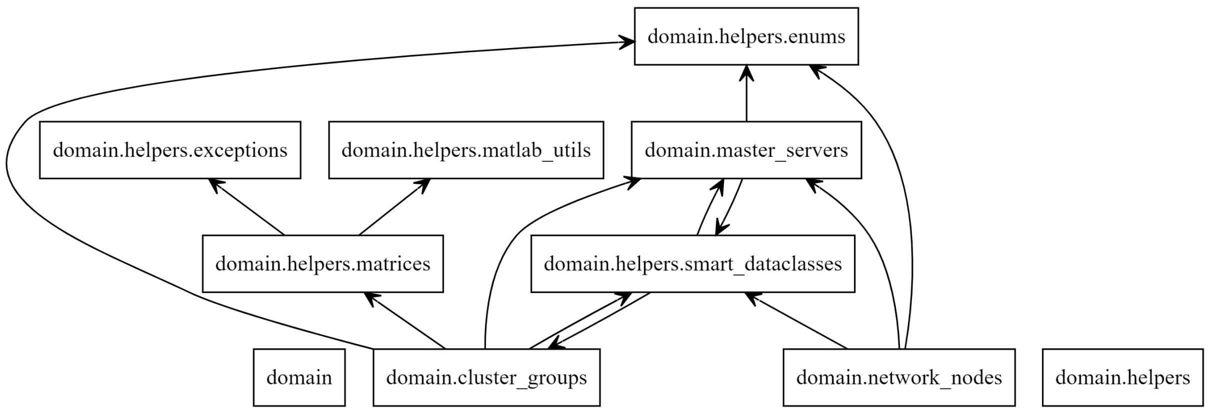


Figure A.5: Hives' packages dependencies, generated auto-generated using *pyreverse* tool.

B

Introduction to Markov Chains

To understand the concept of MCs, we first need to understand some probabilistic theory's baseline definitions. Let X be a **random variable**. Then, X is a scalar, vectorial, or another random event's outcome. For example, X could be the number facing up after rolling dice or the exposed face after flipping a coin. The space of possible outcomes of X can be either discrete or continuous. In this dissertation document, we concern ourselves only with the discrete case. Thus, the space of possible outcomes for X could be, in the case of rolling dice, the numbers one through six, assuming the dice had six faces. Knowing this, we define a stochastic or **random process** as a collection of random variables indexed by a set T, which represents different time instants. If we rolled a dice every day, starting today, for seven days than $T = \{0, 1, \ldots, 7\}$, and the random process would be the set $\{X_0, X_1, \ldots, X_7\}$, i.e., it would be the set containing the seven days' results. Note, however, that a random process may have non-finite number of time steps t. Markov processes are a special kind of random process which have the memoryless property. Suppose we have some system, and as is customary, the system has a set of possible states S. Put simply, having the memoryless property means that, at any given time $t \geqslant 0$, the outcomes of the system's future events depend only on the present system's state and never on its past states. With these definitions in mind, a MC is a discrete sequence of states, each drawn from a discrete state space that haves the following property:

$$P(X_{t+1} = s_{t+1} \mid X_t = s_t) \tag{B.1}$$

In other words, a MC describes a system whose state changes over time, and a probability distribution governs changes. Thus, X_t describes the system's state s at time t. Finally, a Markov matrix, $\mathbf{M}$, is a stochastic matrix[1] and $\mathbf{M}_{ij}$ represents the conditional probability of going to state s_j at time $t+1$, knowing that at time t we are at state s_i, as seen in Equation (B.2).

$$\mathbf{M}_{ij} = P(X_{t+1} = j \mid X_t = i) \tag{B.2}$$

Markov chains can help study many real-word processes, including lines of customers arriving at an airport, currency exchange, or city population's growth over the years, accounting for new and departed citizens. By statistically modeling these processes and given an initial distribution of the elements at study, one can predict a system's state at a certain time instant and its equilibrium in the long-run. Consider the population's growth scenario. Let c and a represent the population density living in the city and the suburbs, respectively. Let $\underline{v}_t$ be a stochastic or equivalent distribution vector or a containing the density values of c and a at time t, such that $\underline{v}_0$ is the initial system's state, i.e., how the population is distributed between city and suburbs at the start of the study. Lastly, consider t to be a yearly unit. We list a few vastly proven theorems in the literature [67], and then we use them to demonstrate the usefulness of MCs in the outlined scenario.

$$\mathbf{M} = \begin{array}{cc} c & a \\ \end{array} \begin{bmatrix} 0.7 & 0.3 \\ 0.5 & 0.5 \end{bmatrix} \begin{array}{c} c \\ a \end{array} \tag{B.3}$$

$$\underline{v}_0 = \begin{array}{cc} c & a \end{array} \begin{bmatrix} 5 \times 10^6 & 1 \times 10^6 \end{bmatrix} \Leftrightarrow \begin{array}{cc} c & a \end{array} \begin{bmatrix} 0.83 & 0.17 \end{bmatrix} \tag{B.4}$$

Fact 1. *Let $\mathbf{M}$ be the transition (Markov) matrix of a Markov chain. The $(ij)^{th}$ entry of the matrix $\mathbf{M}^t$ gives the probability that the Markov chain, starting in state s_i, will be in state s_j after t discrete time steps.*

In our scenario, if $\underline{v}_0$ is the 1st of January of the current year. Then, five years from now, the probability of observing a citizen who lives in the city moving himself to the suburbs would be given by entry $\mathbf{M}^5_{ca}$ according to Fact 1.

[1] Square matrix, where all entries are non-negative, and the sum of the entries in each column, row, or both are equal to one.

$$\mathbf{M}^5 = \begin{array}{cc} c & a \\ \left[\begin{array}{cc} 0.625 & 0.375 \\ 0.625 & 0.375 \end{array}\right] \end{array} \begin{array}{c} \\ c \\ a \end{array}$$

Fact 2. *Let M be the transition matrix of a Markov chain, and let $\underline{v}_0$ be the probability vector which represents the starting distribution. Then the probability that the chain is in state s_i after t discrete time steps is the i^{th} entry of $\underline{v}_t$ = $\underline{v}_0\mathrm{M}^t$*

By Fact 2, two years from now, 3.799×10^6 people should live in the city, and 2.201×10^6 should live in the suburbs, according to the expression $\underline{v}_2 = v_0\mathbf{M}^2$, hence, we could say there is ≈ 0.6332 probability of having a citizen living inside the city.

Fact 3. *Any irreducible*[2]*, aperiodic*[3]*, Markov chain defined on a set of states S and with a stochastic transition matrix $\mathbf{M}$ has a unique stationary (steady-state/equilibrium) distribution $\underline{e}$ with strictly positive entries. Distribution vector $\underline{v}$ is an eigenvector associated $\mathbf{M}$'s eigenvalue one. Furthermore, let $\mathbf{M}^t$ be the t^{th} power of $\mathbf{M}$, then $\lim_{t\to+\infty} \mathbf{M}^t_{ij} = \underline{e}_j \forall (i,j) \in S$.*

Simply put, Fact 3 conveys the message that in the long run, if there is a power of $\mathbf{M}$ whose entries are strictly positive[4], we will eventually reach a point where it will no longer matter how many years go by because the dynamics of the citizen's migrational behavior will not change anymore. Meaning that, in our row-major convention and for our example scenario, $\mathbf{M}$'s rows would all be equal to one another, after reaching a given year and will remain the same for all years after that, i.e., $\mathbf{M}^5 = \mathbf{M}^6 = ... = \mathbf{M}^\infty$ and $\mathbf{M}_{i\star} = \underline{e}, \ \forall i \in S$.

$$\underline{e}^\intercal \mathbf{M} = \underline{e}^\intercal \tag{B.5}$$

Suppose the conditions of Fact 3 are not satisfied. As long as we are working with a Markov matrix, we have guarantees that there is always at least one equilibrium vector $\underline{e}$, which satisfies Equation (B.5). One example that proves the existence of multiple equilibrium vectors would be the identity Markov matrix, in which case all distributions, $\underline{e}_k$, are stationary. Although all Markov matrices have a stationary vector, not all Markov chains converge to it as t approaches infinity. To prove this, we need only to think that if a Markov matrix as absorbent states[5] or transient sets[6], even when if all matrices are equal after some $t > 0$, there will always exist at least one row i in $\mathbf{M^t}$ that differs from the remaining ones. The identity matrix also fits the case where convergence is unverifiable.

[2] All states of a system can be reached from any other, directly or indirectly.
[3] There exists $t > 0$ such that all elements of $\mathbf{M}^t$ are strictly positive. A sufficient but unnecessary condition for an aperiodic MC is the existence of an s_i with $P(X_{t+1} = i \mid X_t = i) > 0$.
[4] A stochastic matrix which is strictly positive, is a regular matrix.
[5] Once a system reaches a system state s_i, it can no longer leave it. $(M_{ij} = 0 \iff i \neq j) \vee (M_{ij} = 1), \forall i,j \in S$
[6] A transient set, S', is a set of states composed of one or more $s_i \in S$ and once reached, no other state $s_j \in S$ can be revisited. $P(X_t + 1 = s_j \mid X_t = s_i) < 1, \forall s_i \in S'$.

C

Swarm Guidance Analysis Tables

Table C.1: Scenarios' configurations done with environment_settings.py and JSON simulation files.

Scenario	System	Optimized	Cluster Size	Blocks	Replication	Link Loss	Disk Error	Node Uptimes	Plays
SG8-100P	SG[1]	No	8	33		No	No	100	100
SG8-1000P	SG	No	8	333		No	No	100	100
SG8-2000P	SG	No	8	666		No	No	100	100
SG8-ML	SG	No	8	333		Yes	No	100	100
SG8	SG	No	16	333		No	No	100	100
SG16	SG	No	16	333		No	No	100	100
SG32	SG	No	32	333		No	No	100	100
SG8-Opt	SG	Yes	8	333	3	No	No	100	100
SG16-Opt	SG	Yes	16	333		No	No	100	100
SG32-Opt	SG	Yes	32	333		No	No	100	100
SGDBS-T1	SGDBS[2]	Yes	8	46		Yes	Yes	[4, 32], µ = 18, $\sigma = 8$	500
SGDBS-T2	SGDBS	Yes	8	46		Yes	Yes	[32, 64], µ = 48, $\sigma = 8$	500
SGDBS-T3	SGDBS	Yes	8	46		Yes	Yes	[64, 100], µ = 82, $\sigma = 8$	500
HDFS-T1	HDFS[3]	-	8	46		Yes	Yes	[4, 32], µ = 18, $\sigma = 8$	500
HDFS-T2	HDFS	-	8	46		Yes	Yes	[32, 64], µ = 48, $\sigma = 8$	500
HDFS-T2	HDFS	-	8	46		Yes	Yes	[64, 100], µ = 82, $\sigma = 8$	500

[1] Swarm Guidance Algorithm (perfect environment)
[2] Swarm Guided Distributed Backup System
[3] Hadoop Distributed File System