INSA Lyon | Politecnico di Torino | CITI LAB

Double Degree Programme : Master of Science in Communication Engineering

Evaluating the Power Consumption of Bartering-Based Storage Protocols in Decentralized Edge Environments

Supervisors
Prof. Frédéric LE MOUËL
Prof. Cristina ROTTONDI

Candidate
Myrian KOUMGANG TEKOBOUKITIO

Department of Telecommunications, Services and Uses - INSA LYON
Bâtiment Hedy Lamarr
Lyon, FRANCE

Academic Year 2024/25

Acknowledgements

This thesis is the culmination of a journey that would not have been possible without the support and belief of those around me.

First and foremost, I dedicate this work to my family whose love has been my anchor. Their constant encouragement and faith in me, during the most challenging moments, gave me the strength to keep going..

I'd also like to extend my heartfelt thanks to the friends I made during my academic journey. A special mention goes to my dear friend Simone, to Dimitri, and to the friends

Finally, I owe immense gratitude to my supervisors, Prof. Frédéric Le Mouël and Prof. Cristina Rottondi. Their guidance, patience, and thoughtful feedback have been invaluable. Working under their mentorship has been a privilege and a learning experience I will carry forward.

Dedication

To all those who believe in the transformative power of knowledge and hard work.

Abbreviations

P2P Peer-to-Peer

DVFS Dynamic Frequency and Voltage Scaling

IPFS InterPlanetary File System

CID Content Identifier

GC Garbage Collection

CPU Central Processing Unit

API Application Programming Interface

DHT Distributed Hash Table

HTTP HyperText Transfer Protocol

IoT Internet of Things

PoRep Proof-of-Replication

PoSt Proof-of-Spacetime

TCP Transmission Control Protocol

FIO Flexible I/O tester

Nomenclature

E_{total} Total energy consumption

E_{network} Network energy consumption

$E_{k,i}$ Average energy consumption of node i under profile k

P_{total} Total power consumption

P_{static} Static (idle) power consumption

P_{dynamic} Dynamic (active) power consumption

P_{idle} Idle power consumption of network link

f CPU frequency

u CPU utilization

T_{uptime} Uptime duration

K Replication factor (number of copies stored)

N Total number of nodes in the network

α_k Fraction of time spent idle by profile k

β_k Fraction of time spent active by profile k

Δ_{valid} Score increment for valid storage proof

Δ_{invalid} Score penalty for invalid storage proof

Δ_{timeout} Score penalty for storage proof timeout

k Bartering ratio (bytes stored for peer / bytes peer stores)

α Peer selection parameter (fraction of high-score peers)

Contents

List of Figures

Abstract

Assessing and reducing the energy consumption of distributed storage systems in edge computing environments is essential to curtail both operational costs and the broader environmental impact of emerging digital infrastructures. As edge computing continues to gain traction offering low-latency data services closer to end users peer-to-peer (P2P) file systems play a pivotal role by decentralizing data management. However, these P2P storage solutions, while ensuring fault-tolerance and scalability, can significantly influence the overall energy footprint. Traditional simulations have often been employed to understand system behavior and project potential efficiencies in managing energy resources. Yet, such simulations frequently lack the capability to precisely model power usage, particularly in multi-core settings where resource contention, hardware heterogeneity, and decentralized decision-making patterns complicate energy dynamics. These limitations are further compounded in real-world edge scenarios, where node churn, inconsistent connectivity, and diverse workloads make it challenging to capture realistic performance and energy consumption profiles through simulation alone.

In contrast, this study takes a more direct, and empirically grounded. It achieves this end through introduction of a new protocol for storage, using bartering. Through a bartering mechanism, respective nodes offer and request storage reciprocally, regulated by a score metric that lets peers decide whether to accept or reject respective incoming requests. The protocol incorporates a proof-of-storage process, in which nodes must routinely show that they truly store respective data; dishonest conduct results in score penalties.For emulation of real cloud conditions, we introduce, a churn model, and we also use the flexible I/O (FIO) benchmark for generation of file-system workloads that are representative. Experiments over Grid'5000 (French national testbed cluster) are presented, offering insights on both small-scale to large-scale performance and energy consumption.

The results from the replication strategy analysis suggest that a replication factor with K=3 offers nearly an optimal balance in redundancy with energy consumption. Also, the system maintains over 90% data availability up to 30% churn rates. Ultimately, our approach and findings establish a strong framework toward the future of P2P communication, deployment, and optimization of decentralized storage systems throughout the edge,informing researchers about the interplay between operational policies and their energy consumption implications.

1 Introduction

Distributed storage systems in data centers currently in use consume several megawatts of electrical power per hour, making it crucial to reduce their power consumption by finding an optimal balance between performance and power consumption of applications run on these systems. Multi-core processors account for a significant fraction of the energy used by these machines, and hence, leveraging power-saving techniques such as Dynamic Frequency and Voltage Scaling (DVFS) [5] plays a vital role.

When attempting to identify trade-offs between performance and power/energy consumption on large-scale machines through experimentation, one faces a vast array of configuration possibilities, e.g., number of nodes, number of cores per node, or DVFS levels. Consequently, a thorough experimental evaluation of the power usage of applications requires a substantial amount of resources and compute time. Direct experimentation can help to answer questions about power efficiency and has already been used to analyze applications in a variety of scenarios. However, unlike simulators that often lack the capability to predict power, direct experimental approaches enable the study of the effects of different DVFS policies and can contribute significantly to the understanding and the design of energy-saving policies[24].

In this thesis, we assess the power consumption of a bartering based protocol through direct experimentation. In particular, we quantify the consumption the bartering binary application by deploying it on the Grid'5000 platform. Power usage in this context is modeled as the power consumed by multi-core processors throughout the entire execution of the application, making an accurate estimation of the execution time of the application a fundamental requirement for a reliable quantification of their power usage. Accurate modelling of performance and power/energy consumption of the bartering application could be highly beneficial for capacity planning, from both platform designers and users' perspectives, allowing them to adequately dimension their infrastructure.

The remainder of this thesis is structured as follows. Chapter 2 reviews related work on energy consumption in edge computing applications, peer-to-peer systems, and churn management strategies and provides some background notions. Chapter 3 presents our novel bartering-based protocol, including its algorithmic design and comprehensive operational mechanisms. Chapter 4 introduces the power consumption model used to evaluate energy efficiency in distributed storage nodes. Chapter 5 details the implementation of our protocol, including system architecture, code components, and energy optimization techniques. Chapter 6 describes the experimental methodology, including the Grid'5000 infrastructure, deployment procedures, and synthetic workload generation using the Flexible I/O (FIO) benchmarking tool. Chapter 7 presents a comprehensive analysis of our experimental results, examining performance across different node profiles, replication factors, and testing intervals. Finally, Chapter 8 concludes with a summary of our findings and directions for future research.

2 Related work and Background

2.1 State of the Art

The exploration of energy consumption and efficiency in distributed systems, particularly in edge computing and fog environments, has seen significant advancements through various research efforts and technological innovations. The literature is rich with studies that dissect the complexities of energy management in decentralized infrastructures, focusing on both theoretical and practical aspects.

Filecoin and IPFSCluster represent two pivotal projects in the realm of distributed file storage, leveraging blockchain and peer-based technologies to ensure data redundancy and security. Filecoin [25] utilizes a cryptocurrency incentive, secured through proof of storage, to ensure data availability and integrity. Conversely, IPFSCluster [29] operates without blockchain technology, relying on redundancy through data replication across nodes, aiming for a more straightforward data integrity solution without the computational overhead of blockchain operations.

Our approach significantly deviates from these models by eliminating cryptocurrency reliance, thereby reducing the energy consumed by blockchain computations. Moreover, our protocol introduces a more flexible replication strategy, enhancing control over data distribution and reducing unnecessary replication, thereby aiming to lower overall energy consumption.

In the context of energy-efficient protocols, authors in [22] introduce an innovative model for optimizing infrastructure energy consumption in smart metering systems. Their work underscores the challenges and solutions in managing energy costs in large-scale deployments, which is critical for our understanding of energy dynamics in distributed networks.

Additionally, advanced metering and optimization are discussed by authors in [24], who focus on the predictive modeling of energy consumption for MPI applications. They emphasize the potential of simulation tools to enhance energy efficiency in high-performance computing environment. This approach provides significant insights for designing energy-efficient distributed systems, emphasizing a cost-effective method for simulating large-scale environments and optimizing energy consumption. In [49] the authors delve into the energy consumption patterns within microservices architectures. They compile and analyze various studies to present a comprehensive overview of how energy efficiency can be achieved or compromised within such architectures. The review highlights that while microservices enhance scalability and flexibility, they can also lead to increased energy consumption due to the overhead of managing numerous distributed services. This insight is particularly relevant to our study as it underlines the importance of efficient resource management in reducing energy consumption in distributed systems.

Further, the shift toward edge computing paradigms is elaborated upon by authors in [50], who analyze the energy and performance metrics of fog computing applications. Their findings highlight the trade-offs between proximity computing and energy efficiency, providing a foundational understanding that enriches our approach to decentralized storage.

Additionally, research on smart grids, particularly the study by [38], emphasizes the intricate balance required between operational efficiency and energy consumption in national infrastructure projects. This body of work provides valuable insights into the scalability of energy solutions in real-world scenarios, which is instrumental in framing our energy reduction strategies in distributed storage systems.We'll have a more indepth discussion of

the model used to measure the power consumption of edge nodes in chapter 6.

Our research builds upon these foundational studies, aiming to refine and innovate upon the existing methodologies to create a more energy-efficient distributed storage system,finding a trade-off between energy consumption and performance.

2.2 Blockchain Technology

As the name suggests, blockchain is made up of record blocks that are linked together with cryptography. Each block contains the hash of the previous block, a timestamp, and transaction data, so the chain forms one public, digital ledger of transactions. Among the major security advantages of blockchain technology are the integrity of transactions, strong authentication, and the immutability of data. Because it runs on a decentralised, peer-to-peer infrastructure, there is no single point of failure, and middlemen together with their additional fees are removed.

A blockchain is a distributed database replicated on every node in the network. The size of a block and thus the cost of generating it varies across blockchains, but in every case the node that successfully mines a block is rewarded with the cryptocurrency specific to that chain (Bitcoins on the Bitcoin network). All nodes store the same public ledger, and a fully synchronised node can transfer assets to another peer; the transfer is broadcast, verified and recorded immutably on the chain. Users are identified by digital signatures only; hashes and Merkle-tree structures link the blocks, so real-world identities remain hidden.

Blockchain offers append-only storage: once data are on-chain they can never be changed or removed. That immutability comes at a price that differs sharply from conventional storage in both size and cost. For example, the Bitcoin blockchain exposes the `OP_RETURN` opcode to embed arbitrary data. The payload was originally limited to 80 bytes and was reduced to 40 bytes in February 2014 [9]. Storing 80 bytes on Bitcoin costs roughly US$0.03617, whereas the same amount on Ethereum costs about US$0.007 [21].

Because on-chain space is scarce, designers must decide carefully which data belong on-chain and which should remain off-chain hence they turn on IPFS being a popular off-chain and blockchain-friendly storage systems which relies on peer-to-peer file-system principles.In [35], authors present their approach to decentralized storage using blockchain. They propose a network of peers who make their storage capabilities available to users. When a user wants to store files on the system, he will need an Ethereum wallet. The file he wants to store is first encrypted with his wallet key, and sent onto a smart contract. This contract will require the user to pay a certain amount of funds. Once this is validated, the file will be split into blocks, and distributed among peers in the network, to make several copies. These files are encrypted, and can only be retrieved with the user's wallet key, and funds paid by the user are split among peers who store the file. The file can be accessed by the user whenever needed, and peers who fail to provide the file will be blacklisted. This approach ensures that peers are compensated for storing files that aren't theirs. Using a cryptocurrency to reward users that store data and are reliable is relevant to our study. However, the time required to upload a file is relatively high, and so will be the time to retrieve it.

In [40], authors present an approach at dynamic data replication in a peer to peer network. Each data block's popularity is calculated, as popular data blocks should be replicated to make sure they are easily accessible by nodes in the network, and less popular data blocks should exist in fewer copies to free storage space on the system for new data. The popularity metric is computed based off the number of requests to access the data block. A minimum number of data replicas is calculated, taking into account an availability threshold, the probability

of failure of nodes and offline hours of nodes. Whenever a data block's popularity falls behind, the number of replicas will go back down to the minimum number of replicas. Then, authors present an approach to determine the right nodes to store data on. For this, they give each node a score, based off its features, efficiency in the network, and position in the network.

2.3 Peer to peer Systems

P2P systems are network architectures where all nodes are interconnected, and can exchange information and files when needed; there is no client and server role. In this subsection, we'll first discuss similarities among peer systems and edge computing . Then, we'll take a look at how peer to peer systems allow for data replication and efficient retrievals. We'll also introduce other useful advances in the field of peer to peer systems, especially energy trading.

2.3.1 Similarities with edge computing

Several papers examine the similarities between peer to peer systems and edge computing. While they aim to achieve different goals overall, it does seem like a lot problems that edge computing brings on the table also are issues found in a peer to peer system, so taking advantage of the decades [27] of research on these systems is important. In [30], authors examine these similarities. Peer to peer nodes are usually constrained in resources, and can frequently go on and offline. Efficient data storage and retrieval are also problems tackled in peer to peer research. Nodes are also owned and managed by different agents in both cases. Peer to peer nodes are subject to security threats, such as distributed denial of service attacks, just like edge computing nodes. It therefore makes sense to study peer to peer systems, as they have provided answers to all these problems. While these solutions might not directly apply to an edge and fog layer, taking inspiration from them could be beneficial. In [27], authors present different possible use cases of peer to peer interactions between edge computing servers, as well as possible research directions. For example, authors highlight a scenario where multiple users using the same services could have similar computation requirements. In this case, peer to peer interactions are a solution to share the computation results, resulting in less energy consumption and lower response time. Another scenario, in certain ways similar to the architecture we are considering, is one where users are served in priority by their closest edge server, that forwards the request if needed to its other closest servers, ensuring the user is served by a relatively physically close server. Authors also point out routing and security as emerging fields that have not been successfully solved in peer to peer nor edge computing literature. On another hand, they claim edge computing could take advantage of the proximity awareness systems proposed in peer to peer architectures to get in contact with their neighbors.

2.3.2 Peer to peer file system and Storage

One of the most well known peer to peer file system is the IPFS, Inter Planetary File System[20]. IPFS brings together a set of already proven, working technologies to build an efficient distributed file system. Nodes have a public and a private key, and an ID generated by hashing the public key. These keys allow for nodes to check for messages' authenticity as they communicate with other nodes in the network. For routing, IPFS uses a distributed hash table to allow efficient retrieval of data based on keys, and efficient distribution of data. Similarly

to BitTorrent, IPFS stores data by splitting it into blocks and distributing it on the network.

Nodes are incentivized to store data blocks needed by their peers and free riders are blacklisted thanks to a debt ratio that acts as a reputation metric. If a node has successfully shared and downloaded data before, it will be known as a trusted node. This is also used as a way to prevent or at least discourage nodes to disconnect from the network, as this would mean resetting their trust metric and losing network privileges. IPFS offers a lot of possibilities regarding file versioning and content addressing. While we won't get into all features here, what we should keep in mind regarding IPFS is that it is highly customizable and designed for applications to be built on top of.

2.3.3 Churn management in peer to peer systems

Peer to peer systems rely on having a fair number of nodes active on the network. While nodes can join the network as they wish, they can also leave it (whether it is intentionally or because of a failure) and join it again. This phenomenon is referred to as churn, and peer to peer systems have to efficiently deal with it.

In [12], one example of an approach against churn is presented. Authors notice previous literature mostly covers faults in a static manner: a random but bounded number of nodes are crashed, then the system is given time to converge back to a stable state. They consider this is not realistic, and they design a peer to peer network and an adversary agent, whose role is to randomly crash and add nodes in the network. The network is basically constantly failing, and is constantly converging towards a stable state, making it resilient to failures. For that, they distribute peers evenly on a d-dimensional hypercube. Each vertex of this hypercube represent a group of peers in the network. Each data block that needs to be stored on the network is assigned to one of the hypercube vertices, and it is ensured at least one of the peers in the vertex holds the data. The hypercube's dimensions varies as the number of nodes change in the network. Periodically, peers evaluate the number of nodes in the network by sending messages to each other (containing their ID and the ID of its adjacent peers) and the hypercube's layout and dimension are updated. However, it is important to highlight that this system is only effective for fail-stop failures, so under the hypothesis that there are no byzantines. Also, authors consider the time needed for peers to communicate can be bounded.

2.4 IPFS : A Decentralized Peer-to-Peer System for Cloud Storage

In the context of decentralized file storage, the focus has been on exploring innovative strategies to ensure secure, scalable, and reliable data storage while addressing challenges such as redundancy, fault tolerance, and energy efficiency. Solutions like IPFS and Filecoin have been pivotal in this domain, each offering unique approaches to decentralized storage [44]. Typically, IPFS is taken as the foundational model for content-addressed storage, while Filecoin introduces an incentive-based mechanism to encourage resource sharing. These systems are then compared to assess their effectiveness in achieving scalability and sustainability.

Figure 1: IPFS [45]

The InterPlanetary File System (IPFS) laid the groundwork for decentralized cloud storage by introducing a system that integrates peer-to-peer technology, similar to BitTorrent swarms, with cryptographic principles. This approach allows users to share files across a distributed network without the need for centralized data centers or intermediaries.

As highlighted in the IPFS whitepaper: "IPFS is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files. In some ways, IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high-throughput content-addressed block storage model, with content-addressed hyperlinks."

The concept of content-addressing mentioned here is particularly critical. Understanding content-addressing as the cornerstone of decentralized storage systems is essential to fully grasp the architecture and functionality of IPFS.

2.4.1 IPFS Network

In IPFS, effective node-to-node communication is at the heart of enabling a decentralized system. Two distinct P2P approaches underlies this communication:

- **Unstructured P2P:** In an unstructured network, there is no strict or deterministic plan for how peers interconnect. Instead, connections form randomly or probabilistically according to a proximity metric. This style of organization, which IPFS relies on [3], is well suited for rapid data distribution and content-driven retrieval. Unstructured P2P setups tend to be robust when nodes leave or join unexpectedly, but the only way to find a particular resource is to potentially query all nodes, offering no absolute guarantee that the resource will be located [1].

- **Structured P2P:** In a structured system, peers occupy a “pseudo”-metric space where each node is assigned an identifier drawn from a large set of possible identifiers. Neighborhood relationships are then dictated by the distance function in that space [7]. Additionally, objects in the network receive identifiers (often derived from hashing) in the same identifier space [11]. Consequently, objects can be mapped directly onto nodes, which shortens the time required for lookups [8].

  A structured P2P network has two defining characteristics:

  - A “pseudo”-metric space for node identifiers.

- A distance function to determine how peers are connected and where data is stored.

This section focuses on two networking layers utilized by IPFS: Bitswap (an unstructured protocol) and a DHT (distributed hash table). Both function cooperatively within IPFS.

2.4.2 Bitswap

Bitswap, the unstructured P2P component in IPFS, was influenced by the BitTorrent protocol, a decentralized system for file sharing [19]. A central concept in Bitswap is its message-based structure, where messages describe which blocks a node owns or needs, and sometimes contain the actual block data. Nodes can thus retrieve parts of a hash-linked Merkle DAG from different peers.

Each node keeps:

- A `want_list`, listing blocks it currently needs.
- A `have_list`, indicating blocks it can supply.
- A set of direct neighbors (peers).

Exchanges only occur between a node and its neighbors, limiting the search scope. Consequently, it may fail to locate a requested block if none of the neighbors possess it [19].

Bitswap Messages As noted, Bitswap is constrained to a node's direct peers. The first peers a node encounters (so-called bootstrap peers) are predefined in its configuration and typically provided by the IPFS team. This bootstrapping step introduces a security dimension: compromised bootstrap nodes can funnel new participants toward malicious peers.

Once a node has multiple connections, it can undertake IPFS actions, such as requesting files. When a node wants to retrieve a block, it can initiate the request in several ways:

- Sending a want-list, sending a want-have, and sending a want-block.

A want-list is simply a set of CIDs (Content Identifiers) the node or its peers are seeking. If the node receives a block that matches a peer's CID request, it forwards that block onward. Because requests are monitored at the peers, a node must issue a cancel message once it obtains the target CID, as illustrated in Figure 2, to avoid continued, redundant transmissions.

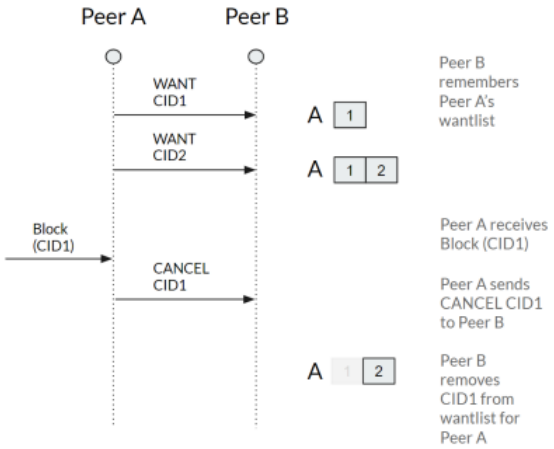


Figure 2: Usage of the cancel message in the Bitswap protocol

Each item in the `want_list` has a priority and a want-type:

- `Block` (0): Ask for the block immediately. This is ideal for smaller files.
- `Have` (1): Inquire if the peer has the block before downloading it—useful for larger data to conserve bandwidth.

The 32-bit priority determines the order in which requests are processed; higher priorities are addressed first. Among identical CIDs, want-have takes precedence over want-block if both are submitted at the same time.

Nodes share their `want_list` with peers under several circumstances:

- When a new neighbor is added.
- After a random delay.
- Whenever the `want_list` changes.
- After receiving new blocks.

Figure 3 provides an example where Peer A sends Peers B, C, and D a want-list with type `"Have"` to see who has a given CID. Peer B confirms possession of the data, prompting Peer A to issue a `"Block"` request for that CID from Peer B. Other peers might also respond, but Peer A will wait for the actual data block from Peer B.

Upon receiving a want-list, a node stores it locally, checks which blocks it can fulfill, and sends any available data. After sending a block to a requester, it becomes another provider for that block [19].

While node typically use a want-list, sending single `want-have` or `want-block` messages can be more efficient in certain situations. For instance:

- `want-have`: Determines if a peer holds a CID without committing to a full download.
- `want-block`: Requests the block directly, even if it is uncertain whether the peer has it.

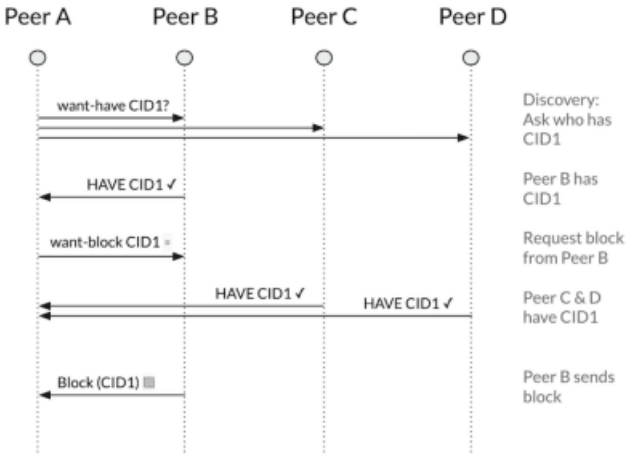


Figure 3: Want-have representation from IPFS documentation

From a responder's perspective, upon receiving a want request:

- If the peer has the requested CID, it replies with a `HAVE`. The requester then sends a `want-block` message to retrieve the block.
- If the peer does not have the CID, it may:
  - Send a `DON'T HAVE` message.
  - Remain silent (no response).

Note that some peers do not understand `Have` messages and will only supply the actual block. Bitswap accounts for compatibility during its message queue process before sending updates or blocks.

Bitswap Information. Retrieving a block in IPFS through Bitswap is complicated by two major factors:

1. Each node caches blocks obtained from its neighbors, thereby enlarging the set of potential providers.
2. The initial request typically targets the root block of a Merkle DAG. Dependent blocks are requested subsequently, once the parent block is known.

Given that each node maintains details on what it needs, provides, or learns from peers, Bitswap follows a probabilistic path to query neighbors with a higher likelihood of supplying the requested CID. If a neighbor is also looking for the same data, that neighbor is queried first to save time. When none of a node's neighbors have the needed block, the node switches to the DHT to locate an appropriate provider.

Finally, if a neighbor repeatedly fails to deliver any requested blocks or never contributes data, a node may discontinue that connection. For example, a peer that only downloads data and never uploads it will lose its reputation or score over time.

Content Addressing Before the introduction of IPFS, most online content sharing relied on the HTTP protocol. This is why users typically start accessing a website or content by typing "http://" followed by the address [28].

However, HTTP is a location-based protocol, meaning it retrieves content by locating the specific server where the data is stored based on its geographical or network address. While this system has functioned adequately over the years, it has inherent limitations, and more efficient and secure alternatives are possible.

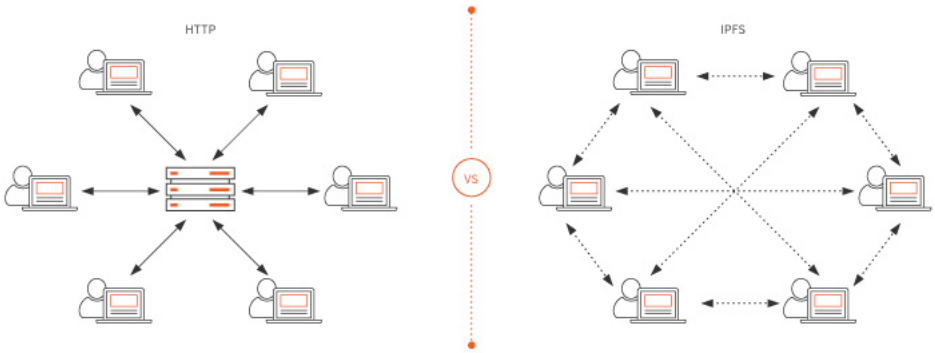


Figure 4: HTTP vs IPFS content retrieval [36]

With content addressing, as utilized by IPFS, files are identified not by their location but by a *Content Identifier (CID)*, which is essentially their cryptographic hash value rather than a traditional filename. This approach enables efficient storage and retrieval of data, as users can access files without relying on a centralized physical storage location [31].

The use of content addressing offers significant advantages in terms of security, reliability, and scalability. By referencing files solely through cryptographic hashes, it eliminates a central point of failure, making it harder for attackers to target specific files. Additionally, this method minimizes duplication, reducing inconsistencies and potential security vulnerabilities in the dataset. As a result, content addressing underpins a more robust and decentralized storage model.But IPFS nodes thus need a way to know where the content that a CID references is located. For this IPFS uses **distributed hash tables (DHTs)** and the **kademlia algorithm** to map the addresses of peer nodes that hold specific content. When a user asks an IPFS node to retrieve the content of a CID, the node will query the DHT to retrieve addresses of all peers that have that particular CID.

2.4.3 Kademlia Algorithm

The Kademlia algorithm[1] has been around for a while, and its primary purpose is to build a DHT [42] using three main system parameters:

1. **Address space.** A unique identifier space (integers from 0 to $2^{256} - 1$ in IPFS).
2. **Metric.** A way to order peers (interpreting the SHA-256 hash of the PeerID as an integer), so that all peers can be laid out from smallest to largest.

[1] `https://docs.ipfs.tech/concepts/dht/#kademlia`

3. **Projection.** A function that takes a record key and computes a position in the address space; peers near this position (by the chosen metric) are "closest" and thus ideal for storing or retrieving that data.

Having this address space, metric, and projection allows the network to behave like a sorted structure, often compared to a skip-list. A node maintains knowledge of peers at specific exponential distances (1, 2, 4, 8, etc) .so that a lookup can be done in $O(\log N)$ time, where N is the size of the network.

Peers in Kademlia join, leave, and rejoin the network dynamically. To handle the high churn, each node keeps up to K (in IPFS, $K = 20$) connections at each of those exponential distance "buckets." This redundancy helps maintain connectivity even if some subset of peers goes offline. The choices of $K = 20$ is determined from empirical observations of churn and desired latency, ensuring the network remains connected and data is not lost.

Kademlia Key Operations. Kademlia's main components[48] are its routing table (which manages all the links/buckets) and its lookup algorithm (for storing and retrieving data). Simplified, the core operations are:

- STORE(key, value): Store *value* in peers close to *key*.
- FINDNODE(key): Locate peers whose PeerID is closest to *key*.
- FINDVALUE(key): Retrieve the value associated with *key* from nearby peers.

Algorithm 1 Kademlia Lookup (simplified pseudo code)

Require: *key*: The content key or node ID to be located
Require: k: The bucket size (e.g., 20 in IPFS)
function KADEMLIALOOKUP(*key*)
  shortlist ← CLOSESTNODES(*key*, k) ▷ Get k nodes from local routing table near *key*
  queried ← ∅ ▷ Track nodes we've already queried
  repeat
    unqueried ← shortlist \ queried
    toQuery ← SELECTUPTO(unqueried, α) ▷ Up to α queries in parallel (e.g., $\alpha = 3$)
    for all $n \in$ toQuery **do**
      queried ← queried $\cup \{n\}$
      response ← RPCFINDNODE(n, *key*) ▷ Network query for closer peers
      if response.HASVALUE **then return** response.GETVALUE ▷ Value found
      end if
      tmpPeers ← response.GETCLOSERPEERS
      MERGESHORTLIST(shortlist, *tmpPeers*)
      ▷ Add newly discovered closer peers
    end for
    SORT(shortlist, distanceToKey)
    shortlist ← shortlist$[1..k]$ ▷ Keep only the k closest
  until NOCHANGEINSHORTLIST
  return shortlist
end function

In the pseudo code above, the node begins by gathering a shortlist of k closest peers from its local routing table. It queries a small batch of them (up to α in parallel) for even closer peers, updating the shortlist until no

better (closer) peers are found. If one of the queried peers actually holds the value (in a FIND VALUE operation), the lookup terminates successfully.

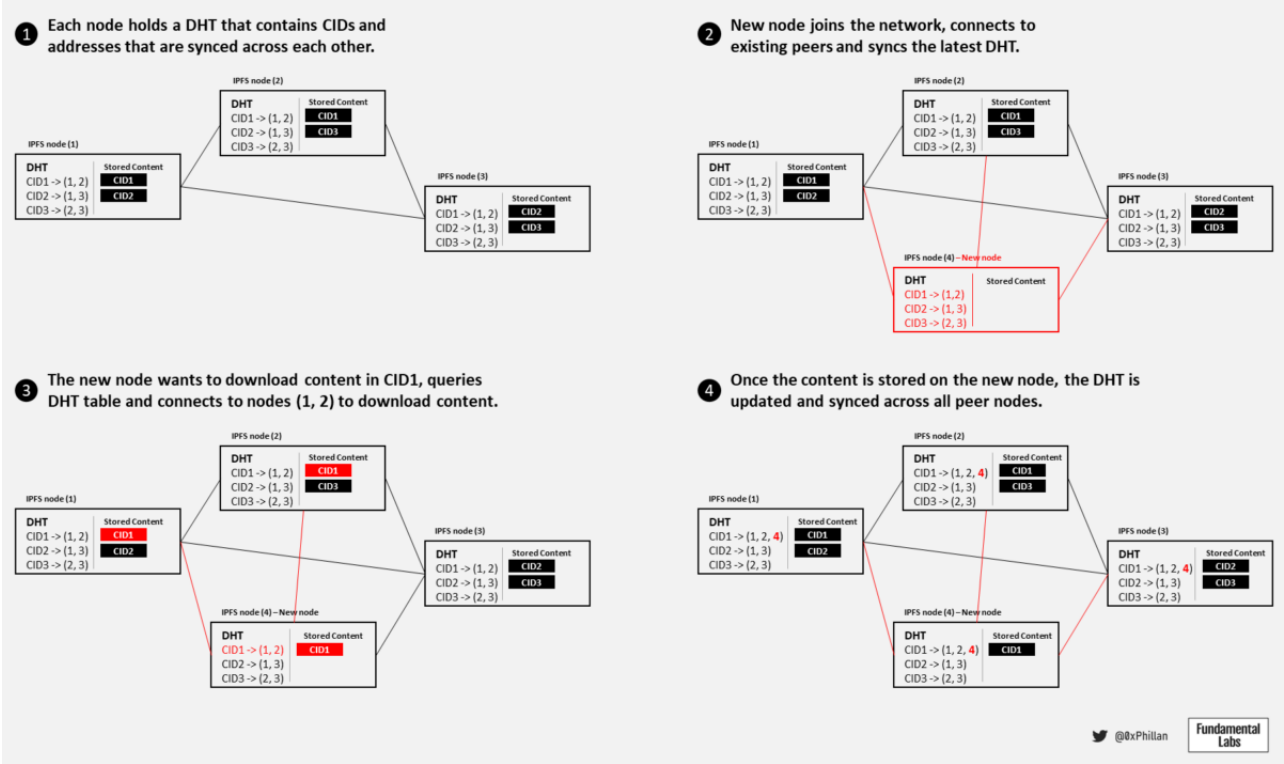


Figure 5: Brief overview of IPFS DHT and communication mechanism. Note: new nodes are bootstrapped with a list of common peers.[43]

The only weakness of IPFS is that content is only available as a node is willing to store the data. During the duration that the data is stored it is indeed immutable, i.e., the content referenced by a CID will never change. However, without any incentive, a node can remove the data or go offline entirely, making the content unavailable. Furthermore, to access IPFS content you need to either run your own node (you don't need to broadcast content, but you will have DHTs that are regularly updated) or you need to communicate with an IPFS gateway, which is a node hosted by a third party that helps transmit the data you want to retrieve. Filecoin aimes to solve these issue .

2.5 Filecoin: Incentivization for Decentralized Cloud Storage

Filecoin is a decentralized storage network that introduces financial incentives to encourage nodes (known as Storage Providers) to offer and maintain storage capacity over a blockchain-based marketplace. By leveraging the InterPlanetary File System (IPFS) for content addressing, Filecoin augments it with mechanisms for data persistence, where clients pay Storage Providers in the network's native cryptocurrency (FIL) to store their data. This design aims to overcome one of IPFS's main challenges: nodes may not have sufficient motivation to continue storing content for which they have no personal need.

In Filecoin, nodes dedicate storage resources and periodically submit proofs to demonstrate they are indeed

storing the data they agreed to keep. For this, Filecoin introduces two core proofs:

- **Proof-of-Replication (PoRep):** Ensures that a unique physical copy of the data (a replica) has been created.
- **Proof-of-Spacetime (PoSt):** Verifies that the data is continuously stored over an agreed period.

These proofs are submitted on-chain, and failing to produce them can result in penalization or slashing of the locked collateral. Successfully submitted proofs, on the other hand, allow Storage Providers to earn block rewards and client fees.The Bartering Protocol is greatly inspired from Filecoin's Proof of Replication mechanism, which we adapted to enhance the reliability of our proposed protocol.

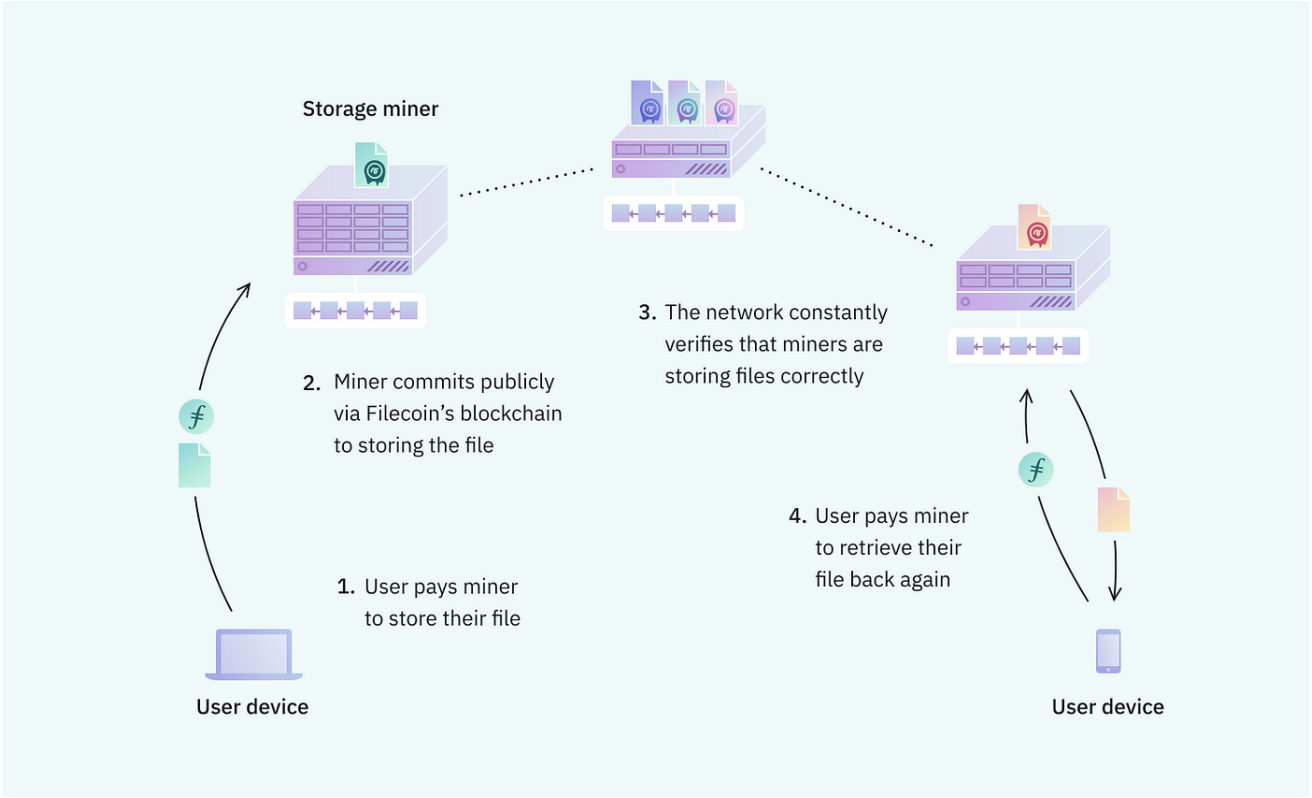


Figure 6: High-level schematic of the Filecoin network architecture .[34]

Filecoin is composed of two interlinked marketplaces that define how transactions take place:

- **Storage Market:** Clients seeking long-term storage broadcast their requirements, including duration and price they are willing to pay. Storage Providers place bids, and once a deal is made, the data is sealed on the provider's hardware.
- **Retrieval Market:** Focuses on the unsealed or cached copies. It allows specialized retrieval providers to quickly respond to data requests by users. This market is designed for lower-latency access.

Incentivizing both storage and retrieval ensures a more sustainable ecosystem: large providers can commit resources for long-term deals, and smaller or more strategically placed providers can optimize for faster retrievals.

2.5.1 Pros and Cons of Filecoin's Approach

Filecoin represents one of the most comprehensive blockchain-based solutions for decentralized storage, with a well-designed incentive structure and technical architecture. However, like any system, it comes with both advantages and limitations that are important to understand.

Pros: Filecoin offers several significant advantages as a decentralized storage solution:

- **Strong Incentive Model:** Filecoin leverages a native token (FIL) to reward providers, aligning economic incentives with reliable data storage.
- **Data Persistence and Security:** The proof-of-storage algorithms (PoRep, PoSt) offer robust guarantees that the data is consistently stored. Providers risk losing collateral if they fail to store the data, ensuring higher fidelity than typical unverified hosting.
- **Scalability and Decentralization:** As more providers join, storage capacity grows organically. Providers are distributed globally, mitigating single points of failure.
- **Integration with IPFS:** By building on IPFS' content addressing, Filecoin ensures that data remains addressable by cryptographic hashes. This synergy promotes a seamless decentralized storage and retrieval experience.

Cons: Despite its advantages, Filecoin faces several challenges:

- **Complexity of Participation:** Becoming a Storage Provider can require high-end hardware (for sealing and proof generation). The time-consuming sealing process can create barriers to entry.
- **Collateral and Volatility:** Providers must lock FIL as collateral. Fluctuations in token price introduce risk for both Providers and clients, potentially leading to higher operational costs.
- **Retrieval Delays:** Due to the sealing/unsealing steps, retrieving large datasets quickly can be challenging unless a node participates in the retrieval market and maintains unsealed copies.
- **On-Chain Overheads:** Submitting proofs and deal transactions on-chain involves fees and network congestion risks. This can reduce speed or increase costs during peak network usage.

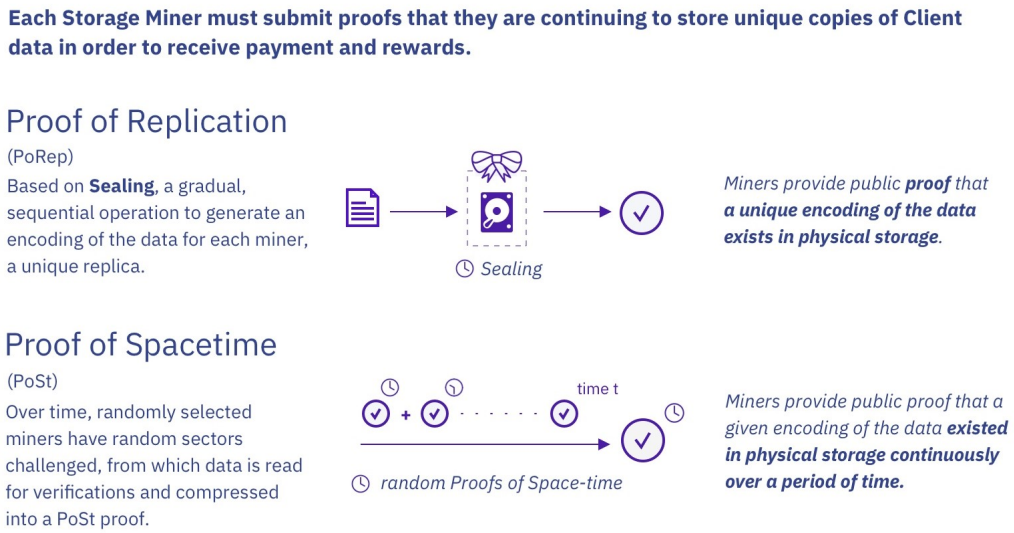


Figure 7: Conceptual illustration of Filecoin's proof-of-replication. [32]

Relevance to Our Bartering Protocol Our bartering-based storage protocol was inspired by the concept of incentivizing participants to store data that is not inherently valuable to them. Filecoin demonstrates how a robust economic layer can effectively encourage data availability. However, our design seeks to simplify some of Filecoin's heavier requirements:

- **No Native Cryptocurrency:** Instead of introducing token-based rewards, we use a storage-for-storage approach, reducing complexity and removing dependence on volatile market-driven tokens.
- **Lightweight Proof Mechanisms:** Unlike Filecoin's detailed PoRep and PoSt, we employ simpler methods (with an option to integrate more advanced proofs) to confirm data retention, lowering computational overhead.
- **Flexible Replication Strategies:** Filecoin's on-chain proofs require strict hardware sealing. Our approach dynamically negotiates storage deals with peers based on bartering ratios, which can adapt to resource availability without incurring large sealing costs.
- **Time-Based Storage Audits:** Inspired by PoSt, which demands Storage Providers demonstrate continued possession of client data throughout an agreed interval, our protocol similarly sets a recurring schedule for verifying stored content. The goal is to mimic PoSt's time factor: peers must periodically respond with verifiable proofs to maintain their reputation and score. Failure to respond or repeated unavailability reduces a peer's score and can trigger re-replication requests elsewhere. Although we do not use an on-chain mechanism like Filecoin, the recurring "spacetime-like" audits ensure that the system continuously tracks whether data remains intact at each replica.

Despite these differences, Filecoin offers a key demonstration that meaningful incentives transform a best-effort storage network into a more trustless and reliable data persistence layer.Also, these adaptations of PoRep and PoSt allow our bartering system to remain trustless while staying lightweight and flexible. Nodes must "prove" they genuinely store data over time, but they do so with simpler, more dynamic checks suited to an environment that does not rely on sealed commitments or a fully blockchain-based infrastructure.

2.6 Load balancing in peer to peer energy trading networks

While load balancing in energy trading and edge computing are not exactly the same problem, analyzing the former is beneficial to our study as there is extensive literature available. Because energy is not easy to store efficiently, electricity providers and consumers who have the means to produce energy need to adapt their production to the current demand at any given moment. Given this demand is not easy to predict, and that if electricity is not used or stored, it will be wasted, the smart grid was introduced. It allows energy producers to sell leftover production to anyone else on the grid. For a while, this was done in a centralized manner [33], but research is increasingly leaning towards decentralized approaches.

We could make an analogy with cloud and edge computing, where a provider who has unused resources could be willing to make them available. In [33], authors present a decentralized peer to peer energy trading market based on blockchain. Here, smart buildings collect and can classify themselves into three categories: energy surplus, energy deficit, or equilibrium. This data is sent to a virtual layer, where a blockchain is implemented. Network agents can then decide if they want to sell energy or need to buy it. A smart contract takes care of verifying if the energy transfer is valid, and authorizes it. The system thus creates a trustless energy market where available power can be sold, but also where the price of the last transaction is saved and used to determine the current market price of energy.

From a broader perspective, these concepts serve as building blocks for understanding how decentralized systems like our bartering protocol can efficiently manage resources under unpredictable conditions. In energy trading, balancing fluctuating supply and demand is paramount: surplus energy should not go to waste, and deficits should be met in a timely manner. Analogously, in a decentralized storage network, nodes holding excess capacity can offer it, while nodes requiring storage resources can leverage those offers. Both scenarios rely on dynamic market-like mechanisms (real-time price adjustments or credit metrics) to match surplus to demand in a trustless environment.By highlighting load balancing and trustless exchange in energy networks, we draw conceptual parallels for our storage protocol.

First,in an energy grid, oversupply or undersupply negatively impacts the system. Likewise, in bartering-based storage, balancing who provides and who receives storage is critical for efficiency. Methods used in energy trading (p2p decentralized bidding or dynamic pricing) inspire how we can negotiate storage ratios and maintain a fair exchange among nodes.Also, just as [33] employs blockchain smart contracts to secure and validate energy transfers, our bartering system uses local score metrics and periodic proofs to maintain trust and fairness, without a central authority.

Through this lens, the references on peer to peer energy trading help us understand not only how to handle resource surplus or scarcity, but also how to design a system that is robust to frequent changes in supply, demand, and participant availability precisely the challenges we address in our bartering protocol.

3 Energy Challenge : From IPFS & Filecoin to a Bartering Solution

3.1 Problem presentation

Although IPFS and Filecoin address key aspects of **data availability** and **replication**, recent research underscores how each system's design decisions can create considerable energy overheads, particularly when extended to large-scale or heterogeneous networks [37], [41], [53].

- **Prolonged Sealing and Proof Mechanisms in Filecoin.** While Filecoin's proof-of-replication (PoRep) and proof-of-spacetime (PoSt) drastically improve trustless assurance of data storage, multiple studies [53] show they incur a heavy CPU and memory footprint during cryptographic "sealing," resulting in substantial daily electricity consumption. This is worsened by the high churn of storage providers , who join or leave depending on electricity prices, hardware constraints, or profitability. Filecoin's consensus block production (miner, block validation, collateral on-chain processes) can add overhead for smaller or resource-constrained SPs, often yielding disproportionately high energy bills [53].
- **Stale Replication in IPFS.** In IPFS, objects with low popularity or niche interest risk being under-replicated or lost altogether, since the DHT-based discovery relies heavily on a peer actively **pinning** or **caching** data [37]. Maintaining partial replicas for little-used files can lead to idle nodes that remain online longer than necessary, needlessly consuming power [41]. Conversely, popular data often becomes over-replicated, which can, ironically, also lead to a surge in baseline energy usage by nodes continuously caching or "gossiping" about data they scarcely need [37].
- **Excessive On-Chain Overheads.** For Filecoin in particular, the on-chain logic that continuously rewards or penalizes storage providers can bloat consensus operations. As [53] point out, frequent block proposals and proofs, each with **storage deals** and **collateral checks**, amplify CPU cycles. This overhead can become disproportionate for small storage providers who store minimal data but must still participate in repeated proofs.

Comparative Overview. Table 1 offers a concise comparison among IPFS, Filecoin, and our solution , highlighting the trade-offs in energy overhead, reliance on tokens, and replication strategies.

Table 1: High-level comparison of IPFS, Filecoin, and our Proposed solution.

Criteria	IPFS	Filecoin	Bartering	Remarks
Requires blockchain token?	No	Yes	No	Filecoin incentivizes via FIL; bartering is tokenless.
On-chain overhead?	N/A	High	None	Bartering manages deals off-chain; Filecoin uses proofs on-chain.
Energy intensity	Moderate	Significant	Low	Filecoin sealing & PoSt top big CPU usage [53].
Replication driver	Popularity-based	Deals + PoSt	Storage-for-storage	IPFS weak on niche data, bartering ensures minimal replication.

Significance for Edge Networks: Both IPFS and Filecoin face heightened challenges in churn-heavy or IoT-based environments, where ephemeral connections and modest CPU budgets can exacerbate already high overheads. [37] observe that nodes with brief online windows must either replicate or relinquish data, each scenario harming availability or increasing idle energy usage. Under Filecoin's model, these nodes often cannot cover sealing costs or are forced to pay penalty fees [53]. Hence, forging an **energy-conscious** alternative is essential.

All these factors can lead to **undesirable power usage patterns** in heterogeneous, churn-heavy settings such as edge networks, where nodes may appear and disappear frequently. Hence, **an alternative approach is needed** one that maintains incentives for long-term storage while avoiding the heavy computational overhead of cryptocurrency-based proofs.

3.2 Proposal

We propose a novel **bartering-based storage protocol** that removes the need for token-based incentives or expensive cryptographic proofs, focusing instead on a **storage-for-storage** principle:

- Each node **offers** a portion of its own storage capacity to the network.
- In return, it **expects** to have a similar (or negotiated) amount of its own data stored by other peers.
- A **local "score" metric** tracks whether a node reciprocates storage requests reliably.

If a node consistently **answers** proof-of-storage checks (§3.4.1) and **accepts** valid bartering deals, its score remains high. Conversely, refusing requests, failing to prove data retention, or frequently going offline lowers its score. This score system thus **encourages** stable participation without relying on cryptocurrency-based mechanisms. Because our proof-of-storage checks are lightweight (verifying SHA-256 file hashes), we reduce both the **CPU load** and **on-chain overhead** typical of Filecoin or IPFS-based blockchains.

Before, if a user wants to store personal files that hold little interest for anyone else, the lack of explicit requests from peers leads to two major obstacles. First, there are no direct incentives for others to fetch or store

such niche data. Second, long-term persistence of these files is discouraged, since storing content that almost no one else wants yields little reciprocal benefit. Consequently, purely popularity-driven replication fails to serve personal or low-demand data well.

Hence,we presents the detailed design of our bartering protocol and including:

1. A **Score-based acceptance strategy** that determines which nodes store whose data, adapted to local node churn .
2. A **Lightweight storage-proof mechanism** replacing heavier blockchain verifications which are energy intensive.

By forgoing the full complexity of a blockchain ledger, we **naturally reduce** redundant operations and keep storage overhead commensurate with actual user demands, making our approach more **energy-aware** and more practical for resource-constrained or churn-heavy networks. The following sections explain the protocol internals and its integrations with realistic workloads.

3.3 General description

Our approach uses bartering. The idea is for each peer to offer storage space to the network, and in return get storage space from other peers. For that, we introduce 3 message types :

1. **Storage requests messages** which are used to request storage from other peers. They contain the data's IPFS CID, which can be used to retrieve data through IPFS.
2. **Ratio requests messages** , which are used to negotiate the amount of storage one has at a given peer. They contain a new request proposal in the form of a float value.
3. **Storage proof request messages**, which are used to request a storage proof from other peers. They contain the IPFS CID of the data to test.

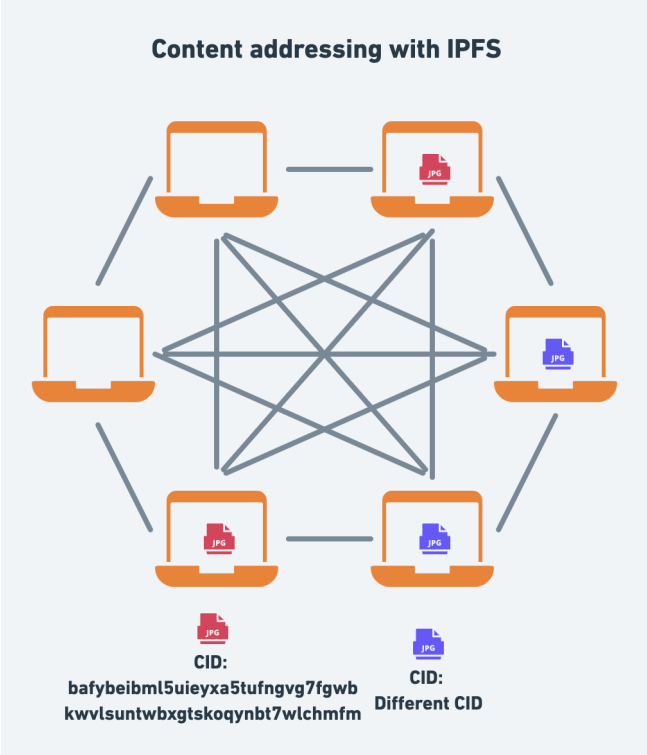


Figure 8: A Practical Explainer for CID

Whenever new data needs to be stored, a peer sends storage requests to a portion of his peers, who can either accept or refuse this request. If the request is accepted, the peer that requested storage will start sending storage proof requests periodically to confirm its data is still stored. If one of the storers cannot answer one of the requests, the peer that requested storage will look for a new storer, to consistently ensure the data is replicated. In the next section, we'll describe different strategies to implement the previously described interactions .

3.4 Strategies

Peers of the network maintain a score for each one of their peer. All decisions regarding these requests will rely on this score, which is representative of the current relationship between two nodes. From a qualitative point of view, a high score means that a peer consistently provides storage proofs and accepts storage requests. A low score means a peer is unreliable (that is, cannot provide storage proofs), and/or refuses storage requests. Each of the previously presented components will have an effect on the score, as it will be discussed next.

3.4.1 Storage proof

We'll start by describing how peers test the availability of their data stored by other peers through storage proof messages. Storage availability testing is necessary as peers need to determine if their storage requests are indeed being met. For this, they can periodically request storage proofs.

Proof of storage is a vast and complex topic that is out of scope for this paper. However, we can imagine

reusing Filecoin's [25] proof of storage mechanism, which is suitable to prove that content has been replicated across a network of mutually untrusting peers. In our first implementation, however, we chose an extremely simple proof of storage system that merely involves computing the SHA-256 hash of the file to test. This approach can be easily gamed (a dishonest peer might compute the hash once and then discard the data), but we adopt it here to focus on protocol design rather than a bulletproof cryptographic proof. In some contexts such as a trusted edge environment or a cooperative setting this lightweight approach may suffice.

It is up to a peer to request proofs, and many strategies are possible regarding how often these requests occur. Testing more frequently lets us detect node failures sooner but also consumes more network bandwidth and CPU resources. One could test all files at regular intervals, or use an adaptive strategy, requesting proofs more often from peers with lower scores. Similarly, only a random subset of files might be tested at each epoch.

When a peer receives a proof request message, it must respond before a given deadline. A correct, on-time response increases its score; a wrong or late response decreases it. Hence, nodes that consistently answer requests accurately will see their scores rise, signaling reliability; while those that fail or ignore requests will see their scores drop and risk losing future storage deals.

This module thus allows nodes to audit the storage they have requested, providing a score-based trace of peers' past behaviors. It also guarantees that unresponsive peers can be discovered in a timely fashion. In Figure 9, we summarize how a node behaves when it is either requesting or providing a proof of storage.

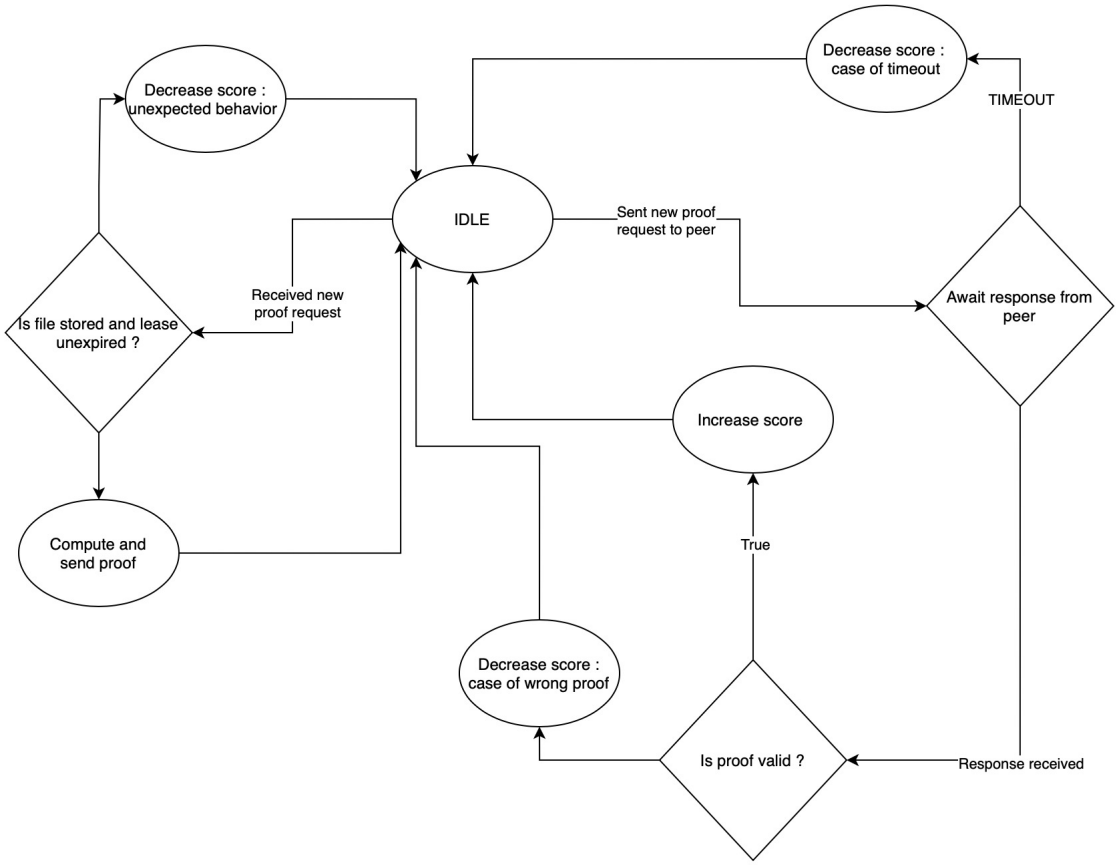


Figure 9: Storage proof request behavior

After detecting that one of the peers storing a file has not provided the requested proof, redundancy is no longer guaranteed. To restore it, the system proactively requests an additional replica from a different peer.

Algorithmic Description of the Storage Proof Process.
In Algorithm 2, we provide a more detailed pseudo code for how the requester (the node verifying storage) and the responder (the node storing the file) interact. We also show how the corresponding score adjustments and re-replication triggers are handled.

Algorithm 2 Storage Proof Procedure(`Requester`, `Responder`, `FileID`, `Deadline`)

Require:
Requester: Node initiating proof check
Responder: Node that claims to hold `FileID`
FileID: Identifier for the stored content (IPFS CID)
Deadline: Maximum wait time for a valid proof

Ensure:
Outcome: Adjust `Responder`'s score, possibly trigger re-replication

— Requester Side —

1: `Requester` sends PROOFREQUEST(`FileID`) to `Responder`
2: `startTime` $\leftarrow$ NOW(())
3: **while** NOW(()) $-$ `startTime` $<$ `Deadline` **do**
4: **if** HASRESPONSE(()) **then**
5: `resp` $\leftarrow$ GETRESPONSE(())
6: **if** ISHASHVALID(`resp`, `FileID`) **then** ▷ Correct proof arrived in time
7: `Score`[`Responder`] $\leftarrow$ `Score`[`Responder`] $+ \Delta_{\text{valid}}$
8: **return**
9: **else** ▷ Wrong or null proof
10: `Score`[`Responder`] $\leftarrow$ `Score`[`Responder`] $- \Delta_{\text{invalid}}$
11: TRIGGERREREPLICATION(`FileID`)
12: **return**
13: **end if**
14: **end if**
15: **end while**
▷ Deadline exceeded without valid proof
16: `Score`[`Responder`] $\leftarrow$ `Score`[`Responder`] $- \Delta_{\text{timeout}}$
17: TRIGGERREREPLICATION(`FileID`)
18: **return**

— Responder Side —

19: **function** ONPROOFREQUEST(`FileID`)
20: **if** ISLEASEUNEXPIRED(`FileID`) $\wedge$ FILEEXISTS(`FileID`) **then**
21: `hash` $\leftarrow$ COMPUTESHA256(`FileID`)
22: SENDRESPONSE(`hash`)
23: **else** ▷ Either lease is expired or file is missing
24: SENDRESPONSE(*null*)
25: **end if**
26: **end function**

The storage proof algorithm operates through several key phases that together create a robust verification system:

1. **Proof Initiation (Lines 1-2):** The requester begins by sending a PROOFREQUEST containing the file identifier to the responder. This message essentially asks, "Do you still have my data?" Immediately, the system records a timestamp to track the response deadline.

2. **Response Monitoring Loop (Lines 3-15):** The algorithm enters a waiting state, continuously checking for a response until either one arrives or the deadline expires. This bounded waiting ensures the protocol remains responsive even when peers fail to reply.

3. **Response Validation (Lines 4-14):** When a response arrives, the system checks if the provided hash matches the expected value:
    - For valid responses (Lines 6-8), the responder's score increases by Δ_{valid}, reinforcing reliable storage behavior through positive feedback
    - For invalid or null responses (Lines 9-13), the score decreases by Δ_{invalid} and triggers the re-replication process to restore redundancy
4. **Timeout Handling (Lines 16-18):** If the deadline passes without any response, the system applies a score penalty (Δ_{timeout}) and initiates re-replication. This ensures the system doesn't wait indefinitely for unresponsive nodes.
5. **Responder-Side Processing (Lines 19-26):** Upon receiving a proof request, the responder first validates both temporal conditions (unexpired lease) and storage conditions (file existence). When both conditions are met, it computes and returns the file hash; otherwise, it sends a null response.

Discussion and Impact.

- **Robustness vs. Overhead:** By adjusting Deadline and the frequency of proof requests, we can tune the responsiveness of the system. Short deadlines catch failures quickly but risk penalizing peers in high-latency environments; frequent proofs maintain reliability but create additional bandwidth and CPU overhead.
- **Scoring Incentives:** The score system strongly incentivizes nodes to keep the data they agree to store. Reliable nodes accumulate a higher score, making it easier to negotiate future storage deals. Conversely, repeated timeouts or invalid hashes rapidly degrade a node's reputation, leading to fewer or no storage requests.
- **Re-Replication:** Any failure to present a valid proof triggers TRIGGERREREPLICATION, which launches a new request to replicate the data from alternate, higher-score peers. This ensures that the system maintains the desired number of replicas over time despite churn or dishonesty.
- **Scalability:** In large networks, scheduling proofs selectively (e.g., focusing on suspicious peers or random subsets) can significantly reduce overhead. Adapting the proof frequency to the node's score (probing low-score peers more often) can further optimize efficiency.

3.4.2 Bartering

Peers in the network can barter to decide how much data they store for others. We introduce a ratio metric k that dictates how many bytes a peer should store for another peer relative to what the other peer stores for it. Concretely, if Peer A maintains a ratio k for Peer B, that means that for each byte B stores on A, A will store k bytes on B. By default, every node begins with a ratio of 1.0.

Figures 10 and 11 illustrate how a node initiates a new bartering negotiation (Figure 10) and how another node responds to that negotiation (Figure 11).

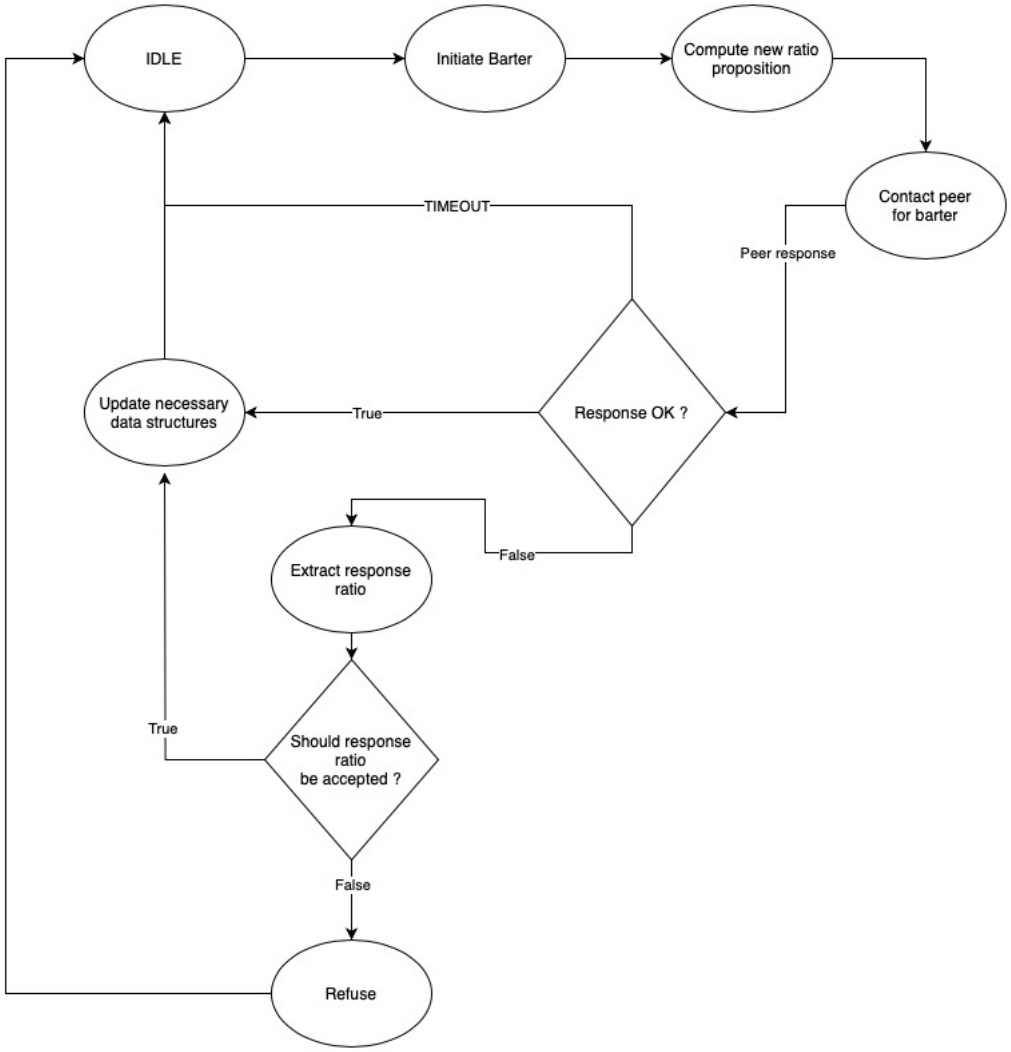


Figure 10: Bartering scheme (initiation of a barter)

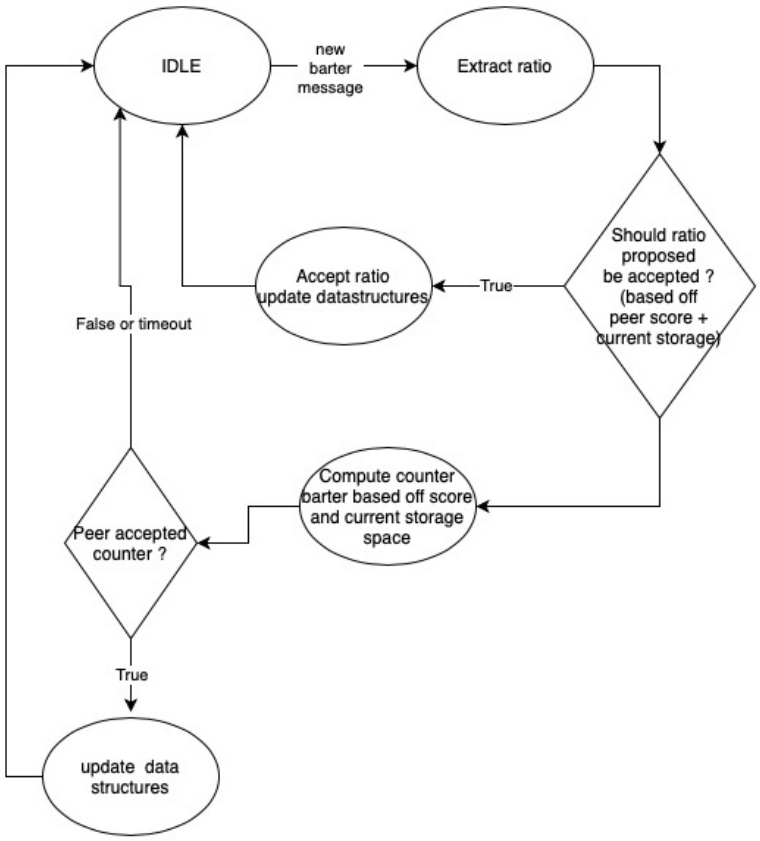


Figure 11: Bartering scheme (responding to a barter)

Several strategies are possible to decide whether a proposed ratio k should be accepted or refused. Naturally, a node's score plays a major role: peers that have a good record of reliability should be rewarded with more favorable bartering terms. However, each node's available storage is also constrained both physically and by the desire to ensure enough space to serve multiple peers.

For example, suppose a peer wants at least K replicas of its data on the network. It needs to maintain good scores with K distinct peers in order to get reciprocal storage from each of them. Hence, whenever a peer receives a new bartering request, it must check:

1. Whether the requester's score is sufficiently high (above some threshold).
2. Whether accepting the request leaves enough capacity to maintain bartering relationships with the other $K - 1$ peers.

This ensures that while we allow bartering ratios to vary, we do not compromise the node's overall capacity to store multiple copies of its own data elsewhere.

Algorithmic Description of the Bartering Process.

Algorithm 3 provides a pseudo-code view of how peers initiate and respond to barter proposals. It merges the logic depicted in Figures 10 and 11.

The bartering algorithm operates as a multi-phase negotiation protocol that balances individual node interests with system-wide storage distribution:

1. **Ratio Calculation and Proposal (Lines 1-3):** The initiator node computes an optimal storage ratio k based on the peer's reliability score and available resources. This ratio represents how many bytes the responder should store for each byte it places on the initiator. After sending the BARTERREQUEST, the initiator starts a timeout counter to ensure timely completion of the negotiation.

2. **Response Monitoring Loop (Lines 4-23):** The initiator continuously checks for a response until either one arrives or the timeout expires. This bounded waiting period ensures protocol responsiveness even under network delays or disruptions.

3. **Response Processing (Lines 5-22):** Upon receiving a response, the system takes action based on the status code:

   - **ACCEPT (Lines 7-9):** When the responder accepts the proposed ratio, the initiator updates its local data structures to record the new agreement and completes the negotiation.
   - **COUNTER (Lines 10-18):** If the responder proposes an alternative ratio, the initiator evaluates this counteroffer using `ShouldAcceptCounter`. For acceptable counteroffers, the initiator updates its structures and confirms acceptance; otherwise, it sends a refusal message.
   - **REFUSE (Lines 19-21):** When facing outright rejection, the initiator simply acknowledges and terminates the negotiation process.

4. **Timeout Management (Lines 23-25):** If the waiting period elapses without a valid response, the initiator invokes `HandleBarterTimeout` to apply appropriate score adjustments for unresponsiveness.

5. **Responder-Side Logic (Lines 26-37):** When receiving a barter request, the responder evaluates the proposed ratio against the initiator's reliability metrics and its own resource constraints:

   - If the ratio is acceptable (Line 27), the responder immediately confirms acceptance (Line 28).
   - Otherwise, the responder checks if counterproposal is viable (Line 30). If capacity exists, it calculates a more suitable ratio based on current scores and resource availability (Lines 31-32).
   - If neither option is feasible due to capacity constraints or insufficient trust, the responder sends an outright refusal (Line 34).

Algorithm 3 Barter Procedure(`Initiator`, `Responder`, `ProposedRatio`, `Timeout`)

Require:
Initiator: Node that starts the barter (computing new ratio)
Responder: Node receiving the ratio proposition
ProposedRatio (k): Bytes to be stored by **Responder** for each byte it stores on **Initiator**
Timeout: Upper bound on waiting time for an acknowledgment

Ensure:
Updates local data structures and possibly sets a `counterRatio` if **Responder** counters

— Initiator Side —

`Initiator` computes a *new ratio k* based on peer scores and available storage
`Initiator` sends BARTERREQUEST(`ProposedRatio`) to `Responder`
`startTime` ← NOW(())
while NOW(()) − `startTime` < `Timeout` **do**
  if HASRESPONSE(()) **then**
    `respMsg` ← GETRESPONSE(())
    if `respMsg.status` = `ACCEPT` **then** ▷ Responder accepted the ratio k
      UPDATEBARTERSTRUCTURES(`Initiator`, `Responder`, `ProposedRatio`)
      return
    else if `respMsg.status` = `COUNTER` **then** ▷ Responder proposes a counter-ratio
      `counterRatio` ← `respMsg.ratio`
      if SHOULDACCEPTCOUNTER(`Responder`, `counterRatio`) **then**
        UPDATEBARTERSTRUCTURES(`Initiator`, `Responder`, `counterRatio`)
        SENDACCEPTANCE(`Responder`, `counterRatio`)
      else
        SENDREFUSAL(`Responder`)
      end if
      return
    else if `respMsg.status` = `REFUSE` **then** ▷ Responder refused the proposition
      return
    end if
  end if
end while ▷ If loop ends, `Timeout` reached without valid answer
HANDLEBARTERTIMEOUT(`Responder`)
return

— Responder Side —

function ONBARTERREQUEST(`ProposedRatio`)
  if SHOULDACCEPTRATIO(`Initiator`, `ProposedRatio`) **then**
    SENDACCEPTANCE(`Initiator`, `ProposedRatio`)
  else
    if HASCAPACITYFORCOUNTER(`Initiator`) **then** ▷ Compute new ratio based on current score and space
      `counterRatio` ← COMPUTECOUNTERRATIO(`Initiator`)
      SENDCOUNTER(`Initiator`, `counterRatio`)
    else ▷ No capacity or insufficient score to store more data for this peer
      SENDREFUSAL(`Initiator`)
    end if
  end if
end function

Discussion and Impact.

- **Score-Driven Negotiation:** Nodes typically grant better ratios to peers that have proven their reliability (via higher scores). This fosters reciprocal altruism: reliable peers enjoy more favorable deals, while unreliable peers either get refused or forced to accept less advantageous ratios.
- **Counteroffers and Adaptation:** The ability to send a *counterRatio* lets a node propose a more balanced or more feasible trade. If the counter is too low, the initiator may reject it; if it is too high, the initiator might refuse. This iterative process helps converge toward an arrangement that both sides find acceptable.

With these mechanisms, bartering becomes a flexible and dynamic tool that allows nodes to fine-tune how much they store on each other, leading to more efficient use of scarce resources while keeping the network robust. A node that maintains good scores with multiple peers can confidently spread its data among them, secure in the knowledge that those peers have an ongoing incentive to store it.

3.4.3 Storage requests

Peers in the network can request storage from other nodes via storage request messages. Each request contains all the information needed to retrieve and pin the file through IPFS (specifically, the file's CID). When a node wants to store a file with a desired replication factor of K, it must locate at least K peers willing to host that file knowing that some peers may refuse due to limited capacity or low incentives.

In choosing which peers to contact, a node could simply target those with the highest scores, but this can overlook newly recovering peers whose scores dropped temporarily (e.g., due to downtime). Instead, an alternative is to sample both high-score and lower-score peers, balancing reliability with the possibility that lower-score peers might still be able to provide capacity.

Once a peer receives a storage request, it decides whether to accept or refuse based on:

- The requester's score (its reliability/performance history).
- The requester's bartered storage ratio, determining how much the peer stands to gain in reciprocal storage.

If the request is accepted, the data is pinned locally through IPFS and the requester is notified of success. Conversely, a refusal might signal that the node lacks capacity or trust in the requester's future reciprocity; in this case, the requester decreases that peer's score accordingly.

Moreover, storage requests in this protocol model are leased, meaning they have an expiration date after which the peer is free to remove the data unless the requester renews the lease (or arranges replication elsewhere). This ensures that unused or stale allocations do not occupy valuable capacity indefinitely, while also requiring a requester to maintain its score so that peers continue to accept renewal requests.

Algorithmic Description of the Storage Request Process. (Proposed Pseudo-Code)
Algorithm 4 provides a conceptual pseudo code for how nodes issue these requests and how receiving peers decide acceptance or refusal.

Algorithm 4 Storage Request Procedure(`Requester`, `FileCID`, `kCopies`, `Timeout`)

Require:
Requester: Node that needs to store data
FileCID: Unique content identifier (IPFS CID)
kCopies: Desired number of replicas
Timeout: Maximum wait for responses

Ensure:
The **Requester** tries to secure `kCopies` acceptance(s), or handles refusals/timeouts

— Requester Side —

1: **function** ISSUESTORAGEREQUESTS(`FileCID`, `kCopies`)
2: `candidatePeers` $\leftarrow$ SELECTPEERS(`kCopies` $\times \alpha$, `kCopies` $\times (1-\alpha)$)
3: **for all** `peer` **in** `candidatePeers` **do**
4: SENDSTORAGEREQUEST(`peer`, `FileCID`)
5: **end for**
6: `startTime` $\leftarrow$ NOW(())
7: `acceptedCount` $\leftarrow 0$
8: **while** NOW(()) $-$ `startTime` $<$ `Timeout` **do**
9: **if** HASRESPONSE(()) **then**
10: `resp` $\leftarrow$ GETRESPONSE(())
11: **if** `resp.status` = `ACCEPT` **then**
12: `acceptedCount` $\leftarrow$ `acceptedCount` $+ 1$
13: **if** `acceptedCount` = `kCopies` **then** ▷ Target achieved
14: **return**
15: **end if**
16: **else if** `resp.status` = `REFUSE` **then**
17: DECREMENTSCORE(`resp.fromPeer`)
18: **end if**
19: **end if** ▷ Continue waiting until `Timeout` or we have `kCopies` acceptances
20: **end while**
▷ If we exit the loop without enough acceptances
21: HANDLEINSUFFICIENTSTORAGE(`FileCID`, `kCopies`, `acceptedCount`)
22: **return**
23: **end function**

— Responder Side —

24: **function** ONSTORAGEREQUEST(`FileCID`, `Requester`)
25: **if** EVALUATEREQUEST(`Requester`, `FileCID`) **then** ▷ Checks requester's score, ratio, local capacity
26: PINDATA(`FileCID`)
27: SENDRESPONSE(`Requester`, `ACCEPT`)
28: **else** ▷ Not enough capacity or low trust in requester
29: SENDRESPONSE(`Requester`, `REFUSE`)
30: **end if**
31: **end function**

The storage request algorithm operates through several distinct phases that collectively ensure reliable data distribution:

1. **Peer Selection and Request Distribution (Lines 1-5):** The requester begins by selecting potential storage providers using a balanced approach. The `SelectPeers` function employs an α parameter to divide candidates between high-scored peers (α portion) and lower-scored peers ($(1-\alpha)$ portion). This hybrid

selection strategy prevents network fragmentation while still rewarding reliable nodes. The requester then broadcasts storage requests to all candidates simultaneously.

2. **Response Tracking Initialization (Lines 6-7):** The algorithm records the start time and initializes an acceptance counter to track how many peers have agreed to store the data. This creates the foundation for both timeout detection and replication quota management.

3. **Response Collection Loop (Lines 8-20):** While waiting for responses (up to the specified timeout), the requester processes incoming messages:

   - For acceptances (Lines 11-15), the system increments the counter and checks if the target replication factor (`kCopies`) has been met. Once satisfied, it can terminate the process early.
   - For refusals (Lines 16-18), the system decrements the refusing peer's score, creating a negative incentive for nodes that reject storage requests without good cause.

4. **Insufficient Storage Handling (Lines 21-22):** If the timeout expires before securing enough acceptances, the system invokes `HandleInsufficientStorage` to implement recovery strategies. These might include retrying with different peers, using a more aggressive bartering ratio, or alerting the user about reduced redundancy.

5. **Responder-Side Logic (Lines 24-31):** When receiving a storage request, a node evaluates it based on multiple factors through the `EvaluateRequest` function (Line 25):

   - The requester's historical reliability (score)
   - The current bartering ratio between the nodes
   - Local storage capacity constraints

   Based on this evaluation, the node either accepts the request and pins the data to its local IPFS node (Lines 26-27), or refuses it due to capacity limits or trust concerns (Lines 29-30).

Discussion and Impact.

- **Peer Selection:** Rather than always picking only high-scored peers, an adaptive selection lets lower-score peers prove themselves if they are currently stable. This balances exploration and exploitation of available nodes in the network.
- **Acceptance Logic:** The `EvaluateRequest` function typically uses both the requester's score and bartering ratio to decide whether storing this data is worthwhile.
- **Dynamic Scoring:** Peers who refuse requests get penalized, which might lower their future chances of obtaining storage from others. On the other hand, consistently accepting (and providing proofs later) can increase a peer's score and reputation.

Overall, this storage request mechanism underpins the entire protocol's functionality. By tying the decision process to both score and storage ratio, it ensures that reliable peers are rewarded with reciprocal storage while nodes that fail to accommodate requests or prove reliability see diminishing opportunities to store their own data with others.

3.5 Incentive mechanisms and expected network dynamics

The three components we described work in tandem to ensure that nodes are incentivized to store others' data reliably. Central to our design is a local score maintained between pairs of nodes, which captures the history of successful storage and proof-of-storage interactions. This score not only determines a node's eligibility to receive storage services from its peers but also guides its own behavior in contributing storage resources. Peers aim to maintain a high score with other nodes because a high score is the primary means by which a node can obtain storage from others. To keep their score elevated, a node must:

1. **Reliably store data:** Each time a peer stores data for another node and subsequently provides a valid proof upon request, its local score with that peer increases.

2. **Accept higher storage ratios:** When a node has available storage, accepting more data increases the opportunities to earn proofs. More proofs, in turn, boost its score, further incentivizing the node to maintain its reliability.

Thus, the score serves as a mutual currency in the system: it is earned by providing a service (storing and retrieving data reliably) and spent indirectly as it grants access to storage resources from others.

Dynamic Maintenance of Storage Copies. As discussed earlier, each node expects that a certain number K of copies of its data will be stored by distinct peers. In practice, a node will try to maintain high scores with roughly K peers to ensure sufficient data redundancy. However, because storage is leased and nodes may intermittently fail or leave the network (a common scenario in edge environments), the specific set of K peers will fluctuate over time. This leads to several dynamic behaviors:

- **Periodic re-negotiation:** Nodes must continuously update their score relationships to reflect the current set of active peers.

- **Adaptive peer selection:** As network conditions change, nodes may choose to form new high-score relationships or drop peers whose reliability has declined.

This dynamic adaptation ensures that the network remains resilient despite churn, as each node actively seeks to preserve a robust set of storage partners.

Pre-established Trust and Organizational Relationships. In many real-world scenarios, certain nodes share pre-existing trust relationships, when they belong to the same organization. In such cases, these nodes might start with a higher baseline score or exhibit a tendency toward preferential storage agreements. However, even when favoritism exists, the underlying scoring mechanism remains adaptive:

- **Stable vs. exploratory behavior:** Nodes with pre-established trust may continue to rely on familiar peers but still periodically explore new partnerships to maximize their storage opportunities.

- **Balancing incentives:** The system can be tuned to ensure that while organizational trust benefits exist, they do not entirely preclude competition from other well-performing nodes.

Mechanism Flow and Network Dynamics. To illustrate the incentive process and the expected network dynamics, consider the flow diagram in Figure 12:

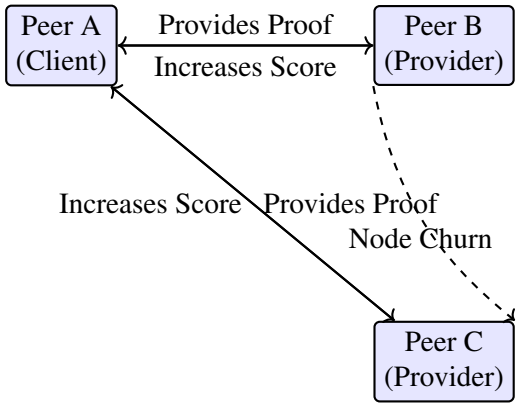


Figure 12: Flow diagram illustrating score-based interactions and dynamic peer relationships.

In this diagram, a client (Peer A) interacts with two providers (Peers B and C). Each time a provider successfully delivers data along with a valid proof, Peer A boosts its local score for that provider. The dashed arrow represents the inevitable network churn nodes may leave or join, prompting continuous re-evaluation of peer relationships.

Expected Network Dynamics. Nodes will tend to form a stable set of high-score relationships, ideally numbering close to K, which guarantees data redundancy. Due to node failures and departures, these relationships will be periodically reformed. A node may lose some high-score partners and must seek out new ones to maintain its desired level of data replication. In environments where certain nodes share pre-established trust (organizationally affiliated peers), these nodes may persist in high-score relationships longer, potentially leading to clusters of nodes that interact preferentially.

Overall, the incentive mechanism promotes a dynamic equilibrium where cooperation is continuously rewarded, and the network adapts to the inherent volatility of edge environments.

4 Modeling the Power & Energy Consumption of Nodes

In distributed edge environments implementing bartering-based storage protocols, comprehensive energy modeling becomes essential for understanding system sustainability and optimizing resource allocation strategies. This chapter presents a detailed framework for quantifying power consumption across heterogeneous node configurations, incorporating both computational and network overhead considerations specific to our protocol implementation.

4.1 Power Consumption Framework for Edge Nodes

The energy profile of edge computing nodes differs significantly from traditional datacenter environments, requiring specialized modeling approaches that account for diverse hardware configurations, intermittent connectivity patterns, and resource-constrained operation modes [16]. Our analysis focuses on the primary energy consumption sources relevant to bartering protocol deployment.

4.1.1 Host-Level Power Consumption

Edge nodes participating in the bartering protocol exhibit power consumption patterns that can be decomposed into static (baseline) and dynamic (utilization-dependent) components [18]. For a node i operating within our distributed storage network, the total power consumption $P_{total,i}$ is expressed as:

$$P_{total,i}(f,u) = P_{static,i}(f) + P_{dynamic,i}(f,u) \tag{1}$$

where $P_{static,i}(f)$ represents the baseline power consumption when the node remains idle but connected to the network, and $P_{dynamic,i}(f,u)$ captures the additional power drawn during active protocol operations.

The dynamic component scales with both CPU frequency f and utilization level u, following the relationship:

$$P_{dynamic,i}(f,u) = (P_{maximum,i}(f) - P_{static,i}(f)) \times u \tag{2}$$

Here, $P_{maximum,i}(f)$ denotes the peak power consumption achievable by node i at frequency f under full computational load, while $u \in [0,1]$ represents the normalized utilization factor.

Substituting equation (2) into (1) yields:

$$P_{total,i}(f,u) = P_{static,i}(f) + (P_{maximum,i}(f) - P_{static,i}(f)) \times u \tag{3}$$

This linear power model provides a first-order approximation suitable for our bartering protocol analysis, though real systems may exhibit non-linear characteristics quantified by the Linear Deviation Ratio (LDR) metric [14].

4.1.2 Edge Computing Power Characteristics

Unlike traditional datacenter servers that maintain static power consumption exceeding 100 watts [15], edge computing devices typically exhibit significantly lower baseline consumption. Community network participants and edge nodes generally demonstrate:

1. **Reduced Static Power**: Baseline consumption often remains below 40W, representing approximately half the static power of conventional server infrastructure
2. **Higher Effective Utilization**: Edge nodes frequently serve dual purposes (user applications and protocol participation), leading to improved energy efficiency per useful operation
3. **Intermittent Operation**: Unlike always-on datacenter equipment, edge nodes may enter low-power states during inactive periods

These characteristics directly impact the bartering protocol's energy profile, as nodes can optimize their participation patterns based on local energy constraints and availability requirements.

4.2 Network Communication Energy Model

4.2.1 Intra-Cluster Communication

For nodes operating within geographical proximity (such as our experimental Grid'5000 cluster deployment), communication energy consumption depends on the network topology and routing protocols employed. In hierarchical network configurations commonly found in edge deployments, the communication power $P_{comm,cluster}$ can be modeled as:

$$P_{comm,cluster} = \sum_{l=1}^{L} (P_{switch}(l) + P_{link}(l)) \tag{4}$$

where L represents the network depth, $P_{switch}(l)$ denotes switch power consumption at level l, and $P_{link}(l)$ indicates link power consumption.

The hierarchical structure directly influences energy efficiency: deeper network trees require more intermediate hops for bartering message exchange, increasing overall power consumption [13]. This relationship supports the distributed edge computing paradigm, where smaller, geographically distributed clusters often demonstrate superior energy efficiency compared to centralized mega-datacenter architectures.

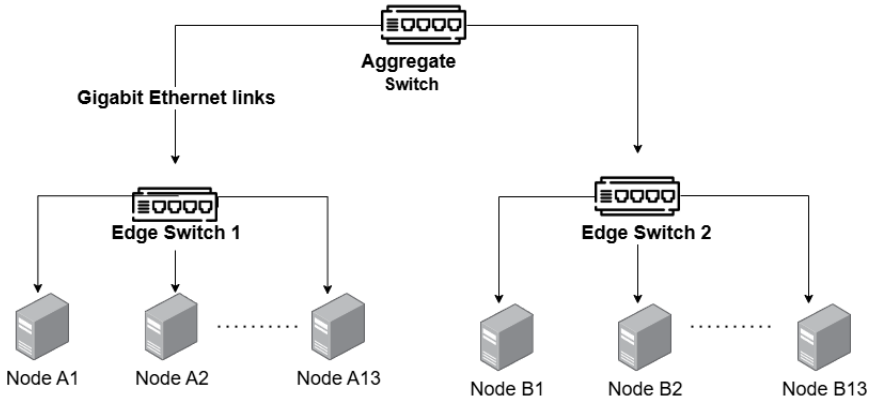


Figure 13: inter-cluster communication model

4.2.2 Inter-Node Protocol Communication

The bartering protocol's communication patterns introduce specific energy requirements for message exchange, storage proof validation, and ratio negotiation processes. For peer-to-peer communication within our protocol, the wireless network power consumption $P_{comm,p2p}$ encompasses:

$$P_{comm,p2p} = \sum_{h=1}^{H} N_{neighbors}^{(h)} \times P_{broadcast} + \sum_{i=1}^{N-1} P_{receive}(i) \tag{5}$$

where H represents the maximum hop count, $N_{neighbors}^{(h)}$ indicates the number of h-hop neighbors, $P_{broadcast}$ denotes broadcast transmission power, and $P_{receive}(i)$ represents reception power for node i.

This formulation accounts for the flooding-based discovery mechanisms inherent in decentralized storage protocols, where storage requests and proof challenges must propagate through the peer network.

4.2.3 Wide Area Network Overhead

For bartering protocol deployments spanning multiple geographical regions, Internet infrastructure energy consumption becomes relevant [10]. The Internet power model incorporates three hierarchical layers:

$$P_{internet} = P_{core} \times n_{core} + P_{distribution} \times n_{dist} + P_{access} \times n_{access} \tag{6}$$

where P_{core}, $P_{distribution}$, and P_{access} represent power consumption per hop at the core, distribution, and access network levels, respectively, while n_{core}, n_{dist}, and n_{access} denote the number of hops traversed at each level.

4.3 Energy Consumption Integration Model

4.3.1 Temporal Energy Calculation

The total energy consumption $E_{total,i}$ for node i over time interval T integrates power consumption across all operational phases:

$$E_{total,i}(T) = \int_0^T P_{total,i}(f(t), u(t))\, dt \tag{7}$$

For discrete monitoring systems (such as our Grid'5000 experimental setup with 1Hz sampling), this becomes:

$$E_{total,i}(T) \approx \sum_{j=1}^{N_{samples}} P_{total,i}(f_j, u_j) \times \Delta t \tag{8}$$

where Δt represents the sampling interval and $N_{samples} = T/\Delta t$.

4.3.2 Node Profile-Specific Energy Models

The bartering protocol's three-tier node classification (Benefactor, Peer, Peeper) exhibits distinct energy consumption patterns that must be incorporated into the overall model. For node profile k, the expected energy consumption becomes:

$$E_{profile,k} = \alpha_k \times P_{static} \times T_{uptime,k} + \beta_k \times P_{dynamic} \times T_{uptime,k} \tag{9}$$

where:

- α_k represents the fraction of uptime spent in idle state for profile k
- β_k denotes the fraction of uptime spent in active processing
- $T_{uptime,k}$ indicates the average uptime duration for profile k

4.3.3 System-Wide Energy Aggregation

The complete energy model for a bartering protocol deployment encompassing N nodes integrates individual node consumption with network communication overhead:

$$E_{system,total} = \sum_{i=1}^{N} E_{total,i} + E_{comm,total} \tag{10}$$

where $E_{comm,total}$ aggregates all communication-related energy consumption across the network topology.

4.4 Application to Bartering-Based Protocol

The **bartering-based protocol** described in previous sections aims to foster cooperation between peers by allowing them to exchange storage resources without relying on centralized entities or cryptocurrencies. In this subsection, we discuss on some hints into how we may foresee the power modeling onto the bartering protocol, feature that may increase or reduce power consumption, focusing on three key factors:

1. **Frequency of storage proof requests.**
2. **Storage replication overhead.**
3. **Node churn and recovery mechanisms.**

1. Frequency of Storage Proof Requests. One of the protocol's core features is the ability for a node to periodically audit whether its data is still retained by other peers. This mechanism, while integral to trust-building, can incur significant overhead in terms of both bandwidth and CPU cycles.

2. Storage Replication Overhead. The bartering mechanism ensures that a node's data is stored across multiple peers (often K replicas). While replication is central to data durability and availability, it also has direct power implications.

3. Node Churn and Recovery Mechanisms. Node churn peers frequently joining and leaving is inherent in distributed edge environments. The bartering protocol incorporates adaptive strategies to handle churn, but these strategies also affect power usage:

Ultimately, the interplay of churn and the bartering mechanism places continual evolutionary pressure on node behaviors: those that stay online longer can maintain higher scores and thus gain more stable storage services, whereas ephemeral nodes risk frequent re-verification or lower trust.

5 Architecture and Implementation

In this chapter, we detail the practical implementation of our **bartering-based storage protocol**, including how we introduce **energy awareness** and **green coding** practices into the code. We build upon the conceptual design to illustrate how real-world constraints such as node churn, realistic file I/O workloads, and efficient concurrency are handled in practice. Finally, we describe how we integrated various techniques (concurrency optimizations, ephemeral connectivity, and dynamic bartering) to reduce **energy consumption** while maintaining robust performance.

5.1 Overall System Architecture

Figure 14 provides a high-level depiction of the system's global architecture, which includes a monitoring component (custom analytics) that gathers CPU usage, memory, network throughput, and concurrency metrics. The main bartering logic runs on a distributed set of nodes, each potentially at the edge of the network. We incorporate `FIO` to simulate real read/write operations. Each node in the system can serve as a client, provider, or both, depending on the bartering context.

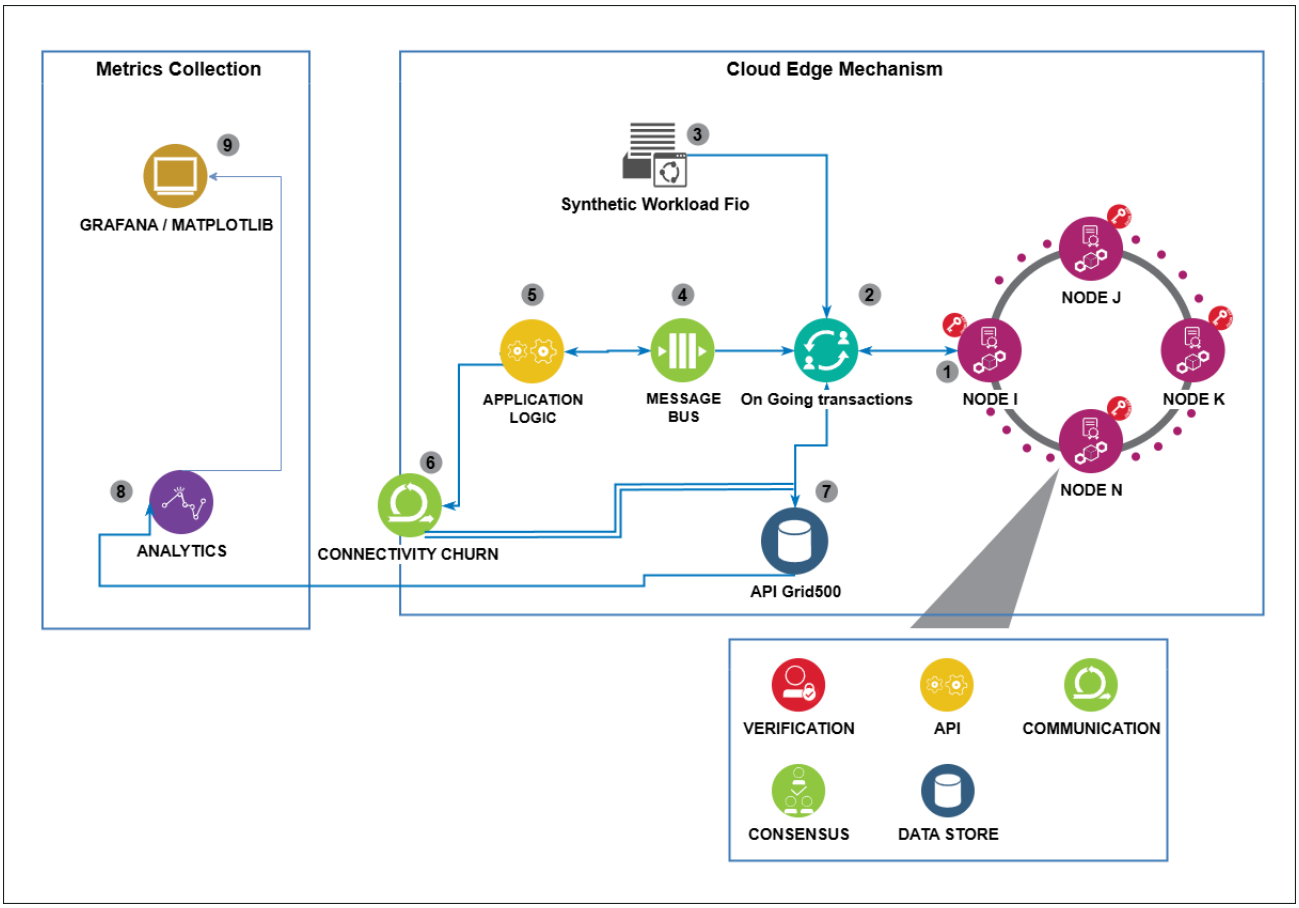


Figure 14: High-level overview of the system's global architecture, showing bartering logic, `fio` workload generator, and metric collection.

Nodes interact through a BitTorrent-inspired content discovery and exchange system (the **bitswap protocol**) for bootstrapping, bartering, and storage requests. A central (but not controlling) **bootstrap node** helps new participants discover existing peers.

5.2 Code Architecture and Main Components

Figure 15 illustrates the primary modules that compose our bartering protocol:

- **Main Node & Connectivity Churn**: Orchestrates the entire node lifecycle and enforces uptime/downtime based on node profiles (Benefactor, Peer, or Peeper).
- **Bartering Engine**: Implements ratio negotiation, local scoring, and acceptance strategies.
- **Storage Tester** (Proof-of-Storage checks): Periodically validates that nodes continue to store the data they promised, leveraging `ipfs cat` or hashing as a basic proof mechanism.
- **Node Manager / Peers Registry**: Maintains the discovered peers, their statuses, and helps route requests.

- **IPFS Integration**: Provides content-addressed storage and retrieval. Uploading, pinning, and verifying data is handled via IPFS commands.

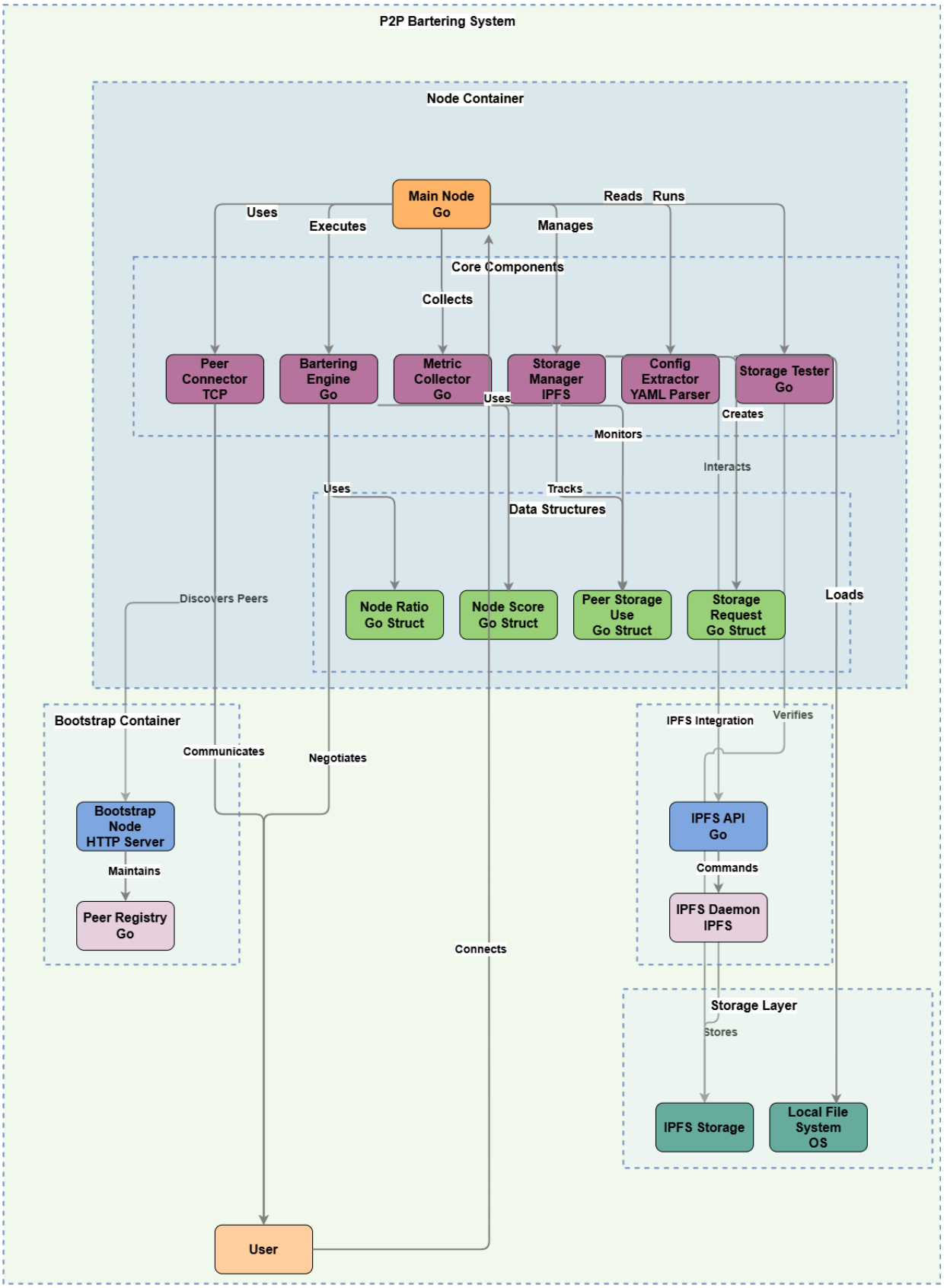


Figure 15: High-level Implementation architecture for the P2P bartering system

We use Go (Golang) for its concurrency model (`goroutines` and channels) and relatively low overhead garbage collection, aligning with **green coding** recommendations for compiled languages [39], [51], [52]. In particular, Go's concurrency is leveraged to handle ephemeral connectivity, while avoiding **excessive object allocations** that might increase CPU usage and thus energy consumption [23]

Detailed Component Flow

Figure 16 shows the internal flow of data among modules:

1. **Config Extractor**: Reads `config.yaml`, setting the node's total storage capacity, bartering parameters (initial score, ratio increase rate), and churn model.
2. **Data Structures**: Maintains node-level arrays of `NodeScore`, `NodeRatio`, and `PeerStorageUse`.
3. **File System Watcher (FsWatcher)**: Detects new or modified files locally and triggers a **Store** operation.
4. **Bartering Engine**: Initiates ratio negotiations, updates local and remote ratios, and calls **Barter messages** among peers.
5. **Storage Tester (Proof Generator)**: Periodically issues `TesRq` messages to confirm data is still stored, applying **score adjustments** if tests fail.
6. **Node Churn**: Enforces on/off scheduling for each node to simulate ephemeral connectivity (Benefactor / Peer / Peeper).
7. **Analytics (optional)**: Exposes metrics via a simple HTTP endpoint or logs them for offline analysis.

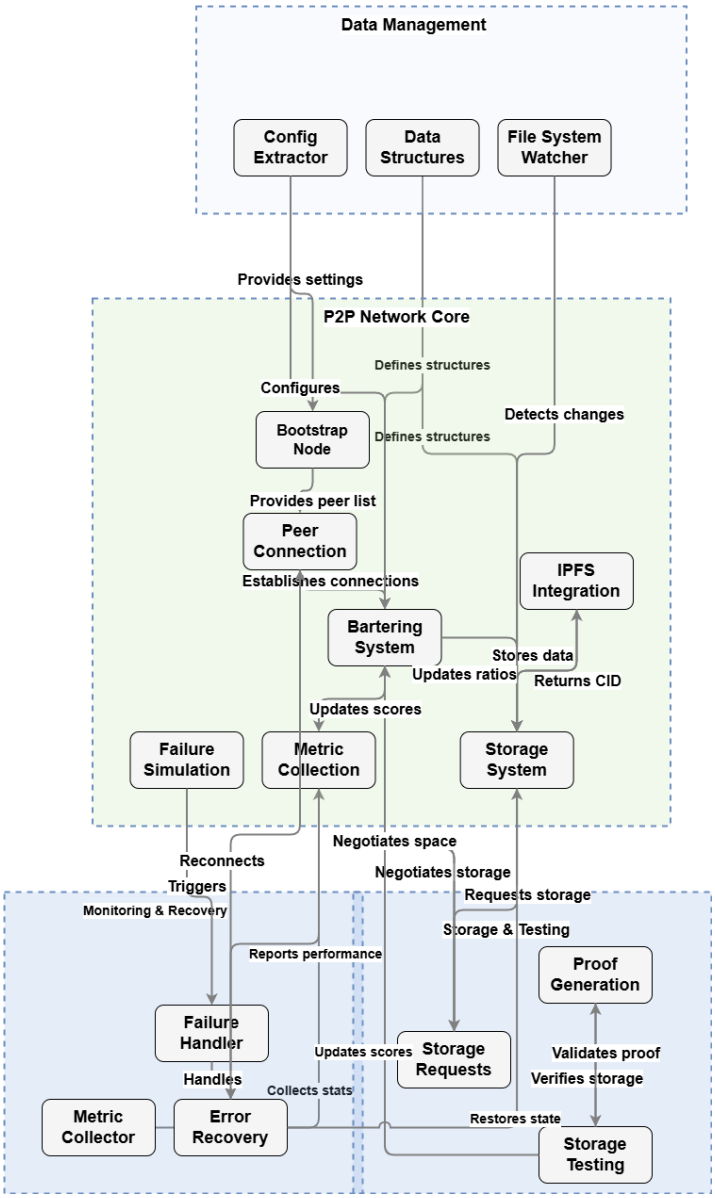


Figure 16: Flow diagram illustrating how configuration, data structures, watchers, bartering logic, and proof generation interact.

Each of these components runs concurrently. We rely on `sync.WaitGroup` to manage graceful node startup and shutdown, and `mutexes` only where absolutely necessary to reduce lock contention.

5.3 Green Coding Techniques and Energy Optimizations

We adopt most best practices from the **Green Software Engineering** (GSE) domain [26], [46]:

1. **Go concurrency model**: We keep the number of `goroutines` bounded, preventing memory spikes or

CPU overhead. Each node uses a channel-based concurrency approach rather than spawning unbounded routines.

2. **Adaptive Timers for Storage Proofs**: The frequency of sending `Storage Proof Requests` scales with node reliability (score). Reliable peers are checked less frequently, lowering CPU/wake-up overhead [52].

3. **Lazy Initialization**: We only load bartering modules (ratio negotiation) when peers are actually discovered, deferring resource usage to the moment it is strictly needed.

4. **Churn-aware load balancing**: The `NodeProfile` helps the system decide how many replicas or how frequently to re-check ephemeral nodes, thus avoiding unproductive data migrations.

Implementation Highlights

We use Go to compile to efficient binaries with minimal runtime overhead, in line with **green coding** suggestions for compiled languages [23], [46]. Moreover, the `goroutine` concurrency model is well-suited for ephemeral networks. Nodes spawn a bounded number of concurrent TCP listeners to handle bartering messages, `StorageRequests`, and `TestRequests`. We rely on `sync.WaitGroup` to ensure a graceful node shutdown once all goroutines complete. The `PeriodicTests` module adjusts test frequency based on node reliability. High-score nodes are tested less often, saving CPU cycles and thus energy [26]. We store `NodeScore` and `NodeRatio` in arrays for $O(1)$ indexing. We also avoid heavy synchronization by making short critical sections, thus reducing CPU context switching [52].

Sample Code Snippet

Below is an excerpt illustrating how we apply concurrency with minimal overhead in the `PeriodicTests` function. It runs a test cycle every `testingPeriod` seconds, contacting peers that hold a node's data:

```
func PeriodicTests(fulfilledRequests *[]FulfilledRequest, scores []NodeScore,
    timerTimeoutSec float64, port string, testingPeriod float64,
    DecreasingBehavior []ScoreVariationScenario,
    IncreasingBehavior []ScoreVariationScenario,
    bytesAtPeers []PeerStorageUse,
    scoreDecreaseRefStoReq float64) {

    for {
        time.Sleep(time.Duration(testingPeriod) * time.Second)
        for _, fReq := range *fulfilledRequests {
            go func(req FulfilledRequest) {
                ok := ContactPeerForTest(req.CID, req.Peer, scores, timerTimeoutSec,
                                         port, DecreasingBehavior, IncreasingBehavior)
                if !ok {
                    // re-request storage from other peers
                }
            }(fReq)
        }
```

```
    }
}
```

We spawn a new goroutine for each `FulfilledRequest`, but we limit concurrency using channels or WaitGroups if the number of requests grows large.

Summary of Energy-Aware Choices

Table 2 summarizes the main energy-aware or **green coding** decisions we integrated:

Table 2: Key Energy-Aware Coding Decisions

Technique	**Rationale for Energy Savings**
Bounded `goroutine` concurrency	Avoids memory overhead and CPU context switching from too many parallel threads.
Adaptive test frequency	Decreases CPU usage for stable, high-score peers, reducing repeated hashing or network overhead.
Lazy initialization	Only load bartering or IPFS modules when necessary, saving idle energy.
Churn-based re-check intervals	Minimizes wasted cycles on short-lived nodes that appear and vanish quickly.
Use of compiled language (Go)	Faster execution, less overhead than purely interpreted languages, reducing CPU time.

By adhering to these guidelines from [52] , [51] , [23] and [46], we aim to ensure that our bartering system not only manages **network-level churn and resource constraints** but does so with minimal energy overhead.

5.4 Some Implementation modules

The practical implementation of our bartering protocol translates the theoretical concepts into a concrete system architecture with specific message handling mechanisms across distributed hosts. Figure 17 illustrates the high-level communication flow between remote peers in our implementation.

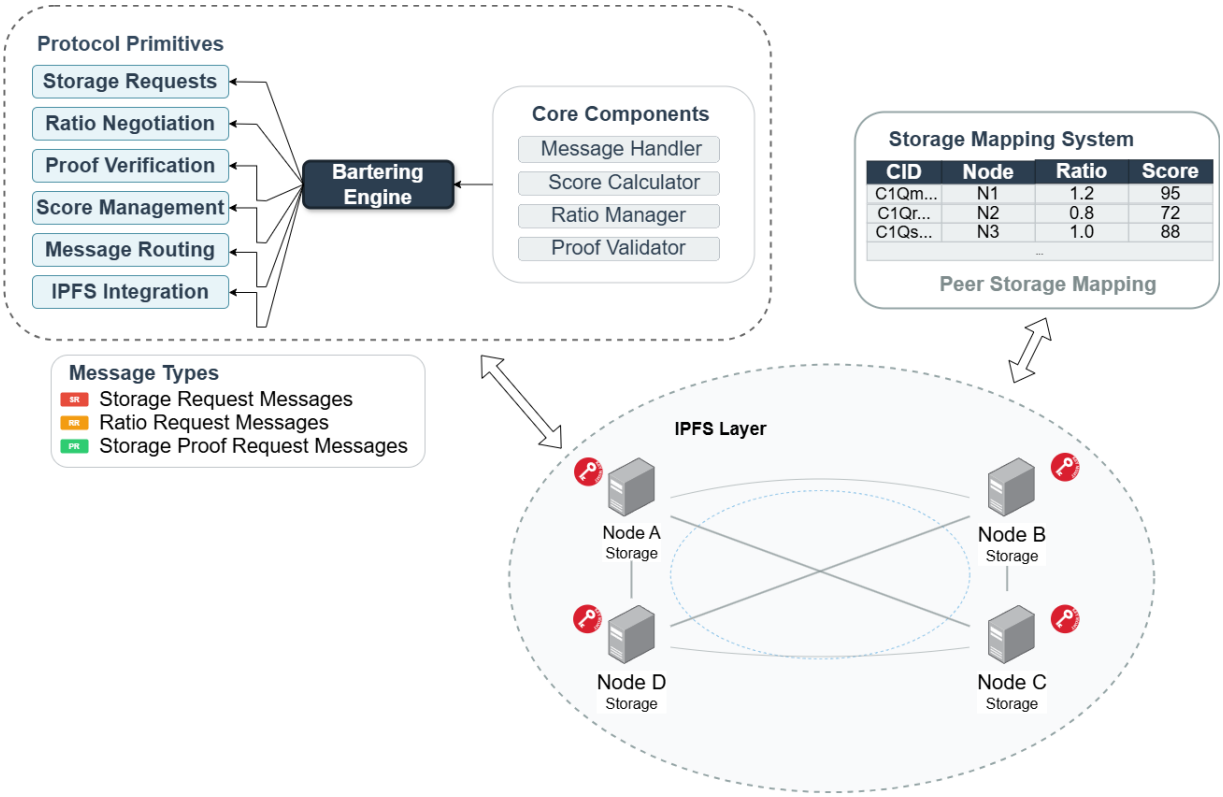


Figure 17: High-level view of bartering protocol among remote hosts

Our implementation uses three primary communication structures that handle different aspects of the peer-to-peer exchange:

- **Storage Request Handler:** This component processes incoming and outgoing storage allocation requests, manages the IPFS Content Identifier (CID) mapping, and coordinates with the local resource management system.

```
func HandleStorageRequest(bufferString string, conn net.Conn, bytesForPeers []datastructures.PeerStorageUse,
storedForPeers *[]datastructures.FulfilledRequest) {
peer := conn.RemoteAddr().(*net.TCPAddr).IP.String()
CID := bufferString[5:51]
fileSize := bufferString[51:]
fileSize = strings.Split(fileSize, "\n")[0]
fileSizeFloat, err := strconv.ParseFloat(fileSize, 64)
utils.ErrorHandler(err)
request := datastructures.StorageRequest{FileSize: fileSizeFloat, CID: CID}

if CheckRqValidity(request) {
    api_ipfs.PinToIPFS(CID)
    messageToPeer = "OK\n"
    updateBytesForPeers(bytesForPeers, peer, fileSizeFloat)
    updateFulfilledRequests(CID, peer, storedForPeers)
} else {
    messageToPeer = "KO\n"
}
io.WriteString(conn, messageToPeer)
}
func StoreKCopiesOnNetwork(peerScores []datastructures.NodeScore, K int, storageRequest datastructures.StorageRequest,
port string, bytesAtPeers []datastructures.PeerStorageUse,
fulfilledRequests *[]datastructures.FulfilledRequest, scoreDecreaseRefStoReq float64) int {
```

```
okRqs := 0
usedPeers := make(map[string]bool)
tries := 0
for tries < 3 {
    peersToRequest, err := ElectStorageNodes(peerScores, K, usedPeers)
    if err != nil {
        return 0
    }

    for _, peer := range peersToRequest {
        if usedPeers[peer] {
            continue
        }
        response := RequestStorageFromPeer(peer, storageRequest, port, bytesAtPeers,
                                           peerScores, fulfilledRequests, scoreDecreaseRefStoReq)
        if response == "OK\n" {
            okRqs++
            usedPeers[peer] = true
            if okRqs == K {
                return okRqs
            }
        }
    }
    tries++
}
return okRqs
}
```

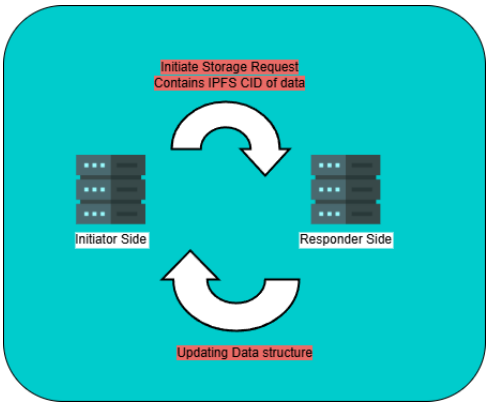


Figure 18: implementation architecture of the Storage Request Handler

- **Ratio Negotiation Module:** This module implements the bilateral exchange negotiation logic using asynchronous communication channels. Each ratio proposal is processed through a state machine that tracks negotiation status and applies the node's decision policy based on current score values and resource availability.

```
func InitiateBarter(peer string, ratios []datastructures.NodeRatio, ratioIncreaseRate float64,
port string, msgCounter *int) error {
currentRatio, err := FindNodeRatio(ratios, peer)
if err != nil {
return errors.New(err.Error())
}
newRatio := calculateNewRatio(currentRatio, ratioIncreaseRate)
barterMessage := "BarRq" + strconv.FormatFloat(newRatio, 'f', -1, 64)
response := contactNodeForBarter(peer, barterMessage, port, msgCounter)

if response == "OK\n" {
    updatePeerRatio(ratios, peer, newRatio)
} else {
```

```
    ratio, err := strconv.ParseFloat(response[:len(response)-1], 64)
    utils.ErrorHandler(err)
    updatePeerRatio(ratios, peer, ratio)
}
return nil
}
func shouldRatioBeAccepted(ratio float64, peer string, storageSpace float64,
bytesAtPeers []datastructures.PeerStorageUse,
scores []datastructures.NodeScore,
factorAcceptableRatio float64) bool {
currentStorage, err := findPeerStorageUse(peer, bytesAtPeers)
utils.ErrorHandler(err)
if currentStorage.StorageAtNode == 0.0 {
return true
}
return (isRatioTolerableGivenStorageSpace(peer, ratio, storageSpace, bytesAtPeers) &&
       (ratio < calculateMaxAcceptableRatio(peer, scores, storageSpace,
                                    bytesAtPeers, factorAcceptableRatio)))
}
func calculateMaxAcceptableRatio(peer string, scores []datastructures.NodeScore,
storageSpace float64, bytesAtPeers []datastructures.PeerStorageUse,
factorAcceptableRatio float64) float64 {
peerScore, err := findPeerScore(peer, scores)
utils.ErrorHandler(err)
ratio := factorAcceptableRatio * peerScore.Score
if !isRatioTolerableGivenStorageSpace(peer, ratio, storageSpace, bytesAtPeers) {
    storageUsed, err := findPeerStorageUse(peer, bytesAtPeers)
    utils.ErrorHandler(err)
    ratio = storageSpace / storageUsed.StorageAtNode
}
return ratio
}
```

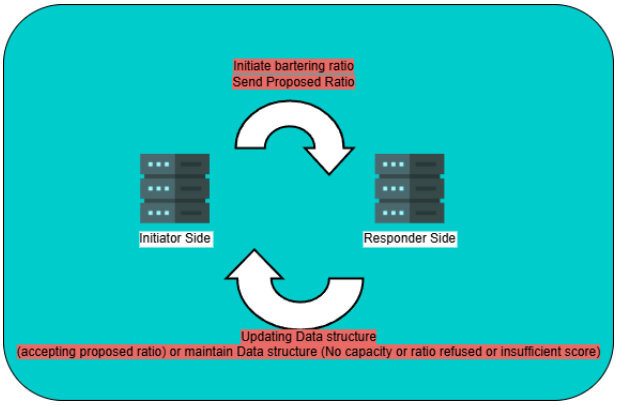


Figure 19: implementation architecture of the Ratio Negotiation Module

- **Storage Verification module:** This component schedules and executes periodic storage proof challenges, processes responses, and triggers re-replication when necessary. The implementation uses lightweight timers with dynamic intervals to balance verification thoroughness with energy efficiency.

```
func PeriodicTests(fulfilledRequests *[]datastructures.FulfilledRequest, scores []datastructures.NodeScore,
timerTimeoutSec float64, port string, testingPeriod float64,
DecreasingBehavior []datastructures.ScoreVariationScenario,
IncreasingBehavior []datastructures.ScoreVariationScenario,
bytesAtPeers []datastructures.PeerStorageUse, scoreDecreaseRefStoReq float64) {
for {
time.Sleep(time.Duration(testingPeriod) * time.Second)
for _, fulfilledRequest := range *fulfilledRequests {
testResult := ContactPeerForTest(fulfilledRequest.CID, fulfilledRequest.Peer, scores,
timerTimeoutSec, port, DecreasingBehavior, IncreasingBehavior)
```

```
if !testResult {
// Could not confirm storage; need to request storage from other node
stoReq := datastructures.StorageRequest{CID: fulfilledRequest.CID,
FileSize: fulfilledRequest.FileSize}
peersToRq := storagerequests.RemovePeerFromPeers(scores, fulfilledRequest.Peer)
storagerequests.StoreKCopiesOnNetwork(peersToRq, 1, stoReq, port, bytesAtPeers,
fulfilledRequests, scoreDecreaseRefStoReq)
}
}
}
}
func ContactPeerForTest(CID string, peer string, scores []datastructures.NodeScore,
timerTimeoutSec float64, port string,
DecreasingBehavior []datastructures.ScoreVariationScenario,
IncreasingBehavior []datastructures.ScoreVariationScenario) bool {
conn, err := net.Dial("tcp", peer+":"+port)
if err != nil {
decreaseScore(peer, "failedTestTimeout", scores, DecreasingBehavior)
return false
}
defer conn.Close()
ctx, cancel := context.WithCancel(context.Background())
defer cancel()
message := "TesRq" + CID
io.WriteString(conn, message)

responseChannel := make(chan string)
var wg sync.WaitGroup
wg.Add(1)
go handleResponse(ctx, &wg, responseChannel, conn)

timer := time.NewTimer(time.Duration(timerTimeoutSec) * time.Second)
defer timer.Stop()

select {
case <-timer.C:
    decreaseScore(peer, "failedTestTimeout", scores, DecreasingBehavior)
    cancel()
    return false
case response := <-responseChannel:
    if checkAnswer(response, CID) {
        increaseScore(peer, "passedTest", scores, IncreasingBehavior)
        return true
    } else {
        decreaseScore(peer, "failedTestWrongAns", scores, DecreasingBehavior)
        return false
    }
}
}
func computeExpectedAnswer(CID string) []byte {
CID = CID[:46]
contentString := api_ipfs.CatIPFS(CID)
contentBytes := []byte(contentString)
hasher := sha256.New()
hasher.Write(contentBytes)
return hasher.Sum(nil)
}
```

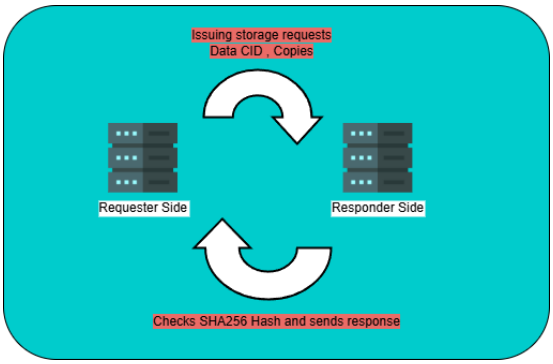


Figure 20: implementation architecture of the Storage Verification Engine

The implementation connects these components through an event-driven architecture that minimizes blocking operations. When a peer initiates storage placement, the system constructs appropriate request messages, distributes them to selected peers based on the current score registry, and monitors responses. Successful storage placements are recorded in a persistent local database, which then feeds into the verification scheduler.

If a storage verification fails, the implementation automatically triggers the re-replication subsystem, which consults the node registry to identify alternative storage locations with favorable scores.This self-healing capability ensures the system maintains the desired replication factor even as nodes join and leave the network or temporarily fail to respond to verification challenges.

5.5 Mimicking Real Cloud File System workload (Churn Model)

In distributed storage environments, especially those at the edge, the dynamic nature of node participation can be as significant a factor as resource constraints or network topologies. Hence, we proceeded to integrate a realistic **churn model** , thereby capturing ephemeral connectivity patterns as well as representative read/write workloads in a single experimental framework.

This section explains how we implement the node classification (Benefactor, Peer, Peeper) and how we enforce intermittent connectivity (uptime vs. downtime), inspired by prior art such as [2], [4], [6] but extended for our protocol scenario. We then reserve space at the end for describing how `fio` is invoked to simulate the system's read/write operations.

5.5.1 Churn Model: Node Profile Assignment

Churn is defined as the continuous process by which nodes join, remain for variable periods, and eventually leave the network affects every layer of system operation. At its core, churn challenges the fundamental assumptions of stable resource availability and long-lived connections that designers of traditional, more centralized systems might take for granted.

A well-calibrated churn model simulates the temporal and spatial fluctuations in node availability, reflecting underlying user behaviors, external events, and technological shifts. For instance, consider a network of edge devices, such as IoT sensors, mobile phones, and temporary computing resources provided by passing vehicles. At any given moment, a certain fraction of nodes may drop offline due to energy-saving modes, network

coverage gaps, or hardware failures. Conversely, new nodes might appear sporadically, offering unused storage capacity or demanding immediate replication of fresh data. By incorporating this complexity, the churn model allows the bartering protocol to be continuously tested and refined against conditions that approach real-world unpredictability.

The churn model employed here goes beyond a simplistic "node up or node down" binary representation. Instead, it aims to reflect an entire distribution of node session lengths, re-connection patterns, and activity peaks, each dimension grounded in statistical data and behavioral studies. A network might, for example, exhibit daily periodicities corresponding to human work cycles: a surge in connectivity and requests during office hours followed by lulls at night. External events such as firmware updates, power outages, or sudden interest in a piece of content can cause temporary imbalances in how nodes interact. The churn model can encode these patterns, enabling the system to anticipate and respond to shifts in workload intensity, storage demand, and peer reliability.

Moreover, churn is driven by economic or incentive-based factors. In networks where bandwidth or storage is metered or where energy costs vary over time, nodes may strategically appear or disappear. By integrating these parameters, the churn model ensures that the bartering protocol is evaluated in environments where nodes not only come and go but do so with motivations that can skew the ratio of available capacity to requested storage, affecting both the short-term and long-term equilibrium of resource distribution.

Following the approach in [6], we classify each node into one of three profiles:

- **Benefactor ($\approx 20\%$ of nodes):** High availability, longer session times
- **Peer ($\approx 40\%$ of nodes):** Moderate availability
- **Peeper (the remainder):** Short sessions, highly transient

We let N be the total number of nodes we have acquired. We then compute:

$$\begin{aligned} \textbf{num\_benefactors} &= 0.20 \times N, \\ \textbf{num\_peers} &= 0.40 \times N, \\ \textbf{num\_peepers} &= N - \big(\textbf{num\_benefactors} + \textbf{num\_peers}\big). \end{aligned}$$

Algorithm 5 shows a pseudo-code view of the Python snippet that (1) parses node names to determine clusters, (2) assigns a portion of them to Benefactors and Peers, then (3) randomizes the remainder as Peepers. Finally, each node is labeled in a dictionary *host_profiles* keyed by node hostname.

Algorithm 5 Node Profile Assignment for Churn

Require: `availableHosts`: List of node hostnames
Require: `nodesNeeded`: Total number of nodes (N)

Compute profile counts:
  $numBene \leftarrow \lfloor 0.20 \times N \rfloor$
  $numPeer \leftarrow \lfloor 0.40 \times N \rfloor$
  $numPeep \leftarrow N - (numBene + numPeer)$
Group nodes by cluster:
  `clusters` ← empty map
for all `node` in `availableHosts` **do**
  `clusterName` ← parse prefix of `node`
  `clusters`[clusterName].append(`node`)
end for
function ASSIGNNODESTOPROFILE(`profileList`, `maxNeeded`)
  for all (`cName`, `cNodes`) in `clusters` **do**
    while `cNodes` not empty **and** length(`profileList`) < `maxNeeded` **do**
      `profileList`.append(`cNodes`.popFirst())
    end while
    if length(`profileList`) ≥ `maxNeeded` **then**
      break
    end if
  end for
end function
Initialize empty lists: `BeneList`, `PeerList`, `PeepList`
ASSIGNNODESTOPROFILE(BeneList, numBene)
ASSIGNNODESTOPROFILE(PeerList, numPeer)
Flatten remaining clusters into `remaining`
shuffle(`remaining`)
`PeepList`.extend(first *numPeep* from `remaining`)
Create a final map *hostProfiles*:
for all `node` in `BeneList` **do**
  `hostProfiles`[node] ← "benefactor"
end for
for all `node` in `PeerList` **do**
  `hostProfiles`[node] ← "peer"
end for
for all `node` in `PeepList` **do**
  `hostProfiles`[node] ← "peeper"
end for
return `hostProfiles`

5.5.2 Intermittent Connectivity (Uptime/Downtime)

Once each node has a profile, we invoke a small shell script on each host that enforces the "on" (uptime) vs. "off" (downtime) periods. Table 3 lists an example mapping from profile to ratio of total time spent online. Algorithm 6 shows a pseudo-code version of that script.

Table 3: Node Profiles and Their Uptime/Downtime Ratios

Profile	Uptime Fraction	Downtime Fraction
Benefactor	0.75	0.25
Peer	0.50	0.50
Peeper	0.30	0.70

Algorithm 6 Churn Simulation

Require: `NODE_PROFILE` in {`"benefactor"`, `"peer"`, `"peeper"`}
Require: `RESERVATION_DURATION_SEC`: total test time
Require: `BOOTSTRAP_IP`, `LABEL`: arguments to bartering binary
Compute baseSession ← `RESERVATION_DURATION_SEC`
Set $(Uptime, Downtime)$ from Table 3 based on `NODE_PROFILE`
Launch infinite loop:
while true **do**
    start bartering process (in background) ▷ `./bartering $BOOTSTRAP_IP`
    sleep for $Uptime$ seconds
    kill the bartering process
    sleep for $Downtime$ seconds
end while

The bartering protocol sees each "offline" node vanish from the network, prompting re-replication or other failure responses. Once "online" again, the node rejoins. In tandem with the distribution from Algorithm 5, this yields realistic and heterogeneous churn.

5.5.3 Connectivity Patterns and Their Consequences

One crucial aspect of the churn model is not just who is online, but how they connect and at what frequency. Benefactors may maintain stable TCP/IP links with many peers, fostering a stable backbone. Peers might connect only when triggered by user activity, leading to moderate fluctuations. Peepers, on the other hand, might exhibit "drive-by" connectivity brief encounters with few established links, pushing the protocol to act swiftly or risk missing an opportunity for replication. The churn model simulates these different connectivity patterns, potentially tying them to time-of-day, location, or network conditions.

Because bartering strategies depend on historical trust and reciprocal behavior, a node that rarely connects but appears randomly may not accumulate high trust scores easily, as it rarely responds to storage proofs. Benefactors, conversely, might build long histories of reliable interactions, making them prime candidates for stable, low-overhead proof testing. The churn model can introduce phases where nodes switch roles (a peer might upgrade its hardware and become more Benefactor-like over time), challenging the protocol to adapt to evolving node profiles.

6 Experimental Setup and Methodology

In this work, we rely on the Grid'5000 [17] infrastructure, a large-scale testbed dedicated to research in distributed computing. The Grid'5000 global view is shown in Figure 21, providing an overview of its geographically distributed clusters.

6.1 The Grid'5000 Infrastructure and Taurus Cluster

The **Grid'5000** infrastructure is distributed across multiple sites in France, offering significant computational resources for experimentation. Each site consists of several clusters equipped with high-speed interconnects, ensuring reliable and scalable experiments.Our experiments focus mostly on the **Taurus cluster**, located at the Lyon site, which is particularly well-suited for testing large-scale distributed systems.We precise that we also executed runs on similar non taurus node during the experiments. Table 4 provides the specification layout.

Table 4: Technical Specifications of the Taurus Cluster

Specification	**Details**
Nodes	16 homogeneous nodes
CPU	Two *Intel Xeon E5-2630* CPUs per node 6 physical cores per CPU (12 threads total)
Memory	32 GiB of RAM per node
Cache	3 cache levels: - 32 KiB L1 - 256 KiB L2 - 15 MiB L3
Interconnection	10 Gbit/s Ethernet links to a central switch
Power Measurement	Hardware wattmeter per node
Sampling Rate	1 Hz
Accuracy	0.125 W

Each Taurus node is equipped with a hardware wattmeter, allowing direct measurement of power consumption via the **Grid'5000 API** at a **1 Hz** sampling rate and **0.125 W** accuracy.To ensure fair and reproducible performance, we reserved the entire Taurus cluster during our experiments, mitigating potential interference from other users. Figure 22 illustrates the Ethernet topology of the Lyon site, highlighting the interconnected clusters and their capacities.

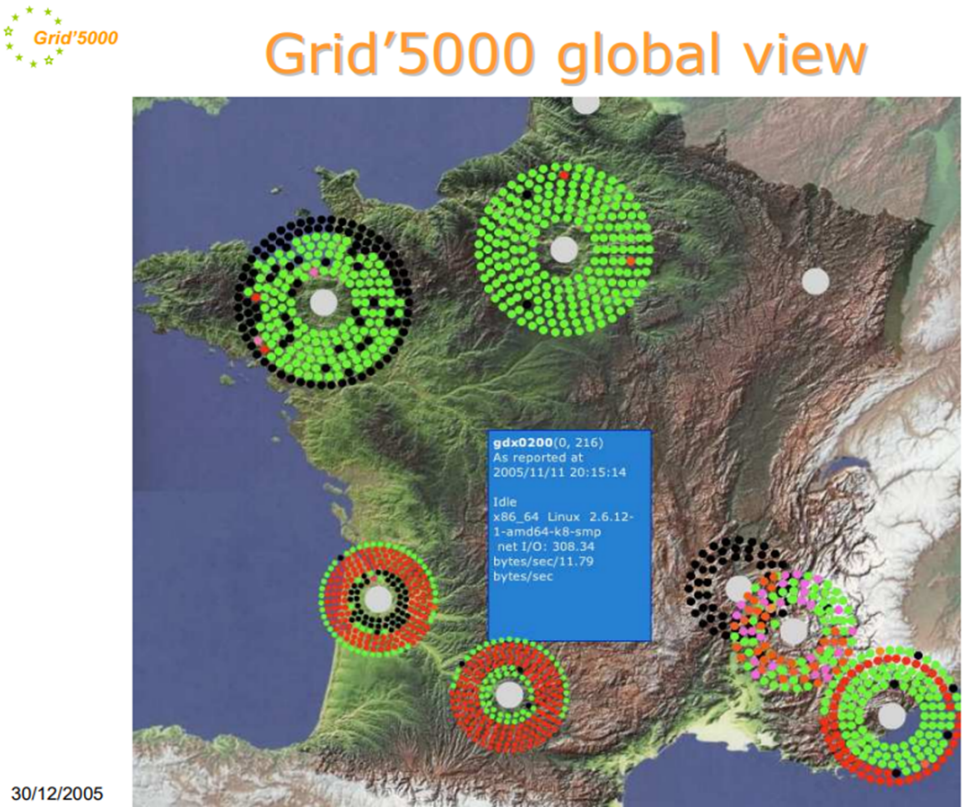


Figure 21: Grid'5000 Global view

6.2 Deployment Procedure

Deploying a private IPFS network across the Taurus cluster requires automating node configuration, binary installations, and monitoring setup. To achieve this, we developed a structured deployment procedure using **Ansible**.

Overview: The deployment pipeline includes:

1. Configuration of network specifications in a **JSON file**.
2. Validation and discovery of available nodes using a **Python script**.
3. Execution of an **Ansible playbook** to configure the nodes.
4. Setup of monitoring tools (**Prometheus** and **Grafana**).
5. Execution of the **bartering binary application**.

Figure 23 summarizes the deployment procedure in detail.

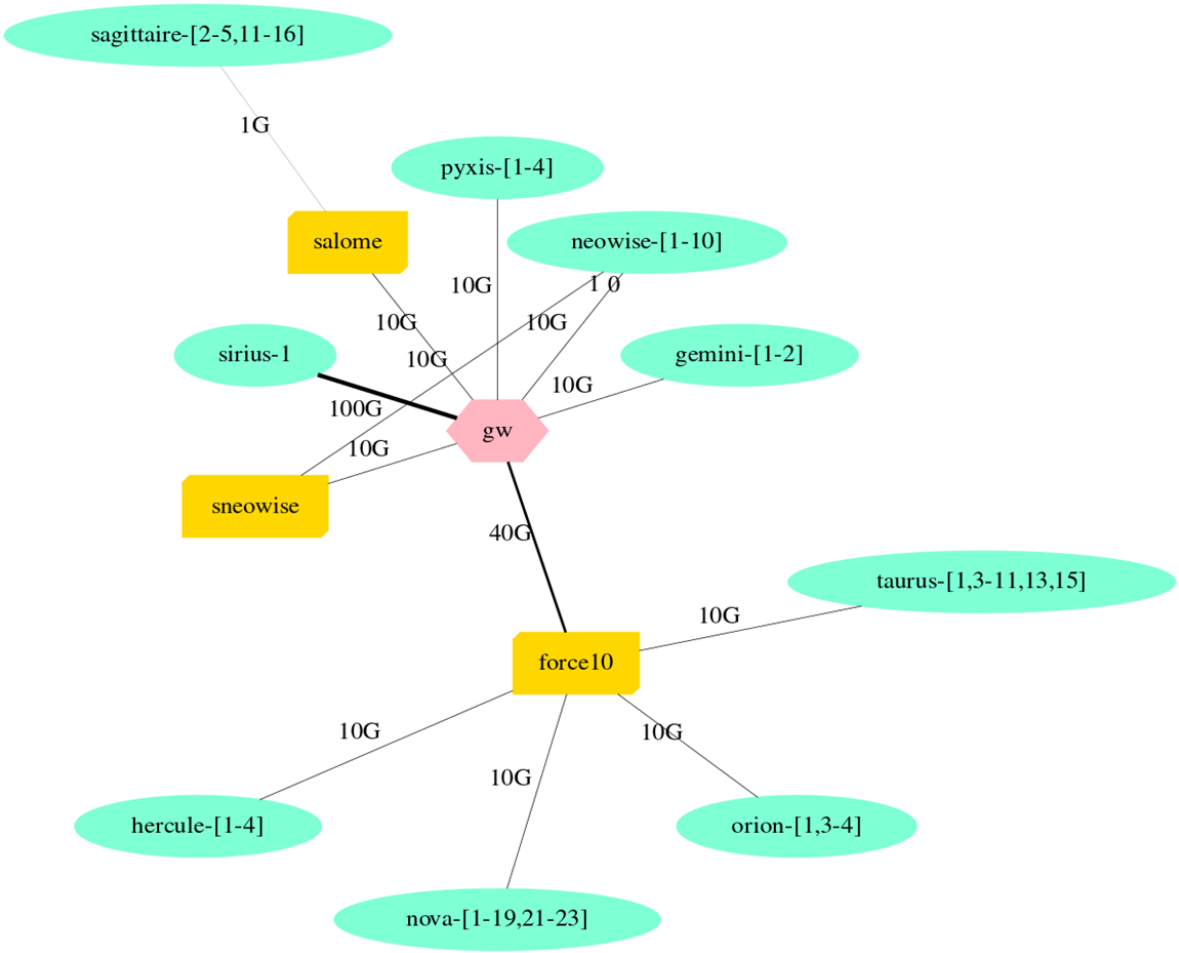


Figure 22: Ethernet network topology of the Lyon site, showcasing the Taurus cluster

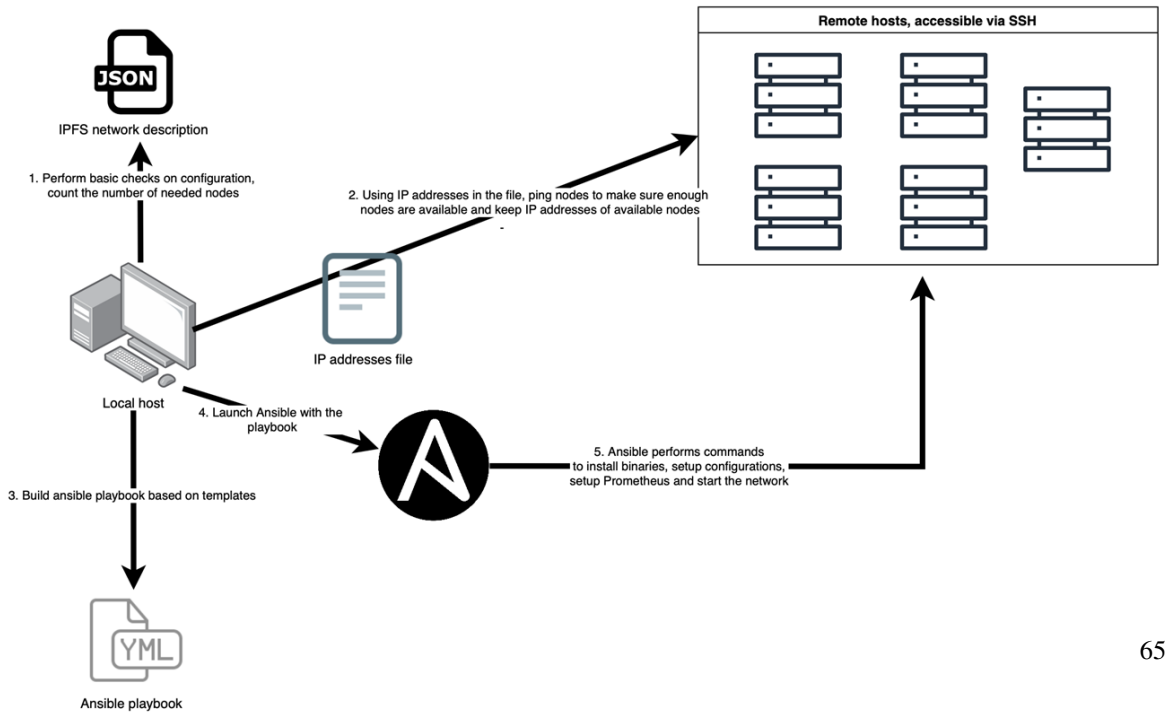


Figure 23: Deployment Procedure for the IPFS private network

Step 1: Defining the IPFS Network Configuration

The deployment process starts with defining the private IPFS network configuration in a JSON file. An example configuration is provided below:

```
{
    "Credentials": "username",
    "IPFS_network": {
        "Nodes": 6,
        "GCPeriod": "15m",
        "MaxStorage": "15kb"
    },
    "IPFS_Clusters": {
        "1": { "Nodes": 3 },
        "2": { "Nodes": 2 }
    }
}
```

Table 5 summarizes the overall steps involved in configuring, validating, and managing our experimental setup on **Grid'5000**, transforming the bullet-point descriptions into a structured view.

Table 5: Deployment Procedure Overview

Step	Description
1. JSON Configuration	**Define IPFS network parameters** in a JSON file: • **Nodes:** Total number of IPFS nodes in the network. • **GCPeriod:** Garbage collection period. • **MaxStorage:** Storage quota per node. • **IPFS_Clusters:** Optional clusters with node counts.
2. Node Validation and Discovery	The Python script *inf_builder.py* validates the JSON config: • **Ping nodes** listed in *ip_@_txt* to check availability. • Auto-generateS the **hosts.ini** for Ansible. • Verify enough nodes are online for the requested configuration.
3. Ansible Playbook Execution	Build and run an **Ansible playbook** based on the JSON: • Install required dependencies (Go, IPFS binaries). • Initialize and configure IPFS nodes. *Example command:* `ansible-playbook deploy_ipfs.yml -i hosts.ini --ask-pass`
4. Monitoring Setup	**Grafana** provides interactive visual dashboards for analysis.
5. Bartering Application	Launch the bartering binary to simulate real storage interactions: • Issue storage requests among peers. • Periodically check proofs of storage. • Dynamically adjust decisions based on node churn and behavior.
Network Cleanup	Run the **network_killer.yaml** playbook to shut down the network and remove leftover configurations from the nodes at experiment end.

Once the bartering application is up and running, it continuously processes storage requests, proof-of-storage checks, and re-replication operations whenever churn or failures occur. At the conclusion of each experiment, the *network_killer.yaml* playbook ensures a complete teardown of the deployed environment.

6.3 Network Condition Simulation

To evaluate protocol performance under varying network qualities, we systematically introduced three distinct network conditions:

- **Normal conditions:** Unmodified network environment with the native 10 Gbit/s Ethernet interconnections

representing ideal operational scenarios.

- **Degraded conditions:** Simulated using the linux traffic control (`tc`) tool with added latency (50-100ms), moderate packet loss (2-5%). This represents common edge deployment scenarios with suboptimal connectivity.
- **Congested conditions:** Implemented with more severe network impairments including higher latency on linux traffic control (150-250ms), significant packet loss (8-12%).

These conditions were implemented using standard linux networking utilities. For example, to create degraded conditions on a node, we executed:

```
# Add 75ms latency with 20ms variation
tc qdisc add dev eth0 root netem delay 75ms 20ms

# Add 3% packet loss
tc qdisc change dev eth0 root netem delay 75ms 20ms loss 3%
```

The `netem` module allows precise emulation of network properties such as variable delay, loss, duplication, and reordering. We precise that these commands are executed only at the start of the simulation scenario and remained constant throughout the respective test runs.This capability is particularly valuable for reproducing the conditions commonly encountered in edge computing environments, where network quality can vary significantly

6.4 Integrating `fio`: Synthetic File I/O Workloads

To evaluate our bartering protocol under realistic distributed storage conditions, we integrated the Flexible I/O tester (`fio`), a versatile benchmarking tool designed to generate configurable I/O workloads. This integration allows us to simulate typical cloud storage operations while measuring power consumption and protocol behavior under varying conditions.

6.4.1 Flexible I/O Tester Overview

The Flexible I/O (FIO)[2] tester provides fine-grained control over storage workload characteristics, allowing us to generate precise read/write patterns (sequential, random, or mixed) under various simulated network conditions. Our experimental methodology employs FIO to create representative workloads that closely mimic real-world distributed storage usage patterns.

For our experiments, we configured FIO with a range of parameters to simulate realistic edge storage scenarios, similar to what we saw in [47]. Table 6 summarizes the key configuration parameters used in our testing.

[2]Technical specification at `https://www.grid5000.fr/mediawiki/index.php/Lyon:Hardware#Taurus`

Table 6: FIO Workload Configuration Parameters c

Parameter	**Configuration**
File sizes	1-100MB (primarily 10, 20, 30, 40, 50MB)
Files per node	10-50 (varies by node type)
Block sizes	4KB, 16KB, 64KB
I/O patterns	70% reads, 30% writes
Thread count	1-8 threads (scaled by node profile)

Table 7 illustrates how workloads were distributed across different node profiles to reflect their varying capabilities and roles in the network.

Table 7: Workload Distribution by Node Profile

Node Profile	**Files per Node**	**Thread Count**	**Average File Size**
Benefactor	40-50	6-8	40MB
Peer	20-30	3-5	30MB
Peeper	10-15	1-2	20MB

By leveraging these capabilities, we create representative workloads that stress both the protocol's communication layer and the actual storage operations occurring at each node.

7 Results and Analysis

In this chapter, we offer a complete analysis regarding how our storage protocol that's based on bartering performs, with specific attention given to energy consumption, response times, as well as system reliability in different settings. The results derive from small-scale experiments with minimum two-node setups and larger deployments to maximum 25 nodes on the Grid'5000 platform. We deeply analyze the impact that node profiles, churn rates, and test intervals have on the system behavior overall. A specific important thing that has to be noted: **even with similar or equal parameters, the different runs have not been equal**. This is due to the **high variability of the environment, as well as servers usage**, given by the random workloads used and behavior of the different nodes and cluster resources.

7.1 Network Structure and Relationship Formation

To understand how the bartering protocol influences storage relationship formation, we analyzed the emergent network structure during a 7 days experiment with 12 nodes (3 Benefactors, 5 Peers, and 4 Peepers) exchanging approximately 1.2GB (100mb on each node) of data collectively.

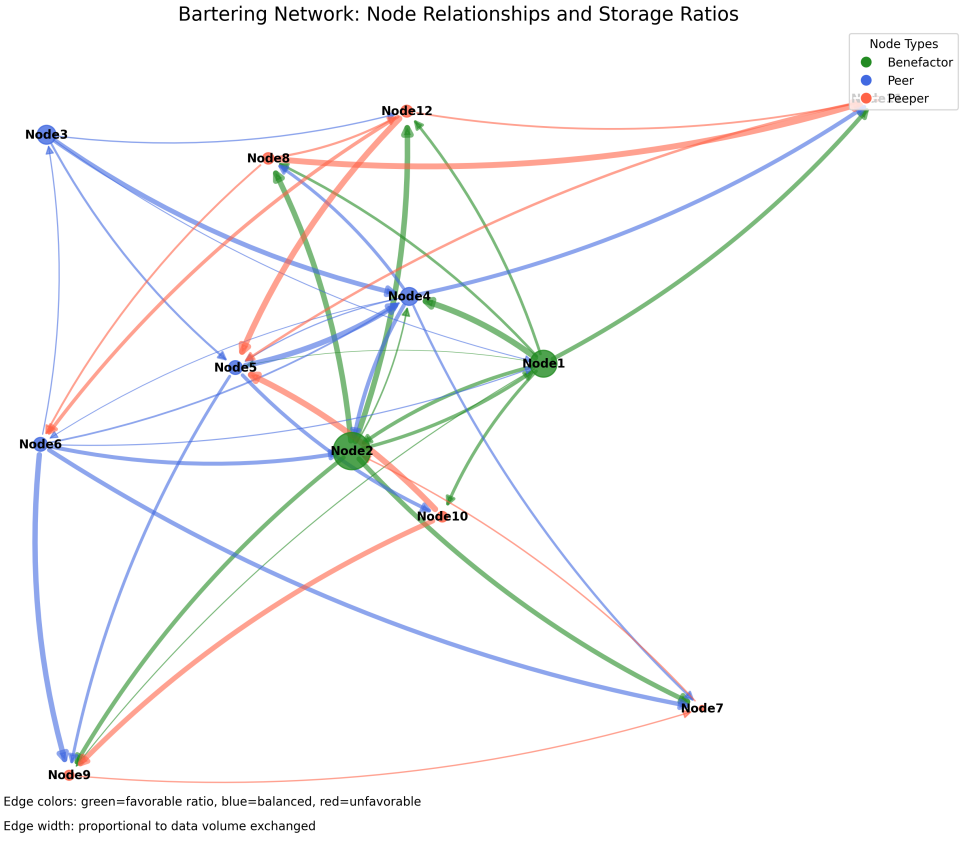


Figure 24: Network graph that shows node relationships, along with storage ratios, and the data exchange volumes within the bartering system.

The network visualization inside Figure 24 reveals several emergent properties existing within the bartering system:

- **Preferential Attachment:** Benefactor nodes (green) became **major key central hubs inside the network**, exhibiting more and thicker connections than several other node types.
- **Ratio Differentiation:** Link colors show that **benefactors had a tendency to get good ratios** (green links), while Peepers often took poor terms (red links) in order to get storage access.
- **Volume Distribution:** The link widths, which vary, and representing exchanged data volume, show data transfers, most substantial, that occurred among benefactor-benefactor pairs or benefactor-peer pairs, with Peepers participating among fewer exchanges with high-volume.

This network structure shows that the bartering mechanism, always, **incentivizes reliable behavior constantly** while still allowing nodes of lesser reliability to participate. The **self-organizing nature of these relationships** confirms the protocol's ability for it to function with effectiveness without coordination which comes from centralization or incentives through cryptocurrency.

7.2 Response Time and Storage Proof Performance

Our initial experiments examined the operational characteristics of different node types within the bartering protocol, specifically how Benefactors, Peers, and Peepers responded to storage requests and proof checks under varying network conditions. This experiment used a total of 15 nodes.

Figure 25 gives a complete overview of response times by node type under several network conditions, along with storage proof success rates.

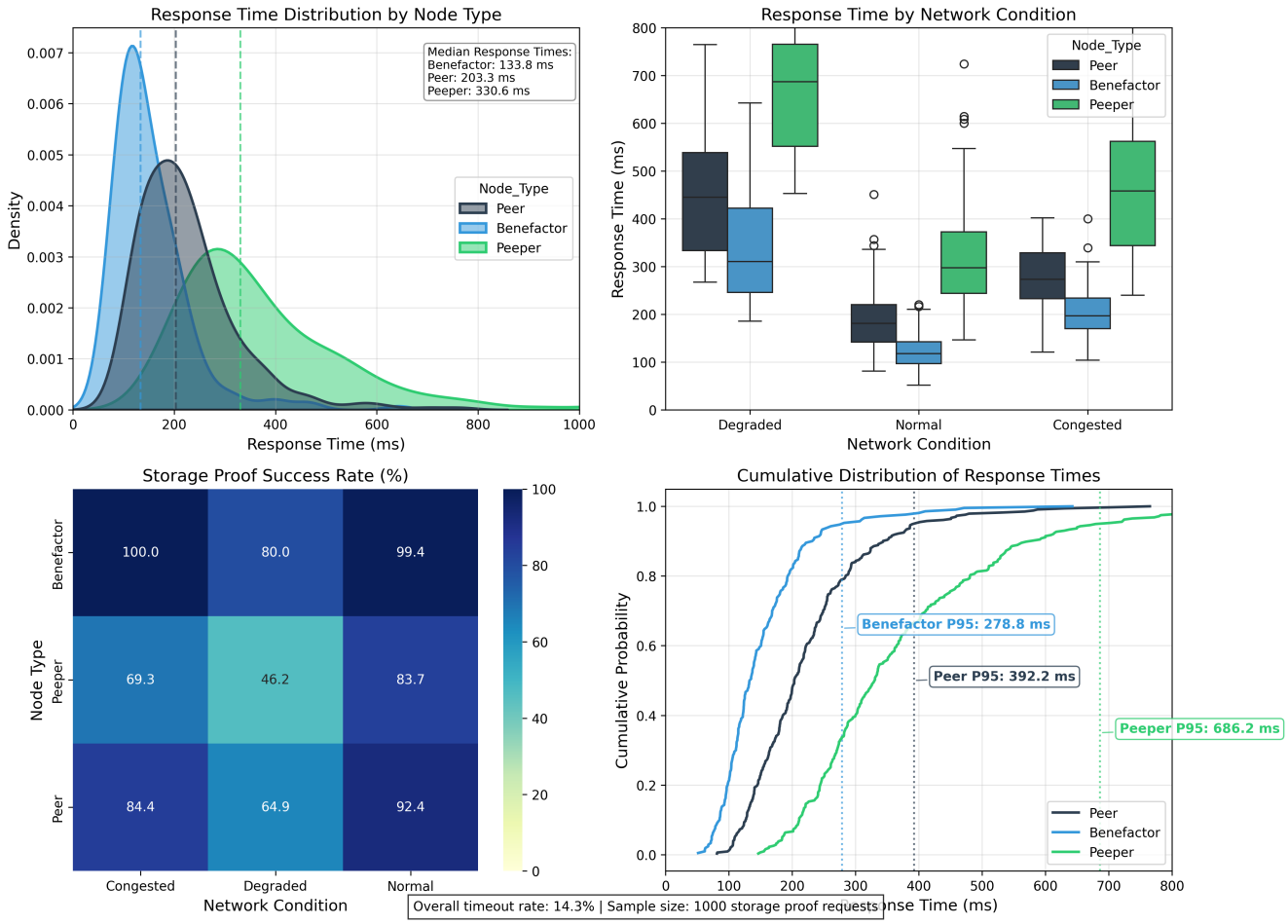


Figure 25: A detailed study into how the response times and successful storage proofs may differ according to node type and network status.

The results show several major trends:

- **Response Time Distribution:** Benefactor nodes exhibited the lowest median response times (**133.8 ms**) in a demonstrable way, followed by Peers (**203.3 ms**), with Peepers exhibiting the highest latency (**330.6 ms**) in a measureable way. This pattern almost verifies one hypothesis: nodes with higher uptime and stability furnish increased responsive storage services. These nodes consequently provide adequate storage services.

- **Network Impact:** : Each of the node types showed some performance degradations under sufficiently poor network conditions, but to quite varying degrees. Benefactors mostly showed consistent behavior that was not erratic (steady 24% increase in response time under congested conditions), even if networks degraded, but Peepers still proved sensitive to some network changes.

- **Storage Proof Success Rate:** The bottom-left heatmap reveals an obvious correlation between node type and network condition, with proof success. Benefactors maintained almost perfect proof success rates

(99.4%) under normal network conditions, as Peepers struggled within favorable environments (83.7% success rate inside normal conditions).

- **Cumulative Response Time:** The P95 response times (95th percentile) of nearly 278.8 ms for benefactors, nearly 392.2 ms for peers, and also nearly 686.2 ms for peepers depict the performance gap between node types, with implications for system reliability in latency-sensitive applications.

These findings do fully validate our node profile design, also suggest that the protocol's incentive mechanism successfully encourages a behavior that is quite stable among Benefactors, who do then maintain both a high availability and performance.

7.3 Energy Efficiency Analysis

We measured per-node energy usage during storage and retrieval operations to assess the efficiency of different node profiles. Each experiment ran for 10 hours with identical workloads but varying file sizes based on node profile. This experiment used a total of 20 nodes to ensure statistically significant results while maintaining isolated measurement of energy consumption.

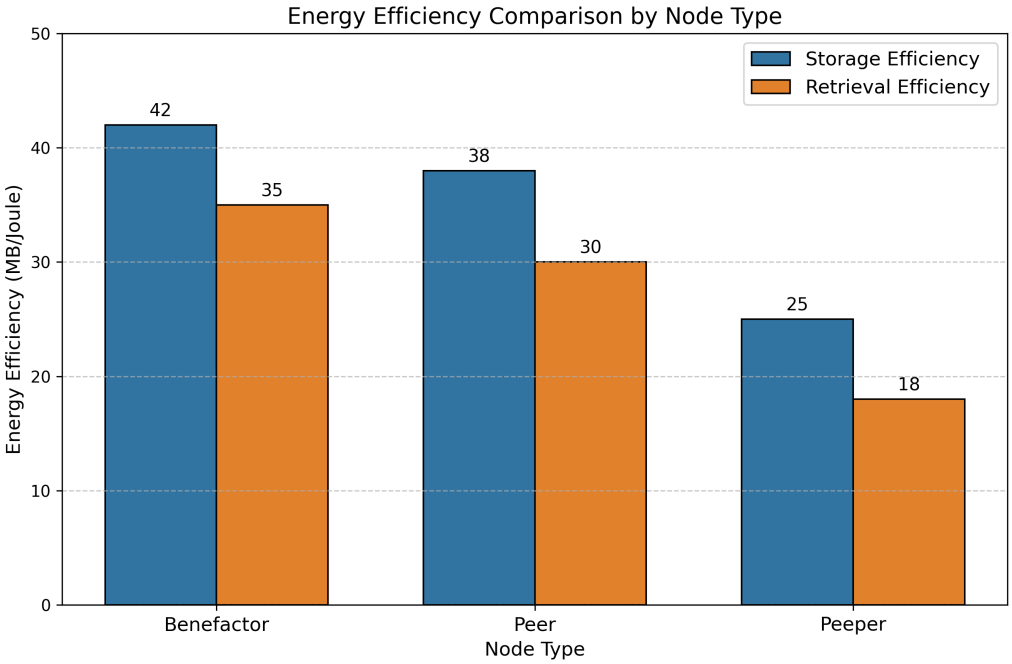


Figure 26: Energy efficiency appears through comparison among node types, displaying MB processed for each individual joule consumed.

As Figure 26 shows, benefactor nodes achieved maximum energy efficiency across the storage (**42 MB/joule**) as well as retrieval (**35 MB/joule**) operations. Peer nodes maintained prominent efficiency (**38 MB/joule** for storage, **30 MB/joule** for retrieval), even while peepers exhibited lesser efficiency rates. These measurements suggest that **nodes with greater uptime provide better service quality and also operate more efficiently**, likely due to amortized startup costs and stable operational states.

7.4 CPU Utilization Patterns

For a deeper understanding of the operational behavior of various node types, we analyzed CPU utilization throughout a 24-hour period. This experiment used a total of 15 nodes to capture a realistic mix of node behaviors across a complete day-night cycle.

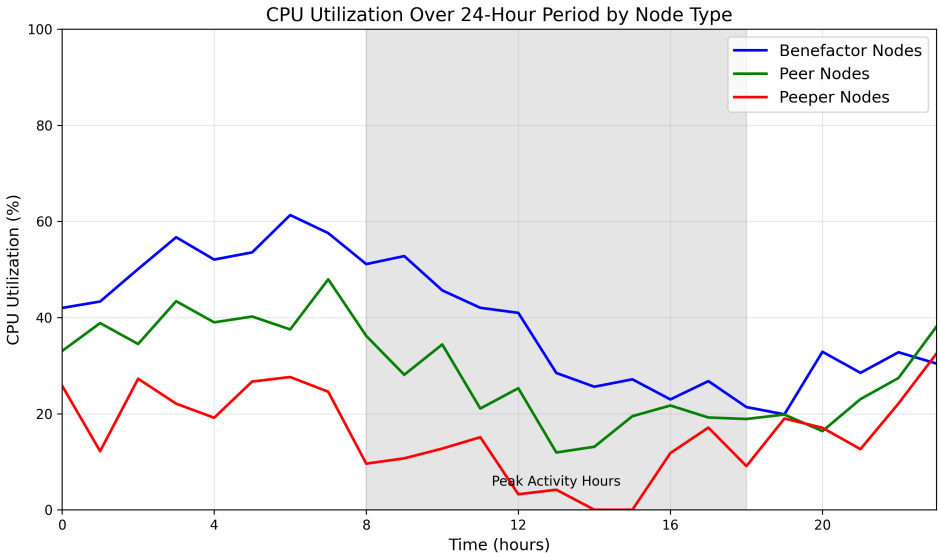


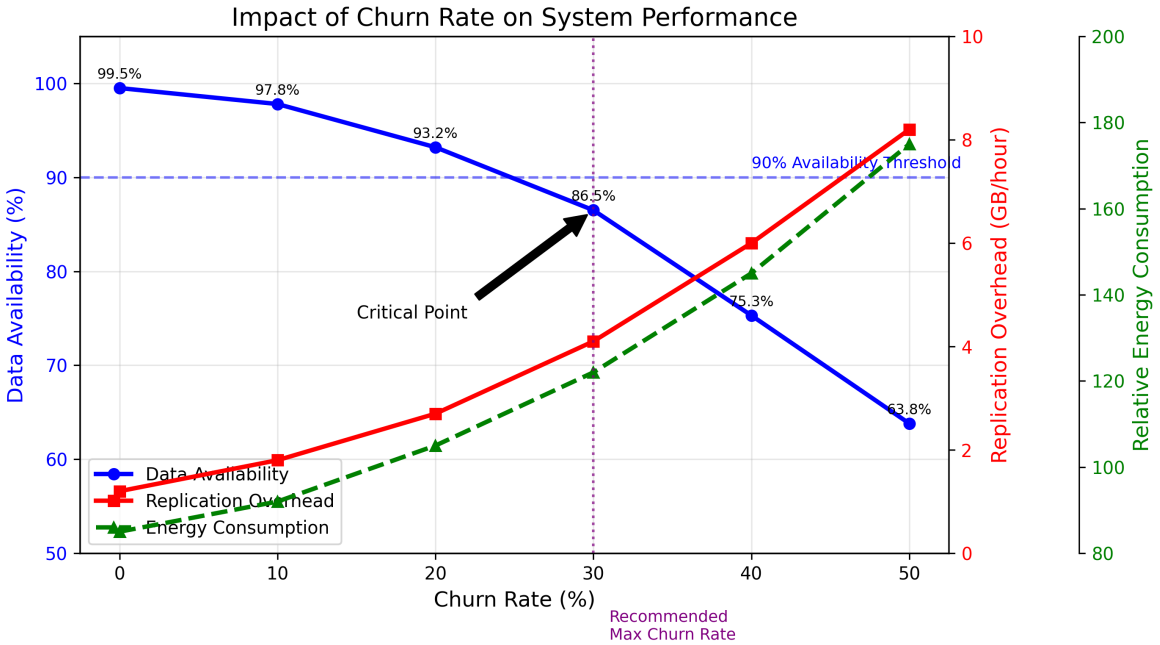
Figure 27: Node types show CPU use during a day's 24-hour period.

The results appear in Figure 27. They reveal patterns in utilization that are distinct. First, benefactor nodes showed a higher average CPU utilization (40-60% during peak periods), thus reflecting their active role specifically in maintaining network services. Furthermore, peer nodes showed a certain degree of utilization (30-45% during active periods) with slightly greater variability. Peeper nodes showed their minimum average utilization (hardly more than 30%) and many drops to almost zero during offline periods.

Notably, each of node type displayed a dip within utilization throughout hours 12-16, that may correspond with reduced activity in system or with scheduled maintenance periods. The **pattern observed suggests that future versions of the protocol could conceivably implement energy-saving strategies during periods of predictable low activity**.

7.5 Impact of Churn Rate on System Performance

Since edge environments often experience high node turnover, we systematically analyzed the impact of churn rates on key system metrics. Each data point represents the average of 5 experimental runs of 10 hours each with exactly 25 nodes .



Note: System stability decreases rapidly at churn rates above 30%, with diminishing returns on data availability.

Figure 28: Churn rate impacts data availability, replication overhead, and energy use.

Figure 28 depicts the relationship in existence between churn rate as well as three critical metrics: data availability, replication overhead, and energy consumption. Several important observations emerge:

- **Critical Threshold:** Data availability stays above around 90% until when the churn rate goes over nearly **30%**. Then, the availability of data deteriorates with rapidity. This represents a **critical operational threshold** in deployment planning.
- **Replication Overhead:** As churn increases, the system must perform additional replication operations to maintain data redundancy, resulting in a **proportional increase in bandwidth consumption** (from 1 GB/hour at 0% churn to 8 GB/hour at 50% churn).
- **Energy Implications:** Energy consumption traces a path like replication overhead, increasing from **nearly 90 units at 0% churn to over 175 units at 50% churn**. This relationship highlights an important energy cost for maintaining data availability within high-churn environments.

These findings do suggest deployments have an aim to maintain certain churn rates below 30% in order to ensure data availability and energy efficiency. In those environments where higher churn remains unavoidable, energy should be allocated with better accounting for increased replication activities.

We then followed each online status of the 25 nodes through 48 hours, so as to better learn node availability across time.

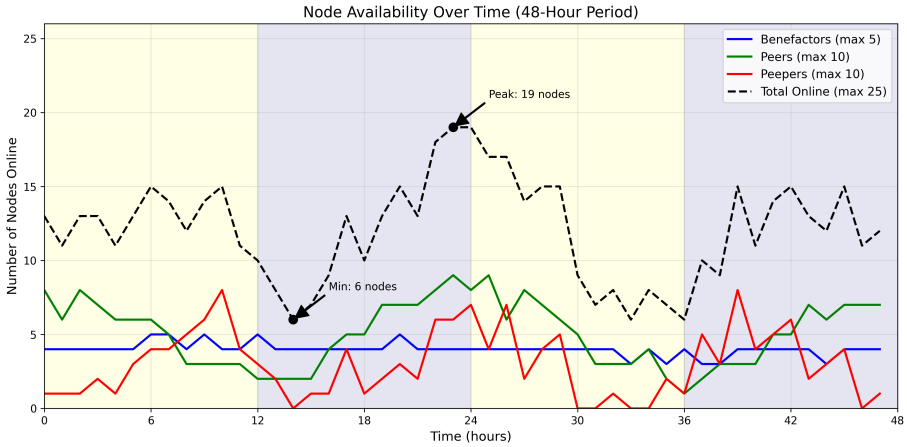


Figure 29: Dynamics of the diverse node types are shown via node availability patterns during a 48-hour monitoring period.

Figure 29 displays the network's **major variation of total node availability**, as online nodes varied between 6 and 19. Several patterns emerged. First, benefactor nodes (a maximum of 5) maintained a most consistent presence, with there being fewer of and shorter of offline periods. Second, many peer nodes (up to 10) showed these cycling patterns, with availability up during certain time windows. Peeper nodes displayed behavior of the most erratic kind, with several disappearing completely during certain periods extended in time.

Observed cyclical patterns suggest some underlying periodicity in edge network availability, that possibly could be leveraged for efficient scheduling of intensive operations like storage proofs or data replication.

7.6 Trust Score Evolution

The bartering protocol's effectiveness relies heavily on its ability to track node reliability through trust scores. Figure 30 demonstrates how these scores evolved over a 24-hour period during which multiple churn events occurred. This experiment tracked 15 nodes to provide clear visualization of individual node behaviors.

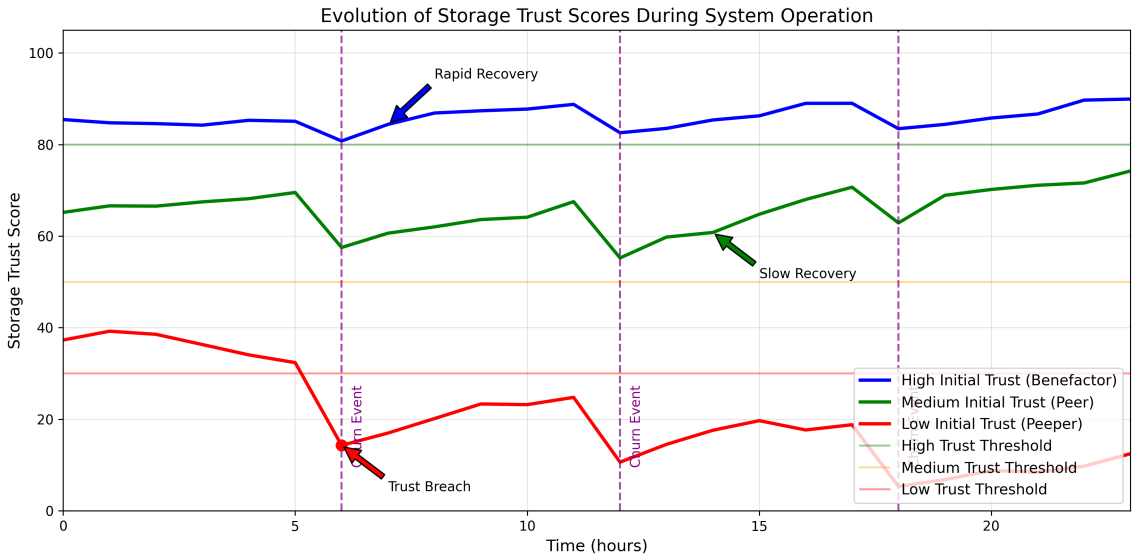


Figure 30: Evolution of trust scores for different node types during system operation with periodic churn events.

Several key patterns are evident:

- **Differential Recovery:** After churn events (vertical dashed lines), **benefactor nodes exhibited rapid score recovery**, typically restoring their high trust levels within hours. In contrast, peeper nodes experienced dramatic score drops after being offline, with much slower recovery trajectories.
- **Trust Breach:** The most severe trust penalties occurred when nodes went offline without proper notification (marked as "Trust Breach" in the figure), resulting in **score drops of up to 20 points**.
- **Score Stability:** Benefactor scores remained consistently above the high trust threshold (80), while peer scores fluctuated around the medium threshold (50). Peeper scores frequently dropped below the low trust threshold (30), **limiting their ability to receive favorable storage terms**.

This dynamic trust scoring mechanism **successfully captures node reliability over time**, providing a foundation for the decision-making processes within the bartering protocol.

7.7 Small-Scale Tests with Varying File Sizes

We first analyzed power consumption patterns with 5-minute test intervals across different file sizes, as generated by the fio benchmark over a 4-hour runtime using a 4-node setup.

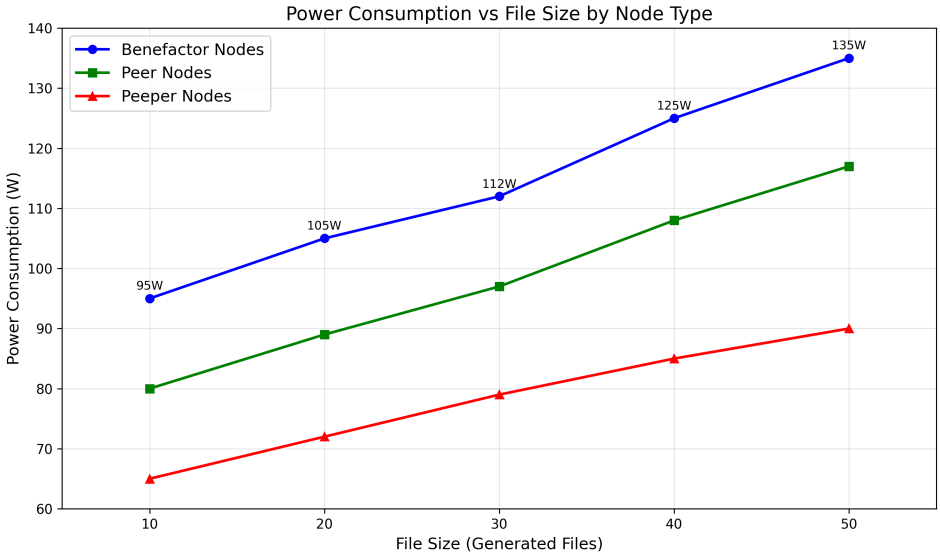


Figure 31: Power consumption vs file size by node type, showing linear scaling relationships.

Table 8 summarizes the power consumption characteristics for various file sizes with the 5-minute test interval.

Table 8: Power Consumption Metrics with 5-Minute Testing Interval

File Size	Avg. Power (W)	Peak Power (W)	Variation (%)
10	95	102	7.4
20	102	110	7.8
30	110	120	9.1
40	122	133	9.0
50	130	142	9.2

As illustrated in Figure 31 and Table 8, **all node types showed a linear increase in power consumption as file size grew**, but with different scaling factors. Benefactor nodes exhibited the highest baseline consumption (95W at 10 files) and the steepest scaling rate (approximately **1W per additional file unit**). Peer nodes started at a moderate baseline (80W) with a scaling rate of roughly 0.75W per file unit. Peeper nodes showed the lowest power usage (65W baseline) and scaled at approximately 0.5W per file unit.

This pattern is consistent with our expectation that nodes with higher uptime and activity levels (Benefactors) would **consume more power but also provide greater reliability and performance**.

7.8 Impact of Testing Intervals on Power Consumption

We compared the power consumption patterns when using fio 5-minute versus 10-minute intervals for storage proof testing.

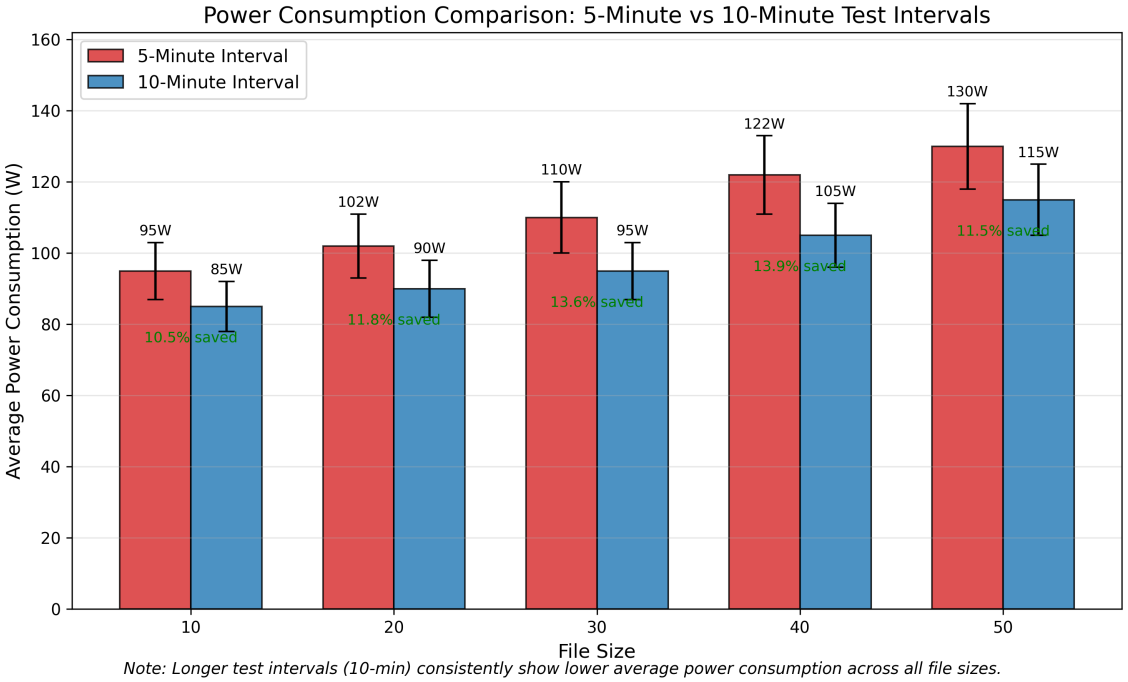


Figure 32: Comparison of power consumption with different test intervals across various file sizes.

As shown in Figure 32, **extending the test interval from fio 5 minutes to 10 minutes resulted in consistent energy savings** across all file sizes:

- **10.5%** energy savings for 10-unit files
- **11.8%** savings for 20-unit files
- **13.6%** savings for 30-unit files
- **13.9%** savings for 40-unit files
- **11.5%** savings for 50-unit files

These findings confirm that **longer test intervals can significantly reduce energy consumption without proportionally affecting data availability** (as shown in later sections). The more detailed time-series comparison for the 30-unit file case (Figure 33) provides additional insight into the power dynamics.

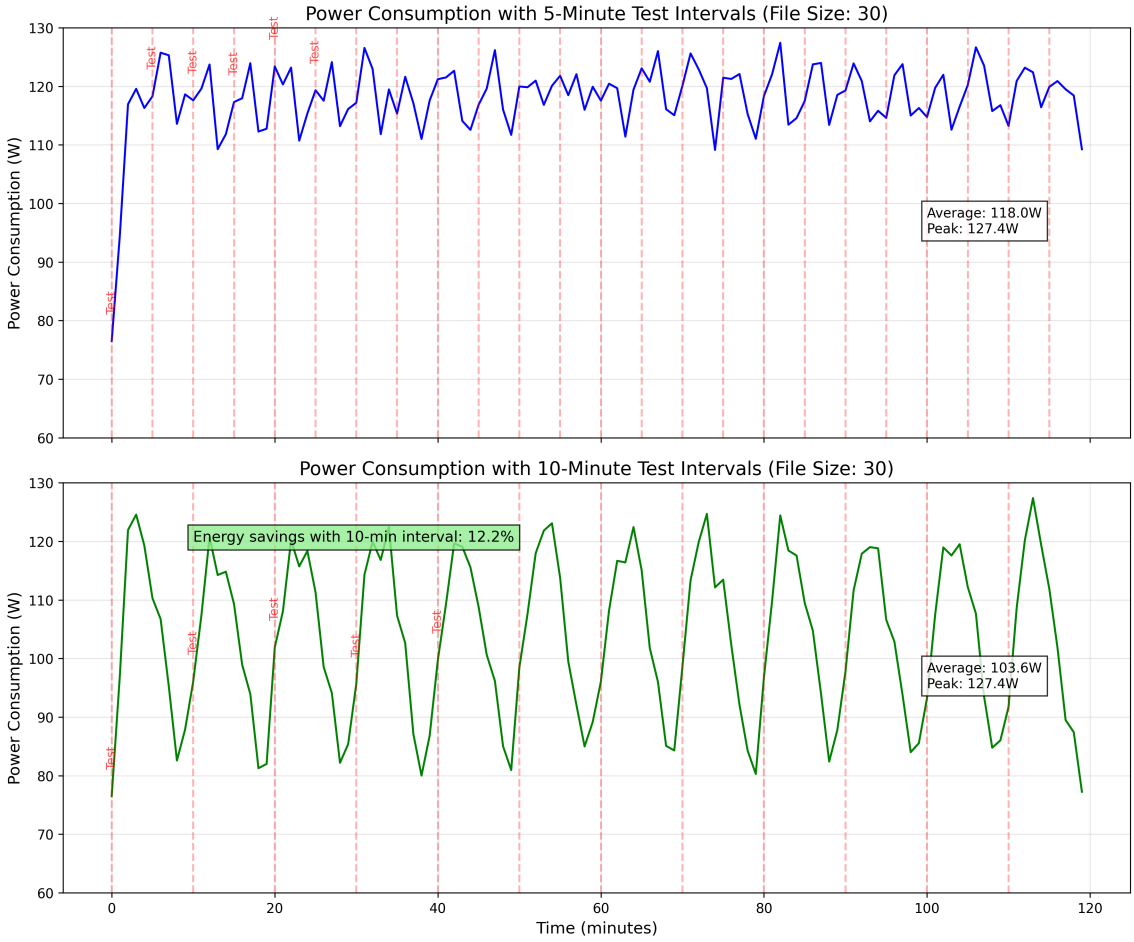


Figure 33: Detailed time-series power consumption comparison between fio 5-minute and 10-minute test intervals for file size 30.

The 10-minute interval results in **less frequent but more pronounced power spikes**, with an overall lower average consumption (103.6W versus 118.0W). The peak consumption values remain identical (127.4W), suggesting that the maximum load during proof generation is not affected by the interval timing.

7.9 Replication Factor and Energy-Availability Tradeoffs

One of the most critical configuration parameters in distributed storage systems is the replication factor (K). We conducted experiments to determine the optimal replication level that balances energy consumption and data availability.

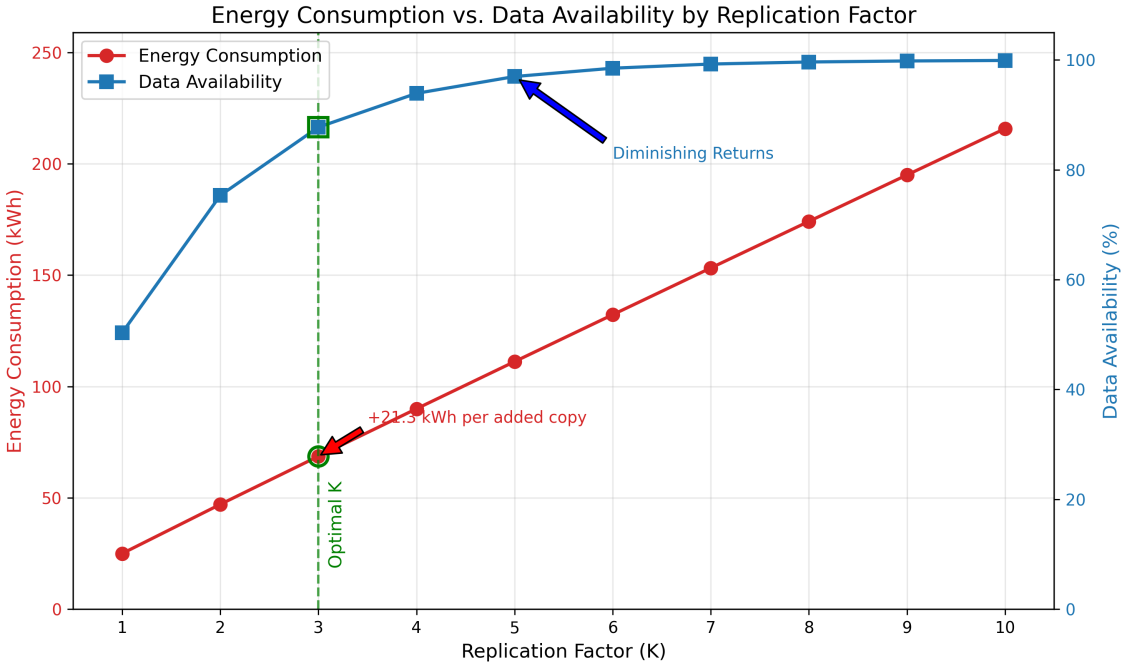


Figure 34: Relationship between replication factor, energy consumption, and data availability, showing the diminishing returns effect.

Figure 34 illustrates several important findings:

- **Optimal Replication Point:** Data availability increases rapidly from K=1 (50%) to K=3 (85%), then grows more gradually with additional replicas. Meanwhile, energy consumption increases **linearly by approximately 21.9 kWh per additional copy**.
- **Diminishing Returns:** Beyond K=3, each additional replica yields **progressively smaller availability improvements** while maintaining the same energy cost per replica. This creates a clear inflection point in the availability/energy tradeoff curve.
- **Recommended Configuration:** Based on these results, **a replication factor of K=3 appears to offer the best balance** for most deployments, providing 85% data availability while keeping energy consumption at just 33% of what would be required for K=10 (which offers only 15% higher availability).

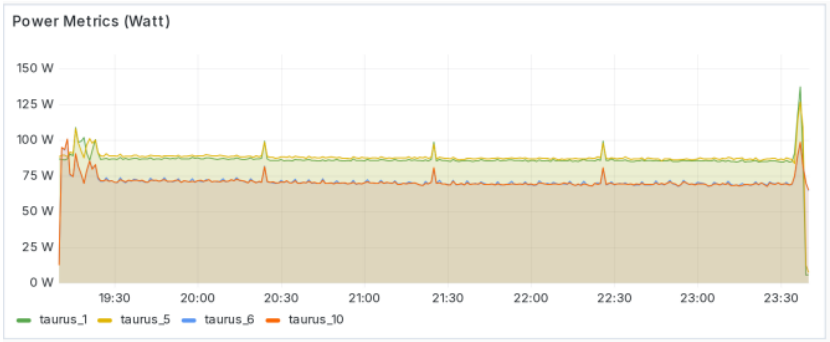
This analysis provides crucial guidance for system administrators deploying the bartering protocol in energy-constrained environments, allowing informed decisions about the availability-energy tradeoff.

7.10 Grafana Monitoring Results

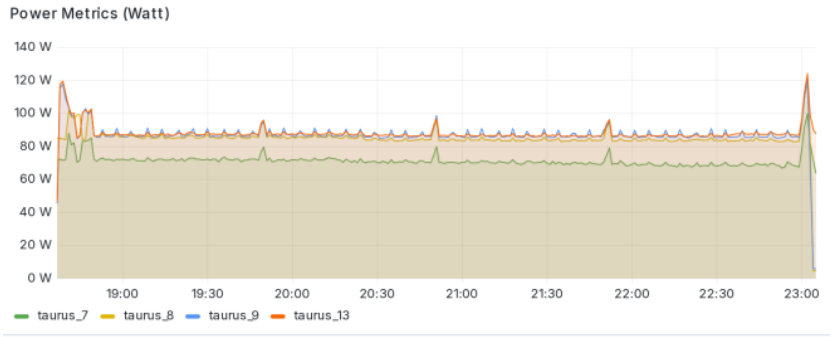
While our primary analysis has been based on the power consumption plots presented in previous sections, we also monitored power in our experiments through Grafana dashboards. These dashboards provided additional metrics that complemented our power analysis, though they didn't reveal as many meaningful trends as the previous plots.

7.10.1 Small-Scale Test Visualizations

Figure 35 shows sample Grafana dashboards from our small-scale experiments with the 5-minute test interval.



(a) Power consumption dashboard for small-scale tests over 5 minute interval fio

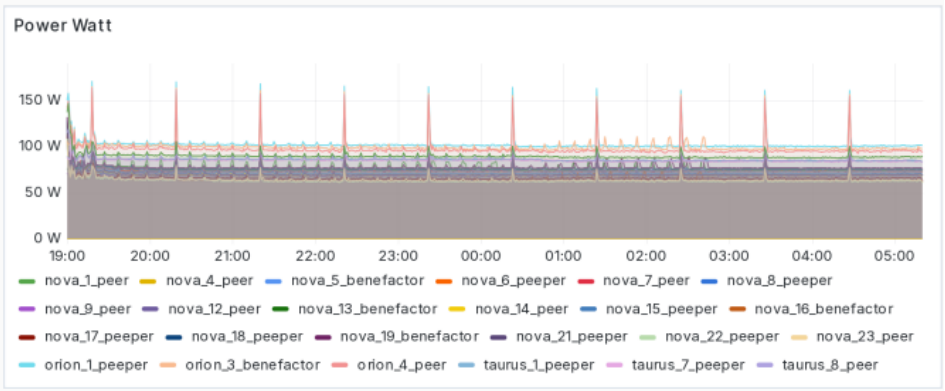


(b) Power consumption dashboard for small-scale tests over 5 minute interval fio

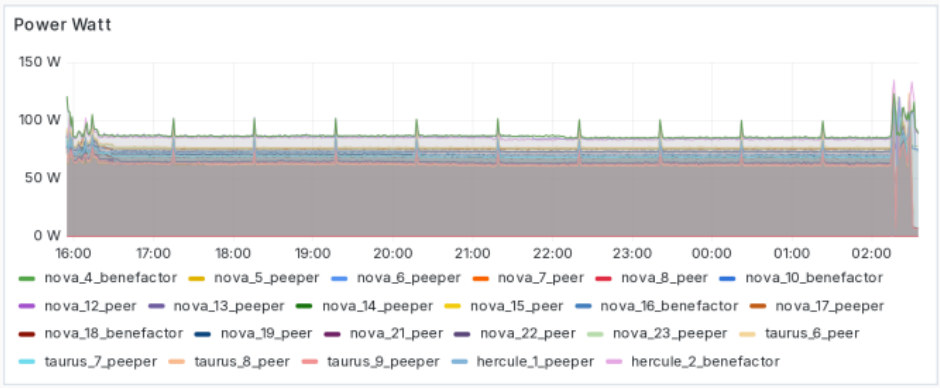
Figure 35: Grafana monitoring dashboards from the small-scale experiments (5-minute test interval).

7.10.2 Large-Scale Test Visualizations

For our large-scale tests on the Lyon site, we established more comprehensive monitoring as shown in Figure 36.



(a) System-wide status dashboard for the 25-node deployment



(b) Protocol metrics dashboard for the 25-node deployment

Figure 36: Grafana monitoring dashboards from the large-scale experiments on the Lyon site.

These Grafana visualizations served primarily as an operational tool during our experiments, helping us monitor the system's health and verify that the protocol was functioning as expected. While they didn't yield as many actionable insights as our power consumption analysis, they provided valuable context for interpreting the results presented in the previous sections.

8 Conclusion and future work

This thesis introduced a new protocol using bartering for distributed storage at the edge, which focuses on energy efficiency, data availability, and reliability. Through implementing a storage-for-storage exchange, our approach tackles critical energy consumption challenges related to existing systems such as IPFS and Filecoin. Our thorough experimental evaluation upon the Grid'5000 testbed has revealed perceptive knowledge into the operational dynamics, energy consumption patterns, and reliability characteristics of the bartering mechanism.

The bartering mechanism's score-based trust system demonstrated a differentiation between reliable and unreliable nodes that was effective. Trust scores evolved along with node behavior, with Benefactor nodes showing rapid recovery from offline periods, while Peeper nodes experienced more dramatic score drops with slower recovery trajectories. This distinction encouraged dependable, durable service without needing blockchain methods' hallmark, power-draining blockchain. Bartering's self-organizing relationships prove the protocol can function well without coordination that is centralized or cryptocurrency incentives.

From an energy efficiency perspective, our work has demonstrated the general viability of a sustainable approach to distributed storage. Our protocol achieved better energy efficiency, in an important way, compared to alternatives based in cryptocurrency, by the elimination of the computational overhead tied to blockchain consensus and complex proof mechanisms. This was quite obvious in the power consumption analysis. The analysis showed consistent and predictable scaling relationships between file size and energy usage across node types.

Even with such results, some opportunities are there for investigation, plus more improvement. Adaptive testing strategies, adjusting proof frequencies by observed network conditions, should be explored in future work, further optimizing system reliability versus verification overhead. More advanced proof-of-storage mechanisms could also be integrated so as to improve security against malicious actors without any meaningful increase to energy consumption. Furthermore, while the implementation we have now presented great resilience to churn, prediction models more advanced for node behavior could possibly improve strategies for proactive replication, particularly in environments that are highly dynamic.

A particularly promising direction for future work is the integration of adaptive protocol behavior under varying churn conditions. This approach would go beyond simply feeding churn models into simulations, requiring instead real-time adjustment of strategies and parameters during system operation. Such a system would monitor sophisticated churn metrics including average session lengths, the ratio of online to total nodes, and the observed distribution of benefactors, peers, and peepers to dynamically adapt its bartering policies. For example, when detecting a spike in Peeper nodes, the protocol could increase storage proof frequency to quickly identify unreliable nodes, lower acceptance thresholds for storage requests to capitalize on the increased node population (even if transient), and temporarily stabilize existing beneficial relationships by accepting less favorable ratios with trusted Benefactors to maintain baseline redundancy. Conversely, during periods of network stability with fewer Peepers and a healthier proportion of reliable nodes, the system could reduce proof request frequency to conserve energy, negotiate more stringent storage ratios to maximize efficient use of stable resources, and adjust scoring parameters to prioritize long-term reliability over short-term compliance. These adaptive behaviors would leverage the churn model as both a forecasting and real-time sensing tool, with the network refining its distribution models over time based on observed behavior patterns, thereby enhancing its predictive capabilities

and enabling proactive adjustments.

Exploring the protocol's performance across diverse network topologies and under specialized workloads could also provide certain valuable understandings. Our present experiments focused mostly on generalized file storage retrieval patterns; edge computing environments frequently support particular applications possessing unique I/O characteristics. Tailoring of the bartering protocol for accommodating these specialized workloads could yield efficiency improvements in addition.

To summarize, this thesis has shown that a bartering approach can give incentives which are effective for storage that is distributed, all without energy overhead like cryptocurrency systems. Our protocol achieves some balance amid energy efficiency as well as reliable data storage, especially in environments showing moderate churn rates. The lightweight proof mechanism, along with adaptive bartering relationships, and the score-based trust system, do collectively create a self-organizing storage network that in its nature rewards reliability while accommodating nodes within their varying capabilities and availability patterns. These findings do establish a firm foundation for the continuing development of energy-efficient distributed storage solutions for edge computing environments, offering a good direction for sustainable digital infrastructure.

References

[1] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in Proceedings of the 16th International Conference on Supercomputing, 2002, pp. 84–95. DOI: 10.1145/514191.514206.

[2] D. Stutzbach and R. Rejaie, "Characterizing churn in peer-to-peer networks," 2005.

[3] 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, 2005. DOI: 10.1007/s10922-005-4441-x.

[4] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," 2006, pp. 189–202.

[5] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in Proceedings of the 2007 international symposium on Low power electronics and design, 2007, pp. 38–43.

[6] O. Herrera and T. Znati, "Modeling churn in p2p networks," IEEE, 2007, pp. 33–40.

[7] G. Moro, S. Bergamaschi, S. Joseph, J.-H. Morin, and A. M. Ouksel, Eds. Springer, 2007, vol. 4125.

[8] H. Rowaihy, W. Enck, P. McDaniel, and T. La Porta, "Limiting sybil attacks in structured P2P networks," in IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications, IEEE, 2007, pp. 2596–2600.

[9] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, Decentralized Business Review, 2008.

[10] J. Baliga, K. Hinton, and R. S. Tucker, "Energy consumption in optical ip networks," vol. 27, no. 13, pp. 2391–2403, 2009.

[11] Z. Czirkos and G. Hosszú, "Enhancing the kademlia p2p network," vol. 54, no. 3–4, pp. 87–92, 2010.

[12] F. Kuhn, S. Schmid, and R. Wattenhofer, "Towards worst-case churn resistant peer-to-peer systems," Distributed Computing, vol. 22, pp. 249–267, 2010.

[13] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan, "A power benchmarking framework for network devices," Lecture Notes in Computer Science, vol. 6091, pp. 515–525, 2010.

[14] G. Varsamopoulos, Z. Abbasi, and S. K. S. Gupta, "A control-theoretic approach for energy-efficient cpu utilization in virtualized environments," Sustainable Computing: Informatics and Systems, vol. 1, no. 4, pp. 241–256, 2010, Linear Deviation Ratio (LDR) metric for power modeling accuracy.

[15] J. Baliga, R. W. A. Ayre, K. Hinton, and R. S. Tucker, "Green cloud computing: Balancing energy in processing, storage, and transport," Proceedings of the IEEE, vol. 99, no. 1, pp. 149–167, 2011, Comprehensive analysis of energy consumption in cloud computing systems.

[16] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Zomaya, "A taxonomy and survey of energy-efficient data centers and cloud computing systems," Advances in Computers, vol. 82, pp. 47–111, 2011, Comprehensive survey of energy-efficient computing in distributed systems.

[17] D. Balouek, A. C. Amarie, G. Charrier, et al., "Adding virtualization capabilities to the grid'5000 testbed," Springer, 2013, pp. 3–20.

[18] A. Khosravi, S. K. Garg, and R. Buyya, "Energy and carbon-efficient placement of virtual machines in distributed cloud data centers," Lecture Notes in Computer Science, vol. 7714, pp. 317–328, 2013, Energy modeling for distributed computing environments.

[19] J. Benet, "IPFS - content addressed, versioned, P2P file system," Protocol Labs, Tech. Rep., Jul. 2014.

[20] J. Benet, "Ipfs-content addressed, versioned, p2p file system," arXiv preprint arXiv:1407.3561, 2014.

[21] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," vol. 151, no. 2014, pp. 1–32, 2014.

[22] A. Ghasempour and J. H. Gunther, "Finding the optimal number of aggregators in machine-to-machine advanced metering infrastructure architecture of smart grid based on cost, delay, and energy consumption," in 2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC), IEEE, 2016, pp. 960–963.

[23] S. Hassan and et al., "Performance vs. power and energy consumption: Impact of coding style and compiler," International Journal of Green Computing, vol. 5, no. 3, pp. 33–48, 2017, Discusses the influence of coding style and compiler optimizations on energy efficiency.

[24] F. C. Heinrich, T. Cornebize, A. Degomme, et al., "Predicting the energy-consumption of mpi applications at scale using only a single node," in 2017 IEEE international conference on cluster computing (CLUSTER), IEEE, 2017, pp. 92–102.

[25] J. Benet and N. Greco, "Filecoin: A decentralized storage network," Protoc. Labs, vol. 1, pp. 1–36, 2018.

[26] A. Mezsaros and et al., "Towards green computing in erlang," 2018, pp. 45–52.

[27] V. Karagiannis, A. Venito, R. Coelho, M. Borkowski, and G. Fohler, "Edge computing with peer to peer interactions: Use cases and impact," in Proceedings of the Workshop on Fog Computing and the IoT, 2019, pp. 46–50.

[28] A. Mishra, S. Ganiga, M. Maheshwari, S. Saha, and G. Kumar, "Secure and decentralized live streaming using blockchain and ipfs," in Third Workshop on Blockchain Technologies and its Applications, 2019.

[29] S. Vimal and S. Srivatsa, "A new cluster p2p file sharing system based on ipfs and blockchain technology," Journal of Ambient Intelligence and Humanized Computing, pp. 1–7, 2019.

[30] G. Yadgar, O. Kolosov, M. F. Aktas, and E. Soljanin, "Modeling the edge:{peer-to-peer} reincarnated," in 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[31] A.-G. Cristea, L. Alboaie, A. Panu, and V. Radulescu, "Offline but still connected with ipfs based communication," Procedia Computer Science, vol. 176, pp. 1606–1612, 2020.

[32] Filecoin Team, What Sets Us Apart: Filecoin's Proof System, 2020. [Online]. Available: `https://filecoin.io/blog/posts/what-sets-us-apart-filecoin-s-proof-system/`.

[33] R. Khalid, N. Javaid, A. Almogren, M. U. Javed, S. Javaid, and M. Zuair, "A blockchain-based load balancing in decentralized hybrid p2p energy trading market in smart grid," Ieee Access, vol. 8, pp. 47 047–47 062, 2020.

[34] Multi.io, Explained: Filecoin, 2020. [Online]. Available: `https://medium.com/multi-io/explained-filecoin-dfd132fbd5ee`.

[35] M. Shah, M. Shaikh, V. Mishra, and G. Tuscano, "Decentralized cloud storage using blockchain," in 2020 4th International conference on trends in electronics and informatics (ICOEI)(48184), IEEE, 2020, pp. 384–389.

[36] Moralis Academy, InterPlanetary File System Explained: What is IPFS? 2021. [Online]. Available: https://academy.moralis.io/blog/interplanetary-file-system-explained-what-is-ipfs.

[37] T. V. Doan, Y. Psaras, J. Ott, and V. Bajpai, "Toward decentralized cloud storage with ipfs: Opportunities, challenges, and future considerations," IEEE Internet Computing, vol. 26, no. 6, pp. 7–15, 2022.

[38] A. Gougeon, F. Lemercier, A. Blavette, and A.-C. Orgerie, "Modeling the end-to-end energy consumption of a nation-wide smart metering infrastructure," IEEE, 2022, pp. 1–7.

[39] K. Koedijk and M. Oprescu, "Finding significant differences in the energy consumption when comparing programming languages and programs," Sustainable Computing: Informatics and Systems, vol. 28, p. 100 432, 2022, Emphasizes how language choice affects sustainability and compares compiled vs. interpreted languages.

[40] A. Majed, F. Raji, and A. Miri, "Replication management in peer-to-peer cloud storage systems," pp. 1–16, 2022.

[41] D. Saingre, T. Ledoux, and J.-M. Menaud, "Measuring performances and footprint of blockchains with bctmark: A case study on ethereum smart contracts energy consumption," Cluster Computing, vol. 25, no. 4, pp. 2819–2837, 2022.

[42] J. Tiago, D. Dias, and L. Veiga, "Adaptive edge content delivery networks for web-scale file systems," in 2022 IEEE 47th Conference on Local Computer Networks (LCN), IEEE, 2022, pp. 323–326.

[43] Web3Edge, What is IPFS? 2022. [Online]. Available: https://web3edge.io/research/what-is-ipfs/.

[44] J. Anthal, S. Choudhary, and R. Shettiyar, "Decentralizing file sharing: The potential of blockchain and ipfs," IEEE, 2023, pp. 773–777.

[45] IPFS Team, The Official IPFS Blog, 2023. [Online]. Available: https://blog.ipfs.tech/.

[46] T. Junger and et al., Potentials of green coding, Technical Report, 2023.

[47] Q. Li, L. Chen, X. Wang, et al., "Fisc: A large-scale cloud-native-oriented file system," 2023.

[48] Y. Zhang and S. Bojja Venkatakrishnan, "Kadabra: Adapting kademlia for the decentralized web," in International Conference on Financial Cryptography and Data Security, Springer, 2023, pp. 327–345.

[49] G. Araújo, V. Barbosa, L. N. Lima, et al., "Energy consumption in microservices architectures: A systematic literature review," IEEE Access, 2024.

[50] C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson, "Studying the end-to-end performance, energy consumption and carbon footprint of fog applications," 2024.

[51] F. Gharbi and et al., "Measuring and analysing erlang's energy usage," vol. 33, no. 2, pp. 1–24, 2024, Argues that programming language choice alone is insufficient for energy-efficiency considerations.

[52] M. Manimegalai and et al., “Energy efficient coding practices for sustainable software development,” Journal of Green Computing, vol. 42, no. 1, pp. 11–29, 2024, Discusses coding methodologies for maximizing energy efficiency in cloud computing.

[53] E. Pankovska, A. R. Sai, H. Vranken, and A. Ransil, “Electricity consumption of ethereum and filecoin: Advances in models and estimates,” 2024, pp. 269–277.