Network coding for automated distributed storage system: adar

Craig Anderson
cand380@aucklanduni.ac.nz
The University of Auckland
Auckland, New Zealand

ABSTRACT

A distributed storage system can meet the increasing requirements of consumer storage capacity. It also enables effective access to data across multiple host devices at once. However, existing distributed storage solutions require configuration and understanding of the underlying system, which is not suitable for consumer use. This paper describes and implements a simple, automated distributed storage system. It encodes the stored data using network coding techniques, to enable efficient and simple data retrieval for use. The designed protocol is evaluated against existing solutions on two fronts: SMB 2 for network transfer efficiency, and RAID for storage redundancy. The proposed design is found to have design advantages, though the reference implementation requires further optimisation to be effective.

KEYWORDS

network coding, distributed storage, automation, automated, storage coding

1 INTRODUCTION

Network coding is a useful technique to allow for data redundancy and effective reconstruction, as well as efficient distribution [16]. Network coding can be applied to the data for distribution and storage amongst the nodes in the system, which enables for storage redundancy, and reconstruction of the data during use. Notably, network coding does not depend on any specific piece of data, and instead abstracts it into a series of symbols, each of which contribute to decode the original data. These properties make it a particularly useful tool to apply to a distributed storage system.

Distributed storage systems are not used in consumer environments, as their complexity and scale makes them largely unsuitable [4, 5]. Distributed systems are becoming increasingly common, in particular with the adoption of Internet of Things (IoT) devices [15]. IoT devices will apply presure to cloud storage providers, which could be relieved by effective and distributed storage on the devices themselves.

Conversely, Network Attached Storage devices are an acceptable consumer solution, and they have been made relatively simple to use [5]. Distributed storage systems would also be an effective consumer solution, if any viable options existed. Applying the same simplicity to a distributed system, will better enable consumer use.

This paper outlines a system that is capable of automated discovery, configuration, and management of a distributed storage system within a consumer networking environment.

The system has been named 'adar' as it is simple and easily pronounceable. The choice in name has very little bearing on the actual use of the system, although a name is required for categorising the advertisement of the service.

2 LITERATURE REVIEW

Within the space of distributed storage systems, both network coding and automation have been explored in various aspects separately. No system has yet investigated a holistic of an automated distributed storage system using network coding.

2.1 Network coding with distributed storage

Network coding has been used to ensure the storage of data in a peer-to-peer distributed network where nodes cannot be guaranteed to be online at all times [10]. The paper weights the random linear coding based on the priority of the data to be stored. They demonstrate Priority Random Linear Network Coding to be an effective solution, and enable partial data recovery of prioritised data when losses occur.

Network coding can also be implemented such that multiple-node failures are tolerated and repaired [16]. This has been shown to require a minimum amount of bandwidth to reconstruct data, when effectively collaboration occurs. This demonstrates that network coding can achieve bandwidth-effective data distribution from multiple source nodes to a single node for data storage.

2.2 Automation with distributed storage

Distributed storage is a useful technique for designing a system that best matches a use case, however the complexity of configuring such a system can be undesirable or even infeasible. An additional monitoring system can be used to adjust parameters automatically to find the most optimal configuration during runtime [10].

Furthermore, the design of storage tiers and assignment of data to tiers is a time-consuming administration task when using a heterogeneous storage system. This process can be automated by suitable algorithms during runtime, where the system is adapted to its use [12].

Whilst bandwidth is typically a limiting factor during data transfer over a network, the capacity of the transmitting and receiving node are also key factors. Both upload and download times of a client can be improved by distributing data to multiple storage nodes directly during the transfer [11].

Ensuring the integrity and security of data in a distributed system is an important aspect to consider during distribution, particularly with regards to privacy. SSSFS is a reliable method for adding authentication at the virtual file system layer [17].

3 BACKGROUND

There are two key concepts to discuss as background for understanding the results and comparison of against the proposed protocol. These concepts are RAID, and network coding.

3.1 RAID

A Redundant Array of Inexpensive (or Independent) Disks (RAID) uses multiple disks together to achieve higher reliability and potentially higher performance [13]. RAID has several different modes of operation, and is configurable by the end user. It requires the host Operating System to support a special disk access driver, which allows the RAID controller to present as a single disk to the Operating System. It then routes disk operations to the relevant individual disks, according to the selected mode.

RAID provides data redundancy in the event of hardware disk failures via several different modes, each of which also impact performance positively or negatively. It does not distribute data beyond a single host, nor protect from hardware failures of the host.

3.1.1 First level RAID. The simplest mode is to mirror the available disks, such that disk operations to one disk are exactly performed with the other, simultaneously. Thus, in the event of a failure of either disk, data is not lost. This provides no performance benefit, though does provide a certain level of data redundancy.

3.1.2 Second level RAID. An alternative mode is to use a Hamming code as an Error Correcting Code over each bit of data. A portion of the disks are used for storing bits, whilst a portion store the corresponding Hamming Error Correction Code.

3.1.3 Third level RAID. This is similar to the second level RAID, except that data is distributed at the byte level. Hamming Error Correcting Codes are also used, and the total data is distributed over several disks.

3.1.4 Fourth level RAID. Data is distributed at the byte level throughout the data disks, with a single disk dedicated to providing a parity check for the data bytes.

3.1.5 Fifth level RAID. Data is distributed at the byte level throughout the data disks, with a parity check. However, the parity byte is also distributed across the disks, such that no one disk is dedicated to storing all parity data.

3.2 Network coding

Network coding is a technique of encoding data to improve the throughput and efficiency of a network. Linear network coding uses linear combinations of data to produce symbols that can be transmitted across a network [9]. The linear combinations are chosen and use coefficients to generate a single symbol that contributes to the data bytes as specified by the coefficients. With a sufficient number of symbols, along with independent combinations of coefficients, the original data can be decoded.

If symbols are lost, such as via dropped packets during transmission, then new and unique symbols can continue to be generated and transmitted. This enables the recipient to decode the data, without any coordination of which data was lost in transit.

This can further be improved by randomly selecting the coefficients used for encoding symbols, resulting in Random Linear Network Coding (RNLC) [7]. When using RNLC, the coefficients used for each symbol do not need to be transmitted with the symbol, which significantly reduces transmission overhead. Instead, the seed used for the pseudorandom number generator can be transmitted along with the symbols. This enables the recipient to regenerate the coefficients, and then apply them to the symbols to decode the original data. This also removes the necessity to carefully chose appropriate and independent combinations of coefficients, resulting in a simpler algorithm.

However, it can still be useful to intentionally bias or choose the coefficients. For example, biasing the chosen coefficients towards the start of the data enables the receiver to start decoding the data immediately. This reduces the restriction of requiring all symbols to decode the data, and can allow for it to be streamed more easily.

4 DESIGN

When designing and specifying a new communications protocol, it is important to begin with a clear set of design concepts and goals to guide each small decision. Referring back to these concepts and goals throughout the design and implementation phase is key to ensuring that a robust and fit-for-purpose outcome is found. As such, the specific design concepts and goals will be briefly discussed.

4.1 Internet connectivity

Whilst internet infrastructure can provide useful tools and systems, such as certificate chains for secure identification, it also adds another external dependency. Particularly as all other aspects of this system are inherently suitable for local area networks, introducing internet connectivity as a dependency is not suitable for the reliable operation of the system in a diverse set of networks. A key design goal is to enable the use of the system between two or more peers via a direct link-local connection, where there is no dedicated networking infrastructure such as a router, DHCP server, or internet gateway.

4.2 3-step process

A key factor of widespread technology adoption is the ease of use, which must be addressed early in the architectural design to create a simple and minimal user interaction surface. Successful existing technologies such as Wi-Fi and Bluetooth utilise a 3-step process: list, select, confirm. Taking Wi-Fi for example, a user navigates their system interface to list the available networks, selects their desired network, and confirms by entering the password to authenticate. Compared to Bluetooth, the user navigates the system interface to the list of available devices, enables pairing mode on both devices, and confirms the same sequence of numbers is displayed on each device (or enters the appropriate pin). Thus, the three key concepts of a user-friendly list of available options, a simplified connection process, and user-verifiable authentication are essential to widespread adoption.

4.3 Peers

Each storage-capable node is defined as a peer, and there is no provision for a non-storing peer. Thus, each node in the network must be capable and willing to store data, eliminating the potential for nodes to “leech” and only ever access data. This ensures that all nodes collaborate to store data, effectively eliminating the possibility of a network of peers where the data is solely stored on one peer. This would be detrimental to the total storage capacity of the network, as well as eliminating the advantage of distributed data storage for both efficiency and resiliency. Furthermore, it simplifies the design and implementation of the protocol, as there is no distinction in function from one peer to another.

4.4 Network infrastructure

Some surrounding network infrastructure is assumed to be present, though further requirements upon the infrastructure is not presumed. For example, there is no distinction between IPv4 and IPv6 connections between clients, and all clients should be supportive of at least one, preferring IPv6 where possible. Furthermore, routing capabilities are not assumed, although hostnames are preferred where possible. This ensures that the system can operate on networks managed at an enterprise level, through to unmanaged home consumer networks, through to link-local connections between two peers. As such, peers should expect to be connected to, and connect from, any available network interface, using any IP version, by way of hostname or address. This further extends the flexibility of the system, ensuring that it can be adopted in several different environments.

4.5 Continuous discoverability via advertising

A key aspect of configuring communication services is the requisite parameters used to establish an initial connection. This task can be effectively removed from the user's responsibility by enabling each service to advertise itself and methods to connect to it. Both Wi-Fi and Bluetooth have such advertising methods in-built to their architecture, such that users need only select the desired network or device from a list of human-readable names. Wi-Fi is typically discoverable by default, with an optional hidden mode, whilst Bluetooth is not discoverable by default, requiring explicit user action to enter a discoverable pairing mode. As devices are expected to be non-portable and connections should be long-lived, it is suitable for each peer to be continuously discoverable, with explicit user action required to allow for a new pairing.

4.6 Automatic connections

Furthermore, as peers are expected to be non-portable, they should always be online and available for connections. As such, connections should be automatically established if a paired peer is discovered. Especially as there is no design limit for how many peers can be interconnected at once like with Bluetooth, the user should not be involved in explicitly initiating a connection or disconnection between peers. The user should be able to turn on and off the peers at will, and connection and disconnection is automatically incorporated into these events. If peers continue to be discoverable, and are paired, they should be connected, minimising the possibility of peers being paired and available not functioning as a whole system, which would cause undue confusion for users.

4.7 Minimising state in operation

To minimise complexity and test coverage, as well as potential corner cases, the system is designed to be stateless during operation. Initial setup processes, such as pairing and connections, are necessarily stateful, however subsequent operations are stateless and/or contained within a single actionable operation wherever possible. This enables the system to be relatively stateless in general operation, such that it should not accumulate errors over time, nor require regular restarts. Minimising the possibility of errors through architectural design is key for the ongoing usability and adoption of a technological system.

4.8 Minimal network structure

Peers do not maintain, nor attempt to build, an understanding of the overall network structure. Their only knowledge is maintaining a list of currently and directly connected peers, for which they can make immediate use of. Thus, the only possible inter-peer connections are either inbound, for which a request should be serviced or actioned, or outbound, where a request is expected to be serviced or actioned. Whilst this could lead to potential inefficiencies, attempts to rectify the situation with a more holistic view of the network would require either a governing peer to orchestrate connections flows, or for each peer to build their own view of the network and attempt to collaborate on a shared understanding. The former approach is undesirable due to the in-built centralisation, which undermines the working of a generalised distributed system. The latter would necessitate additional communication between peers that is not directly related to fulfilling user activities and is thus unnecessary and inefficient. It also adds potential complications and failure modes, which should be avoided in the architectural design of a system.

4.9 Design scalability

As the system does not distinguish between any two connected peers, it is theoretically scalable from as few as two peers, until practical and implementation-specific limits are reached. This further adds to the ease of adoption, as there are no specific requirements for number of peers, nor requirements on the amount of storage that each peer has or offers.

4.10 Limit of design scope

Whilst it is tempting to provide a complete system in its totality, it is equally important to limit the scope of the project such that it does not become too controlling or limiting of other implementations and aspects of computing. Hence, the design scope of the system minimises any portions that replicate the responsibilities of file system, particularly access control and permissions, as well as advanced techniques for efficient storage and management of data. The scope is intentionally restricted to network communication, and as little as possible on how that data is stored on each peer, as those aspects should be left to each implementation to implement in an optimal manner for their environment.

4.11 Transparent security

A final requirement to be included in the architectural design of the system is incorporating industry standard and up-to-date security algorithms for key communications. However, this security should be entirely transparent to the user, as correctly designing and implementing a secure system should not fall to the user's responsibility. As such, the system is designed to incorporate security between each peer, even from the first communication between two unpaired peers.

5 IMPLEMENTATION

A reference implementation has been developed alongside the design, to both inform and verify design decisions [1]. Whilst the reference implementation seeks to be feature-complete, it does differ from the design schema in several places, which will be noted. As such, its usefulness was primarily during development and testing of the system concept, and is not expected to be identically replicated in practice.

The implementation is written in Python 3.11, and designed to use open-source standard libraries and resources wherever they are available, to minimise development time and maximise system compatibility. Furthermore, the system should be compatible with Windows and Unix-compatible systems, and interoperable between them with no loss of features.

The network coding library was selected from amongst all open-source implementations of network coding on GitHub. The list was compiled using a search for "network coding", as well as all public repositories tagged with "network-coding" or "networkcoding". The available implementations were evaluated on several factors, including the ease of integration with the Python programming language, the difficulty of modifications and extensions if required, and the design purpose of the implementation. Several were eliminated due to incompatibility with Python. Others were only simulations of network coding for testing and evaluation, and not fit for integration into a working environment.

The "simple-nc" library was chosen as it is native Python code and thus fully interopable [6]. It was also intended to be used as a network coding library within a working environment. Finally, it was a feature-complete RNLC library, and only required minor extensions to be fully integrated into the rest of the reference implementation. Overall, this enabled effective development of other areas of the reference implementation.

5.1 Initialisation

The system can be split into three separate modules: storage backend; network communication; and service advertisement.

The storage backend is used for communicating with the backing file system. It stores file data as encoded symbols, file metadata, as well as persistent data for the peer, such as the public/private key pair for encrypted communication. For Windows system, the Windows Projected File System was used to intercept file operations and present files to the user and applications in a seamless manner. For Unix-compatible systems, the Filesysem in USErspace (FUSE) can be used to present a mount point for the user and applications to interact with. This module is the first to be initialised upon starting the system, as any errors with it would prohibit the rest of the system from functioning correctly. It is also imperative that data integrity is maintained, as attempting to continue instead could lead to data corruption.

The network communication module consists of two parts: a TCP server; and a UDP listener. The TCP server is used for most file operations, and encrypts traffic using the public/private key pair generated by the system. This TCP server should be available on all network interfaces, on any network address, and respond to IPv4 and IPv6 clients. The UDP listener is used for transferring file data using a symmetric encryption key established for that connection session. It should be available on all network interfaces, on any host address, and respond to IPv4 and IPv6 clients. This module must be fully operational for any useful network communication to occur, and so must be intialised and ready before the service is advertised.

The service advertisement module is the last to be intialised, as it signals that the system is fully operational and ready to engage in communication with peers. Additionally, ceasing the advertisement of the service can be used to indicate that it is stopping, and that relevant peers should disconnect. This ensures that peers will only attempt connections to a ready and working service, and that any errors or faults of one peer do not affect peers by disconnecting as soon as possible and rejecting further communication. The only responsibility of this module is to provide and maintain a DNS-SD or mDNS service listing. The fields and properties of this advertisement are described and specified in the next section.

5.2 Advertisement

The first stage of network communication is to advertise the availability of the system, which is performed once it has initialised all systems and is ready to accept connections. Advertisement is performed by registering a service via DNS-SD or mDNS, which is an industry-standard implementation of zero-configuration networking [2, 3].

DNS-SD requires that services are categorised by a combined type and protocol. Each device that advertises must also provide a uniquely identifying prefix.

Field	Value
Service	_adar._tcp.local.
Name	Hostname | Service
Port	6780
Server	Fully-qualified domain name
Addresses	IPv4 | IPv6

Table 1: DNS-SD and mDNS advertised fields with values

The service field contains the unique categorising identifier for this system, both by name and by protocol type. The top-level domain is the local domain, as this is required for all local area network services.

The name field is a concatenation of a user-friendly device hostname along with the service field. The concatenation must be performed with a '.' delimiting character, as with DNS domain names. The hostname will be presented to the user during pairing and connections, and so it should be user-readable and correctly identify the peer. It is acceptable to use the hostname as defined by the host OS.

The port field is specified to use a text representation of 6780. This port was chosen as it is currently unassigned by IANA.

The server field must be a fully-qualified domain name that can be used to connect to the peer.

The addresses field must contain a list of IPv4 and IPv6 addresses that can be used as a fallback to connect to the peer, if the fully-qualified domain name or hostname cannot be used appropriately.

Along with the direct fields specified in Table 1, custom text properties are also advertised as specified in Table 2, which include key information for the inital pairing and connection process.

Property	Value
Description	Distributed storage system
Key	Base64-encoded public key
Versions	Comma-separated supported versions

Table 2: DNS-SD and mDNS advertised custom properties with text values

The description property should contain a brief description of the service, which may be presented to the user depending on which DNS-SD or mDNS service browser they use.

The key property should contain a base64-encoded text representation of a public key which can be used to encrypt TCP traffic to this peer. The key should be generated by the latest available secure libraries, and is left to the specific implementation to determine. It will be used as a public/private key pair for TCP communication, and to help mitigate a machine-in-the-middle attack via impersonation. The usage of this identifier will be specified in the pairing and connection process. This key should be generated once and stored securely throughout system restarts, though a total reset should cause it to be regenerated. The reference implementation does not encrypt TCP traffic, and uses a Universally Unique IDentifier (UUID) instead.

The versions property should contain a comma-separated list of supported version numbers. The peer should support communication with each of the listed version numbers, though no restriction is placed on maximising this list. Rather, it is preferred to maintain a minimal list to ensure that implementations are kept up-to-date and secure.

5.3 Pairing and connecting

After a peer advertises itself and is discovered, a compatibility check occurs using the advertised versions. If there are any compatible versions, the highest (latest) compatible version number will be selected for use. The protocol corresponding with this version will be used henceforth for all communication.

Once compatibility is established, a check occurs to confirm whether the peer has been paired. This check involves a SHA3-256 hash of the peer's hostname, public key, and supported versions. These are converted to their string representation and concatenated, with the resultant hash persistently stored on the host device. The peer's hostname is used as an identifier when associating hashes.

It is important to verify that the peer is not impersonating another peer by using their advertised information, hence several advertised parameters are involved in this process. This step also ensures that any changes in key peer information (even intentionally) require the explicit approval of the user via pairing again. For example, if the supported versions change, the user must explicitly acknowledge this via a pairing process, which should help mitigate against version downgrade attacks.

If the peer is not paired, explicit user confirmation of the pairing is requested. If this is confirmed, the pairing request can be transmitted to the other peer. A TCP connection is opened to the peer's fully-qualified domain name (FQDN) on port 6780 as provided in the advertised fields. Once the connection is established, the resolved address is confirmed to be amongst the advertised addresses to mitigate a machine-in-the-middle attack and ensure it is a direct connection. If the FQDN does not resolve to an address or connection otherwise fails, the advertised addresses are used instead, attempting IPv6 addresses first.

Table 3 lists the format of the Pairing command.

Value	Example
Pair command value string	1
Colon	:
Comma-separated list of versions	1,2

Table 3: Pair request command components

The peer can choose to ignore this request, which should timeout after 30 seconds, or respond positively or negatively. A declining response would be a 0, whilst a confirming response would use a compatible version number to use for the rest of the communication session. The TCP connection is kept open once pairing is confirmed, to facilitate further communication for this session.

Once pairing is confirmed, the SHA3-256 hash of advertised parameters should be persistently stored locally, as described above. This concludes the pairing process, and once complete the connection process immediately begins.

If connecting to a peer that has previously been paired, a TCP connection is established using the same process as for pairing. If the peer has just been paired, the connection is reused and persists throughout the session. Just as for pairing, the supported versions are sent and a compatible version is selected. If no version is found, or the connection request is refused, a 0 is returned instead.

Following a connection request, a Diffie-Hellman key exchange occurs to establish a secure shared key. This begins by sending a key request, along with the public key component encoded in base64. The format for this message is described in Table 4. The key is generated from MODP group 16, with a length of 1024 bits [8].

Value	Example
Key command value string	3
Colon	:
Base64 public key	SGVsbG8gdGhlcmUh...

Table 4: Key request command components

The peer shall respond with their own public key, without any encoding. This enables both parties to establish a shared key in a secure manner, though it is still vulnerable to machine-in-the-middle attacks. To aid in mitigating this, a similar mechanism to the Bluetooth Core specification is used: the first two bytes of the newly established shared key are displayed to the user in their decimal representation. This results in 6 digits which can be cross-checked manually by the user to ensure that they match. If they do not match, the user should be able to abort the connection. A mechanism to abort is not included in the reference implementation.

Once the TCP connection has been established and a secure key is shared, the UDP connection can be established on port 6781. The UDP connection should use the same protocol as the TCP connection, and equally prioritise the FQDN over choosing an IP address. This leads to an automatic process to ensure that all user data is synchronised between the two peers.

5.4 TCP command format

All TCP commands are UTF-8 encoded strings, which begin with a decimal value for the command being sent. All command contents are encrypted using the advertised public key of the peer to which they are being sent. A valid TCP command is concluded with an unencryted Unicode newline character.

The command decimal value is separated from the rest of the transmitted data by a Unicode colon character. Following this are the appropriate parameters and additional values as necessary, specific to each command, if any. Each parameter is separated using a Unicode Information Separator One character.

Table 5 lists all commands and their decimal byte values for reference. Note that Read, Data, and Write are transmitted over UDP and follow a different encoding format.

Command	Decimal value
Pair	1
Connect	2
Key	3
Sync	4
Ready	5
Disconnect	6
Create	7
Rename	8
List	9
Read	10
Data	11
Stats	12
Write	13
Remove	14

Table 5: Byte values (in decimal) for each protocol command

File paths are represented as UTF-8 strings, where the root point of the available files is denoted as '/', and subsequent directories are delineated by further '/' characters. There is no explicit distinction between file and directory, as the whole path is always used. Relative paths are not permitted. An example is provided in Table 6.

Encoded file path
/directory/subdirectory/file.txt

Table 6: Example file path

5.5 Synchronisation

After establishing a new connection session, the two peers must ensure that user data is appropriately synchronised between them. To do this, a listing of the root directory is requested of the remote side. The listed files and folders are compared to the local side, with directories and files created as necessary. Once a file is created, the contents are also requested. If the file already exists, the modification time is compared, with the latest file contents requested.

To start this process, a Sync request is sent, as described in Table 7. The peer should respond with a 0 byte if they are not able to continue with the initialisation, at which point a disconnection occurs. They may stall for up to 10 seconds until they have established a TCP connection, a shared key, and a UDP connection. Once this is confirmed, they should respond with a 1 byte to confirm their readiness.

Value	Example
Sync command value string	4
Colon	:

Table 7: Sync request command components

For each subdirectory, this process is repeated. Once complete, the peer is able to transition to a Ready state, and informs the peer of this as per Table 8. General file operations for user interaction can now occur.

Value	Example
Ready command value string	5
Colon	:

Table 8: Ready state command components

5.6 File operations

Several high-leve file operations are supported. They include: directory listing; file creation; path renaming; file stats; and path removal.

5.6.1 List. The List command requests a listing of the folders and files for a given path. It expects folder names to be delineated by the Unicode Information Separator One character. File names are also delineated using the Unicode Information Separator One character. Folders are transmitted first, separated from the files by a colon.

Value	Example
List command value string	9
Colon	:

Table 9: List operation command components

5.6.2 Create. The Create command informs peers of a user creating a file or folder for a given path. If a file is created, the type indicator is set to 0 and the seed is also transmitted. If a directory is created, the type indicator is set to 1 and seed parameter is 'None'. The peer sends an empty response as confirmation.

Value	Example
Create command value string	7
Colon	:
Path	/New folder
Type	1
Seed	None

Table 10: Create operation command components

Value	Example
Rename command value string	8
Colon	:
Old path	/New folder
New path	/My fancy new folder

Table 11: Rename operation command components

5.6.3 Rename. The Rename command informs peers of a user renaming a path. The old and new paths are sent as parameters. The peer sends an empty response as confirmation.

5.6.4 Stats. The Stats command requests file metadata information from peers. The given path is sent as the parameter. It expects four integer values in return: size; creation time; modification time; and access time. The time values are measured in nanoseconds since the epoch, 00:00:00 UTC on 1 January 1970, as an integer. The size is an integer count of the bytes of the file data. The peer should respond with the requested information as stored in its metadata.

Value	Example
Stats command value string	12
Colon	:
Path	/file.txt

Table 12: Stats operation command components

5.6.5 Remove. The Remove command informs peers of a user removing a path. Any children of the path should also be removed. The peer sends an empty response as confirmation.

Value	Example
Remove command value string	14
Colon	:
Path	/My fancy new folder

Table 13: Remove operation command components

5.7 Data storage

The key innovation lies in how user data is stored and transmitted. Network coding is applied to the user data for transmission, as described by the UDP command format in section 5.8. However, network coding is also separately applied to the user data for persistent storage. The encoded symbols are stored, along with the pseudorandom number generator seed required to generate the corresponding sequence of coefficients for the persistently stored symbols. This process can be known as storage coding, as it applies encoding to data for efficient storage.

Once the user data is encoded as symbols, it can be effectively apportioned, such that each peer only stores a subset of the total number of symbols. This allows for the data to be easily distributed over an arbitrary number of peers, as well as recalled from an arbitrary number of peers without coordination.

Furthermore, as the symbols are stored in an encoded form, it is trivial for a peer to service a read request, as it only needs to read the bytes and transmit them. All the computational and memory burden for gathering enough symbols to decode, as well as performing the decoding operations, is done by the requesting peer. Thus, the distribution of load is only for each peer that needs to read data.

However, as each peer needs to generate and store a unique sequence of the symbols, when writing files, the entire unencoded file needs to be sent to each peer, and each peer encodes and stores a subset of symbols. Thus, the cost of writing is equally burdening for each peer.

The amount of symbols that each peer should store is to be determined by the peers themselves. If they have a significant amount of storage capacity available, they may choose to store more symbols, whilst if capacity is low, they may store less. In general, each peer should store a portion of a file's total size as symbols. This amount can be determined by Equation 1 describes a method of calculating it, where n is the number of connected peers.

$$\frac{n}{n+1} \tag{1}$$

As an example, with a network of only two peers, each should store approximately 0.67 of the total file size as equations. This would lead to a 50% reduction in file storage compared to simply replicating the complete file contents on each peer. It also only requires a third of the file size to be transferred to each peer at runtime to decode the whole file contents.

The precise equation and amount of symbols to store for each file is left to the specific implementations to determine and tune. It does not need to be consistent for all files, however. For example, files that are frequently accessed could be store in totality on the host, to avoid network traffic adding latency. Meanwhile, larger files accessed less often can be distributed amongst the peers and a minimal amount kept on the host.

This is a careful trade-off to balance, as storing too little could lead to data loss when not enough symbols can be retrieved from the network of peers. However, there is incentive to reduce the amount of symbols such that the total amount of storage capacity required is minimised.

Note that the reference implementation uses a fixed equation of $s \times 2$ where s is the file size. This ensures that even with only one peer in the network, the entirety of a file is guaranteed to be available, which enabled rapid and effective testing.

5.8 UDP command format

Data read requests are sent over a UDP connection, as well as data in response to a read, and data to be written. The initial read request is a UTF-8 encoded string and should be encrypted with the peer's public key as for TCP transmissions. It is terminated by an unencrypted newline character as well. However, it differs from the TCP tranmissions in that the selected command is not sent as a string representation, but rather as a single byte, whose literal value is equivalent to the command. All numbers are encoded as big-endian, regardless of host.

Table 14 details the format of the read command request. Note that a colon is not included, as it is not needed to separate the command from the path (the command is always a single byte). Parameters are separated by the Unicode Information Separator One character as with TCP, e.g. after Path, and after Skip.

Value	Example
Read command byte	0xf
Path	/file.txt
Skip	0
Symbols	10

Table 14: Read operation command components

Once a read request is received, the peer will read the requested number of symbols from the requested path. This is a simple byte-for-byte translation, where the Skip and Symbols parameters are equivalent to a file seek operation, and reading a specified length, respectively.

The actual transmission of data uses a different and completely unique format, which is discussed in detail in section 5.9.

5.9 Data transmission

The unique format for data transmission is sent over the UDP connection on port 6781. It is used for data servicing a read request, and for data that is to be written. When servicing a read request, the data is the encoded symbols as read from the persistent storage on the host. When the data is to be written, it is the complete file data without any encoding.

Regardless of the payload being delivered, the process applied to it is the same. The data is split into appropriately sized packets, e.g. 1024-byte chunks. The payload for each chunk is individually encrypted using the secure shared key established for that session. Once encrypted, the ciphertext is encoded as one symbol. This symbol, and associated metadata, is transmitted over the network as bytes. Each operation will be discussed in more detail separately.

5.9.1 Encryption. The XChaCha20-Poly1305 is selected as the latest and most secure symmetric-key encryption algorithm [14]. The key used for encryption is the last 32 bytes of the shared secure key that was established using Diffie-Hellman when initially connecting this session. As a new key is established for every connection session, the limit of 256 GB of data encryption is only for each session. A new and random 24-byte nonce is generated for each chunk of data to be transmitted, and an unlimited number of chunks can be securely sent each session.

The cipher is updated with the UTF-8 encoding of the file path, which is used as Additionally Authenticated Data. This can be used later to verify that the path has not been tampered with whilst in-flight. The non-symbol data is not encrypted, as this allows network devices to read and parse the packet, if configured to do so. This could enable these network devices to more efficiently transmit the data packets, as they are able to identify data flows, and can perform further network coding on the symbols whilst in-flight. This would not be possible if the entire packet were encrypted.

The outputs from the encryption step includes the ciphertext, as well as a 16-byte tag to ensure that the encrypted data has not been tampered with.

5.9.2 Network coding. The entire ciphertext is used to generate a single symbol. If the data has been split into several chunks, each chunk is encoded as a single symbol. For example, when transmitting 4096-bytes of encrypted data, it could be split into four 1024-byte chunks, which are encoded into four 1024-byte symbols. This results in a packet containing one symbol, as losing one packet results in one symbol being lost and requiring some retransmission.

The encoder uses a fixed pseudorandom number generator seed in the reference implementation, and there seems to be little motivation to change it dynamically. It could be possible to establish a fixed seed for a single session, such that there is some variability. This seed could be determined from the secure shared key as generated from establishing the connection.

Once the ciphertext has been consumed by the encoder, coded symbols can be produced. The manner in which they are produced is key to the performance of the protocol for larger file sizes. It is beneficial to begin with a sequence of lightly-coded packets, whose coefficients are not randomly chosen. In fact, the data can be transmitted byte-for-byte, as it is not expected that data losses should occur for most transfers. This allows for zero additional packet overhead when no packet losses occur, and also ensures that data is ordered accordingly. Thus, when the lightly-coded packets are received, they can be immediately decoded. Thus, their contents can be processed when only a part of the total symbols have yet been received. This is a variation on the otherwise standard implementation and usage of Random Linear Network Coding (RNLC), which may be called Semi-Random Linear Network Coding.

In ideal network conditions, where no packets are lost, it is equivalent to no encoding at all. However, when packets are lost, new symbols can be immediately generated and transmitted without delay. These newly generated symbols should follow RNLC, such that they contribute to multiple symbols, and can effectively account for the lost data.

The reference implementation does not support transmitting multiple packets of data, nor splitting the data into chunks. It also does not detect or support the retransmission of lost packets, though this would be trivial to add without modifying the protocol in any way.

5.9.3 Transmission format. Several additional pieces of metadata are required to be transmitted alongside the encrypted and encoded data. These are listed in Table 15.

This can be concatenated together and sent over the network as a UDP packet to the desired peer. The important benefit of the double-encoding scheme with encryption in between is that it enables network coding for transmission, whilst maintaining encrypted data for storage coding.

Value	Example	Length (bytes)
Data or Write command byte	0xb *or* 0xd	1
Path	/file.txt	variable
Unicode separator	0x1f	2
Total packets	0x1	4
Data length	0x256	8
File seed	...	8
Nonce	...	24
Coefficient	0x1	4
Payload length	0x256	2
Payload	...	variable
Tag	...	16
Unicode newline	\n	1

Table 15: Data transmission components

5.10 Disconnecting

When the service is about to stop, a disconnect service may be sent to preemptively inform peers and ensure a graceful exit from the network. After sending a disconnect command, the TCP connection can be closed for writing. After receiving a disconnect command, the TCP connection should be finished and closed. The peer is expected to stop advertising shortly, and once this occurs they can be removed from any state.

Value	Example
Disconnect command byte	6
Colon	:

Table 16: Disconnect state command components

5.11 Stopping

When stopping the service, the modules are stopped in the reverse order from their initialisation, and for the same reasons. If any peers are connected when stopping the network communication module, a Disconnect message is sent to each connected peer.

6 RESULTS

As the reference implementation is a proof-of-concept, it lacks effective optimisations, and so is not comparable in terms of compute or memory performance to existing protocols. However, as the protocol is only dependent on two different operations for reading or writing, the performance can be theoretically evaluated to determine any inherent limitations. A key limitation is file operations, such as reading and writing user data from the underlying storage system. This is dependent on the filesystem and Operating System chosen, as well as the storage hardware, and is irrespective of the networking protocol.

The other key limitation is the encode and decode performance for coding the user data. Existing network coding implementations can demonstrate effective performance already, using consumer-grade hardware [18]. Use of Single-Instruction-Multiple-Data technologies on mid-range consumer processors leads to significant performance improvements, whilst mobile phone processors are comparable to the baseline implementation. This shows that a suitably optimised implementation could be viable on consumer hardware, and even on mobile platforms.

Hence the only reasonable comparison that can be quantitatively made between the proposed system and existing solutions revolves around network overheads. Server Message Block 2 (SMB 2) is an existing solution to network file transfers, and is used for network overhead comparisons.

Table 22 details the components and size of a Server Message Block 2 NEGOTIATE request. Table 23 details the components and size of a proposed Connect request, which performs an equivalent function.

Component	Size (bytes)
Zero	1
StreamProtocolLength	3
Total	4

Table 17: SMB 2 Direct TCP header

Component	Size (bytes)
ProtocolId	4
StructureSize	2
CreditCharge	2
Status	4
Command	2
CreditRequest/CreditResponse	2
Flags	4
NextCommand	4
MessageId	8
Reserved	4
TreeId	4
SessionId	8
Signature	16
Total	64

Table 18: SYNC packet header

Component	Size (bytes)
StructureSize	2
DialectCount	2
SecurityMode	2
Reserved	2
Capabilities	4
ClientGuid	16
ClientStartTime	8
Dialects	2...10
Padding	0...8
NegotiateContextList	(variable)
Total	46

Table 19: SMB 2 NEGOTIATE request

Component	Size (bytes)
ContextType	2
DataLength	2
Reserved	4
Data	(variable)
Total	8

Table 20: SMB 2 NEGOTIATE_CONTEXT

Component	Size (bytes)
HashAlgorithmCount	2
SaltLength	2
HashAlgorithms	2
Salt	(variable) 16
Total	22

Table 21: SMB 2 PREAUTH_INTEGRITY_CAPABILITIES

Component	Size (bytes)
Direct TCP header	4
SYNC packet header	64
NEGOTIATE request	46
NEGOTIATE_CONTEXT	8
PREAUTH_INTEGRITY_CAPABILITIES	22
Total	144

Table 22: SMB 2 NEGOTIATE packet size

Component	Size (bytes)
UTF-8 command value	1
Colon	1
Supported versions	1...n
UTF-8 newline	1
Total	4...n

Table 23: Proposed Connect request components

As demonstrated by the this comparison, SMB 2 requires 144 bytes to negotiate a new connection, with supported encryption algorithms. The proposed system only requires 4 bytes, as it uses a single encryption algorithm for a given version. Both protocols have the potential for larger initial packets, if they support more versions and features. SMB 2 would add an additional 30 bytes (at least) for additional encryption algorithms. The proposed system requires at minimum, two bytes for an additional version.

Table 24 compares the sizes of various commands as specified by SMB 2 and the proposed protocol. This comparison effectively demonstrates the trade-off between simplicity and variability, as the proposed system is efficient, and relies on whole version changes for any variability. On the other hand, SMB 2 is highly variable and configurable for each connection, though it becomes bloated as a consequence.

It should be noted that file paths and data are excluded from the calculations, as these are both UTF-8 strings and will be variable.

Command	SMB 2 size	adar size
Connect	144	4
Create	140	25
Read	116	23
Write	116	69
Total	516	121

Table 24: Comparison of equivalent command sizes in bytes

However, SMB 2 utilises a FileId structure which is 16 bytes to describe a file or folder when performing read and write operations instead of a path string. The proposed protocol always uses the full path to avoided the statefulness of maintaining identifiers, though this will add to the listed size. However, as this comparison is focusing on the fixed overheads introduced by the protocols, it has been omitted in this case. Additionally, this is user-controlled and naturally variable, so is not useful to compare here.

Also not included is any actual file data, as this is inherently variable. The proposed solution encodes byte-for-byte and introduces no additional variable overheads if no packets are lost, thus it is not relevant for this comparison.

Overall, it can be seen that the proposed implementation is notably more efficient, as it utilises a simple architectural design that does not encapsulate packet data with layers of headers. In comparison, a SMB 2 packet may have as many as 5 layers of encapsulation, as demonstrated by the NEGOTIATE request in Table 22.

Furthermore, SMB 2 is an inherently client-server protocol, whilst the proposed protocol is fully distributed and decentralised. This aids in the reliability of the system, as it does not depend on any one particular peer, either for coordination or data. This also allows the user data to be easily distributed automatically, without user intervention to manually move data between different servers.

As the proposed system utilises network coding for storing data, it achieves a similar redundancy effect as can be achieved by RAID. However, as the peers are distributed physically across the network, and each have their own host device, they can protect against many more effects than RAID.

A RAID is capable of protecting against partial data corruption on a single drive, or a drive failure. It cannot protect against physical damage to the host device that affects all drives. For example, a malfunctioning host power supply, or a liquid damage flooding the entire host device. In comparison, the proposed system achieves redundancy by distributing the storage drives amongst several hosts, each of which can be physically distributed.

This minimises the affect that a single host failure has on the overall networks operation. If the RAID host is offline, the entire array is unavailable, meanwhile if a peer fails the proposed network should still be operation with the remaining peer devices.

Additionally, the user does not need to select the RAID configuration of the installed drives, nor match the drive capacity in any way. The proposed protocol can integrate into an existing host Operating System, as demonstrated by the reference implementation, since it does not require byte- or block-level access to the individual drives.

6.1 Limitations

Several limitations must be noted for the proposed protocol. Many of them are currently only applicable to the reference implementation, and should be resolvable by an improved implementation. Others are trivial and could be resolved by an updated protocol revision. Some are genuine restrictions of the design goals for a decentralised and distributed system, and are not easily resolved without additional complexity.

6.1.1 Network coding library. The chosen network coding library is a simple implementation in Python, and does not make use of the full performance of the processor. This severely limits the encoding and decoding performance. As previously mentioned, this is not a technical limitation of network coding, and can be effectively resolved by an improved implementation.

However, as part of this issue, the protocol specifies that each byte of a file is encoded as a symbol, which results in large generation sizes. For example, files larger than c. 300 bytes take over 10 seconds to encode on a modern processor (AMD Ryzen 5 3600). This appears to be a dependent factor for network coding implementations in general.

6.1.2 Reference implementation. The overall performance of the reference implementation is limited by the performance of the Python interpreter environment. This restricts the possible compute performance comparisons that could be made to the existing solutions of both SMB 2, which is typically part of the Operating System, and a RAID, which typically uses dedicated hardware.

Whilst the reference implementation is designed to be threaded, it is currently restricted by the Python interpreters Global Interpreter Lock system that prevents true concurrent execution on multiple threads. Hence, there is the potential for unforeseen race conditions to emerge when implemented with parallelism.

Furthermore, the reference implementation does not fully implement the protocol as described, hence unforeseen issues may arise when attempting to implement the complete protocol. Most key aspects have been implemented and tested in an attempt to avoid this situation, and to guide the design of the protocol in a flexible and permissible manner.

Finally, the reference implementation stores an overabundance of symbols for each file, such that a single peer is guaranteed to be able to fully decode the file. This is an obvious waste of storage capacity, and differs from the specified design of the protocol. The protocol has not been rigorously tested as designed, and though it should theoretically not introduce any errors, unforeseen issues could arise regarding the reliable availability of data to the user.

6.1.3 Data distribution. As the user data is inherently distributed across the network, access times to data will be limited by the latency of a Round Trip Time across the network. File read bandwidth will also be limited to the available network bandwidth, which is expected to be lower than that of internal storage drives.

Whilst it is possible to fully decode a file data after distribution amongst peers, it is dependent on enough symbols being retrieved from the peers. These symbols need to be sufficiently independent from each other to provide meaningful data whilst decoding, such that it is important to avoid identical replication of these symbols amongst the peers. There is no designed mechanism to perform this, and it is left to the implementation and individual hosts to determine a method of randomising which symbols they store. If not enough unique symbols are retrieved, the file will be corrupted when attempting to read the data.

A certain level of redundancy should be chosen by each peer, such that one or more peers may be unavailable and the full file data be retrievable. However, if too many peers are unavailable, or their data corrupt, then data corruption will occur when attempting to read the file data.

6.1.4 File system. The protocol as designed does not seek to implement any modern file system features, such as symbolic links or access control. These are ignored, and it is intended that any implementation or system will use the protocol as a base for this higher-level functions. Alternatively, these features are not expected to be utilised by a simple single-user system for general data storage, and so were not necessary to the reference implementation and design.

Compatibility with various applications and systems has not been extensively tested, with notable issues observed when using Microsoft Office products, such as Word. Their interactions with the file system are complex and not fully supported by the Windows Projected File System. Hence, they cannot be utilised with the proposed system during editing of the documents.

6.2 Future work

There is potential for an improved reference implementation that addresses the limitations of the current implementation. For example, implementing all design features, such as encrypting TCP data with a public-private key pair. An implementation optimised for compute performance with suitable parallelism would enable effective design guidance and enable additional comparisons to existing solutions.

Investigating techniques to improve the compute performance issues of network coding with large generation sizes would result in notable improvements to this application of network coding for storage. For example, optimising the implementation for large generation sizes with a fixed symbol size, or using larger symbols sizes such as 4 bytes each, or splitting larger files into several individually encoded chunks.

There are also theoretical and practical optimisations for the amount of stored symbols for each peer in a network. This could be set statically, or dynamically optimised based on several factors, including frequency of usage for particular files.

The protocol as designed is highly simplistic and does not seek to implement any file system features, such as symbolic links or access controls. Whilst these features are best left to the underlying Operating System and file system, it could be useful to communicate this attributes via the protocol.

Improving application compatibility for complex interactions with the file system, such as Microsoft Office documents, will improve the general usability of the reference implementation. This may require modification of the protocol as designed to support additional features and operations.

7 CONCLUSION

Overall, it can be seen that the proposed protocol provides effective file data transfers over a network, with a simpler configuration and implementation than existing protocols. It also provides equivalent and even improved features compared to traditional host-level redundancy solutions.

These improvements are possible due to the integrated architectural design and use of network coding for both transmission and storage of user data in a decentralised and distributed manner, along with automated management and operation, without any complex configuration requirements.

REFERENCES

[1] Craig Anderson. 2023. Network coding for automated distributed storage systems. https://github.com/theonlytechnohead/adar

[2] S. Cheshire and M. Krochmal. 2013. *DNS-Based Service Discovery*. Technical Report. https://doi.org/10.17487/rfc6763

[3] S. Cheshire and M. Krochmal. 2013. *Multicast DNS*. Technical Report. https://doi.org/10.17487/rfc6762

[4] Lauro B. Costa and Matei Ripeanu. 2010. Towards automating the configuration of a distributed storage system. In *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE. https://doi.org/10.1109/grid.2010.5697971

[5] Thomas M. Coughlin. 2017. Home Network Storage, the Cloud and the Internet of Things. In *Digital Storage in Consumer Electronics*. Springer International Publishing, 177–196. https://doi.org/10.1007/978-3-319-69907-3_9

[6] fisserL. 2022. Simple Random Linear Network Coding (RNLC) in Python. https://github.com/fisserL/simple_nc

[7] T. Ho, R. Koetter, M. Medard, D.R. Karger, and M. Effros. 2003. The benefits of coding over routing in a randomized setting. In *IEEE International Symposium on Information Theory, 2003. Proceedings*. IEEE. https://doi.org/10.1109/isit.2003.1228459

[8] T. Kivinen and M. Kojo. 2003. *More Modular Exponential (MODP) Diffie-Hellmean groups for Internet Key Exchange (IKE)*. Technical Report. https://doi.org/10.17487/rfc3526

[9] S.-Y.R. Li, R.W. Yeung, and Ning Cai. 2003. Linear network coding. *IEEE Transactions on Information Theory* 49, 2 (feb 2003), 371–381. https://doi.org/10.1109/tit.2002.807285

[10] Yunfeng Lin, Ben Liang, and Baochun Li. 2009. Priority Random Linear Codes in Distributed Storage Systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 11 (nov 2009), 1653–1667. https://doi.org/10.1109/tpds.2008.251

[11] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. 2015. A Self-Optimized Storage for Distributed Data as a Service. In *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE. https://doi.org/10.1109/wetice.2015.23

[12] Atsushi Nunome, Hiroaki Hirata, and Kiyoshi Shibayama. 2014. A Distributed Storage System with Dynamic Tiering for iSCSI Environment. In *2014 IIAI 3rd International Conference on Advanced Applied Informatics*. IEEE. https://doi.org/10.1109/iiai-aai.2014.135

[13] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data - SIGMOD '88*. ACM Press. https://doi.org/10.1145/50202.50214

[14] PyCryptoDome. 2022. ChaCha20-Poly1305 and XChaCha20-Poly1305. https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20_poly1305.html

[15] Himadri Nath Saha, Abhilasha Mandal, and Abhirup Sinha. 2017. Recent trends in the Internet of Things. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. https://doi.org/10.1109/ccwc.2017.7868439

[16] Kenneth W. Shum and Yuchong Hu. 2011. Exact minimum-repair-bandwidth cooperative regenerating codes for distributed storage systems. In *2011 IEEE International Symposium on Information Theory Proceedings*. IEEE. https://doi.org/10.1109/isit.2011.6033778

[17] Santar Pal Singh and Hirdesh Kumar. 2018. SSSFS: A Stackable Survivable Storage File System. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*. IEEE. https://doi.org/10.1109/ccaa.2018.8777719

[18] Chres W. Sørensen, Achuthan Paramanathan, Juan A. Cabrera, Morten V. Pedersen, Daniel E. Lucani, and Frank H.P. Fitzek. 2016. Leaner and meaner: Network coding in SIMD enabled commercial devices. In *2016 IEEE Wireless Communications and Networking Conference*. 1–6. https://doi.org/10.1109/WCNC.2016.7565066