SCAR - SCATTER, CONCEAL AND RECOVER

by

Bryan Mills

B.S. in Computer Science and Mathematics

Bethany College, 2001

Submitted to the Graduate Faculty of

the School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

SCHOOL OF ARTS AND SCIENCES

This thesis was presented

by

Bryan Mills

It was defended on

February 12, 2007

and approved by

Taieb Znati, Ph. D., Professor

Daniel Mosse, Ph. D., Professor

Ahmed Amer, Ph. D., Assistant Professor

Thesis Advisor: Taieb Znati, Ph. D., Professor

ABSTRACT

SCAR - SCATTER, CONCEAL AND RECOVER

Bryan Mills, M.S.

University of Pittsburgh, 2007

This thesis describes a secure and reliable method for storing data in a distributed hash table (DHT) that leverages the inherent properties of the DHT to provide a secure storage substrate. The framework presented is referred to as “Scatter, Conceal, and Recover” (SCAR). The standard method of securing data in a DHT is to encrypt the data using symmetrical encryption before storing it in the network. SCAR provides this level of security, but also prevents any known cryptoanalysis from being performed. It does this by breaking the data into smaller blocks and scattering these blocks throughout the DHT. Hence, SCAR prevents any unauthorized user from obtaining the entire encrypted data block. SCAR uses hash chains to determine the storage locations for the data blocks within the DHT. To ensure storage availability, SCAR uses an erasure coding scheme to provide full data recovery given only partial block recovery.

This thesis presents the details of SCAR. First, the framework, related protocols, and mechanisms are described. Second, a prototype implementation is presented showing the feasibility of SCAR. Third, analytical models are discussed that characterize SCAR’s behavior, the models are then validated using experimental results. Lastly, the models are analyzed to further understand the tradeoff between data security and data availability. The exploration of this tradeoff leads to the conclusion that SCAR can effectively balance this tradeoff when the nodes of the network are “sufficiently” available.

TABLE OF CONTENTS

1.0 INTRODUCTION AND GOALS 1
- 1.1 Introduction 1
- 1.2 Problem Statement 3
- 1.3 SCAR's Contribution 3
- 1.4 Methodology 4
- 1.5 Summary 5

2.0 RELATED WORK 6
- 2.1 Peer-to-Peer Networks 6
  - 2.1.1 DHT Security 8
- 2.2 DHT Availability 9
- 2.3 Data Reliability 10
- 2.4 Hash Chaining 10

3.0 SCATTER, CONCEAL AND RECOVER (SCAR) DESIGN 11
- 3.1 Data Scattering 11
- 3.2 Data Storage Interface 12
- 3.3 Packaging the Raw Data 13
- 3.4 Encoding Data into Storable Units 14
  - 3.4.1 IDA Explained 14
  - 3.4.2 Finite Field Arithmetic 15
  - 3.4.3 An IDA Example 17
- 3.5 Packing the Storage Bins 18
- 3.6 Scattering the Storage Bins 19
- 3.7 Data Recovery 20
- 3.8 Hash Collision 20

3.9 Immutable Units 21
4.0 ANALYSIS 22
4.1 Data Availability Model 22
4.1.1 Node Availability 22
4.1.2 Node Types 24
4.1.3 SCAR's Data Availability 25
4.2 Security Model 27
4.3 Tradeoff 30
5.0 IMPLEMENTATION AND EXPERIMENTS 35
5.1 Implementation 35
5.1.1 Details 35
5.1.2 Encoding and Decoding 36
5.2 Experiments 36
5.2.1 Simulator 36
5.2.2 Data Availability 38
5.2.3 Node Collision 39
5.2.4 Network Sensitivity Analysis 40
5.2.5 Likelihood of Recovery and Data Availability 41
6.0 CONCLUSION 45
6.1 Future Work 45
6.2 Summary 46
BIBLIOGRAPHY 47

LIST OF TABLES

1 Effect of increasing perceived capacity 32
2 Simulated average node availability 38
3 Network Sensitivity Analysis Settings 42

LIST OF FIGURES

1 Overview of SCAR's Encoding Process 13
2 Raw Data Header 14
3 Storage Bin Header 18
4 Storage Location Generation 19
5 State Diagram depicting ON/OFF model 23
6 Binary Variable over time 23
7 Difference between availability models 27
8 Comparison between replication and SCAR 28
9 Security Model varying perceived capacity, varying k 30
10 Security model varying perceived capacity, fixed k 31
11 Tradeoff model varying only k 33
12 Tradeoff model varying k and f 33
13 Tradeoff model varying f 34
14 Tradeoff model varying node unavailability 34
15 Compare Model with Experimental results 39
16 Compare Model with Experimental results 40
17 Average number of collisions varying number of fragments 41
18 Maximum number of collisions varying number of nodes 42
19 SCAR's sensitivity to peepers 43
20 Data availability as probability of checking decreases 44

1.0 INTRODUCTION AND GOALS

1.1 INTRODUCTION

Information is more accessible today than it has ever been in the history of mankind. Increasingly, we see libraries of information being digitized and replicated, providing knowledge availability beyond anything our previous generation could have envisioned. Modern communication is the great enabler in knowledge access. From the advent of the telegraph to the immense throughput of a modern network, the primary goal was to connect people to information. Just as the printing press made books available, distributed storage has made information available. We have come to expect that *any* information can be accessed from anywhere at anytime. In addition, we want to have easy access to *our* information, but much of this information is private and should be not be accessible to other users. This struggle between accessibility and security is a central theme in the modern world.

How can we provide a user with highly available information, and at the same time ensure the security of their private data? Current solutions to this challenge fall short of this stated goal, as they often rely on central authorities reducing accessibility and availability in order to achieve security. To fully address this shortcoming, one must integrate several technologies.

The Internet can be viewed as a distributed storage facility that provides access to a bewilderingly variety of documents and services. However, due to its scale and decentralized nature, the Internet has evolved into a chaotic repository of all types of information, making it difficult for its users to locate resources of interest to them. Recently, the Peer-to-Peer (P2P) overlay technologies have emerged to leverage resource discovery and sharing on the Internet [29, 21, 26, 31].

Peer-to-peer networks allow large-scale deployment of a self-evolving infrastructure in an ad-hoc fashion, without the need for centralized management or control. The ease of deployment of peer-to-peer overlay infrastructure enables the development of a new class of applications that can effectively and strategically provide services over the Internet. Distributed storage is a service often synonymous with peer-to-peer

networks. The common approach to distributed storage is through the use of a distributed hash table (DHT).

A DHT provides its users with the ability to store data at and retrieve data from peers within the network. The data is retrieved by querying the DHT for the data stored using a key value, which is specified when the data is stored. Like a hash table, the key values are generated by using a hash function. In the distributed hash table, each peer is assigned ownership for a given key space that is typically a range of possible hash values. As an example, consider storing a file in the DHT. It is likely that the key would be the hash value of the *filename*. The file is then inserted into the DHT based on this key; the peer with ownership of the key space is responsible for storing and managing access to the file. To retrieve the file, a query with the inserted key is made to the DHT. The responsible peer responds with the key associated file.

The DHT provides a mechanism for distributed storage; however, it does not satisfactorily solve the problem of data availability. While the DHT provides a way of storing data, there is no guarantee that the node responsible for the stored data will be available when data is requested. To provide availability in the DHT, the implementor can either replicate data at different nodes or encode the data across multiple nodes such that the data can be re-assembled despite the unavailability of some nodes. Data availability is an essential feature of distributed storage, but in addition data security must be considered.

The security of data being stored in the DHT has not been addressed adequately by the research community. Common approaches to data security in DHT-based peer-to-peer networks make the assumption that data encryption is sufficient to ensure data security. While encryption provides some level of security, this is not a complete solution. Encryption does not prevent an attacker from running brute force cryptoanalysis tools to gain access to the data. This becomes even more apparent in systems designed to provide long term storage, such as Oceanstore [16]. In these systems, given enough time any encryption could be broken.

Other data security solutions rely upon a central access authority to grant permission to authorized users. This solution violates the core properties of the peer-to-peer framework which seeks to provide a fully distributed storage system. Additionally, this suffers from a single point of failure. One solution to address this shortcoming would be to replicate access authorities. This however, suffers from single point of attack. For example, by comprising any one of the multiple authorities the data then becomes available to the attacker. Another major concern stems from the fact that a given data object is typically stored as a whole within a single node. Consequently, while the data may be inaccessible to the attacker directly, being physically located within a single node exposes it to a single point of attack. Any system that relies on central authorities or stores data within a single node suffers from a single point of attack.

The goal of this research is to address the aforementioned issues and provide a uniform framework that provides secure, reliable, and highly available distributed storage using peer-to-peer technologies. This

work was driven by the need to store user's private information in a distributed storage system. This thesis describes the problems faced while trying to build such a system, and presents a framework which overcomes these problems. This framework is referred to as scatter, conceal, and recover (SCAR). A prototype will be developed to demonstrate the feasibility of the proposed framework.

1.2 PROBLEM STATEMENT

Computer users are becoming increasingly more mobile, and require access to their information from a variety of locations. This has driven researchers and companies to build large distributed globally available storage. Each of these systems rely upon a dedicated infrastructure and are typically under the control of a single entity. The desire to create a fully decentralized system motivated the establishment of peer-to-peer networks. These networks provide a framework that gives users the ability to access data from anywhere and no longer requires a dedicated infrastructure nor does it require a central organization to provide this service.

Peer-to-peer networks rely upon multiple un-trusted entities that all participate to provide distributed storage. These peers are not required to meet any availability or reliability requirements. This means the storage facility itself is not reliable, available, or secure. Many researchers have investigated how to increase the reliability and availability of the data being stored, but data security has not received adequate attention. How can peer-to-peer networks establish data security while the storage medium itself relies upon multiple un-trusted entities? Can a secure peer-to-peer storage system be built that does not rely upon a central authority? If building such a system is possible, can the system meet its required availability and reliability requirements? The goal of this thesis is to gain better understanding of issues relevant to these questions and design a framework to enable efficient and secure access to data stored within a DHT-based peer-to-peer network.

1.3 SCAR'S CONTRIBUTION

A main contribution of this thesis is a novel way of securing data in an un-trusted distributed storage system that does not rely on central authorities or complex cryptography. The idea of securely scattering data across multiple un-trusted nodes while maintaining high data availability and reliability is unique in itself.

This thesis also provides an implementation of the framework and details the implementation solutions. Furthermore, this thesis provides analytical models that describe the security and availability of data being stored using SCAR. These models are then used to study the inherent tradeoffs between data security and availability and show how SCAR attempts to balance these competing goals.

The SCAR system was designed to meet the following goals:

- Provide data availability that is at least as good as that provided by any DHT implementation.
- Provide a higher level of security than that provided by data encryption.
- Maintain independence with respect to the underlying distributed storage system.

To achieve these goals, SCAR efficiently combines erasure coding and hash chaining to increase both availability and security. Erasure coding enables SCAR to provide high availability by splitting data into multiple pieces such that only a fraction of these pieces are required to assemble the original data. Hash chaining provides a way to deterministically conceal the data, by scattering the pieces over randomly selected peers. Concealing the location of the data pieces makes it extremely difficult for an attacker to gain access to enough information to reassemble the original data. As such, cryptoanalysis is no longer achievable, thereby increasing the security of the stored data.This combination creates an apparent random distribution of the data while providing an authorized user a deterministic way of gathering the data. SCAR can be adapted to any DHT implementation since hash chains can be produced using any hash function. This is an elegant solution to the difficult problem of providing highly available and secure distributed storage.

1.4 METHODOLOGY

The basic tenet of SCAR's design is to balance data security and availability. The intersection of peer-to-peer networks, erasure coding, and "trackable randomness" was used to achieve the goals of the system. SCAR does not modify the peer-to-peer structure, but simply acts as a DHT user. Erasure coding was used to provide data availability and adapted for use within a DHT structure. The system had to track an apparent random process, this required several simulations resulting in the use of hash chains. The final step was to combine these technologies into a unified system.

During the design phase, availability and security models were developed to determine if the design was beneficial. Experiments were then conducted to determine if our models accurately described the system

behavior. These experiments focused on how erasure encoding and hash chaining performed within the peer-to-peer network.

Once the design was defined, a prototype was developed that implemented the entire system and allowed for more testing. The prototype went is written in Python and uses the OpenDHT [23] interface.

1.5 SUMMARY

The need for highly available private data has prompted the development of a framework that utilizes DHT's to provide a secure distributed storage system. The framework glues together three different technologies, peer-to-peer networks, erasure coding, and hash chains, to create an elegant and unique solution. An implementation of this framework was built and the availability and security tradeoffs were analyzed. This implementation is then provided as a library that an application developer can use to provide secure distributed storage.

In chapter 2, we provide a background on peer-to-peer technologies. In chapter 3, we present the design of SCAR and provide the details of the system. In chapter 4, we describe the models used in the analysis of SCAR. We provide implementation details in section 5.1. We then present various experiments conducted using SCAR in section 5.2. Finally, we provide concluding remarks in chapter 6

2.0 RELATED WORK

2.1 PEER-TO-PEER NETWORKS

A peer-to-peer network enables two or more peers to share access to resources without requiring a separate centralized management authority. Each peer has some capability that it then provides to other peers in the network. Collectively the peers can provide a variety of services, including routing and storage. While the deployment of peer-to-peer network services is relatively new, the concept of peer-to-peer has been around since the early days of computing. Examples of these peer-to-peer applications include email and irc servers. These servers act as peers to one another, each with a common goal of providing services to the network.

Often modern peer-to-peer networks are described by their communication behaviors, specifically how messages are routed within the network. Peers are connected to one another using some physical network. When a peer discovers another peer, there is said to be a direct connection between those peers. Relying upon each other, peers can route messages within the network using multi-hop routing. This leads to what is known as a peer-to-peer overlay network; these networks are classified into un-structured and structured networks.

Un-structured networks allow nodes to be located anywhere, and impose few or no rules about how the nodes are connected. A node can create connections to other nodes arbitrarily, creating a random graph. Because nodes are unaware of the structure, queries are executed by flooding the request to every node in the network. When the query arrives at a node it first determines if it can answer the query, and if so it replies to the requesting node. The node then forwards the query to all other directly connected nodes. Because un-structured networks rely on flooding, query execution is slow and might be incomplete. Queries may not complete if the network becomes disconnected or if the query takes too long causing a query timeout. Popular un-structured networks are Napster, Gnutella [1], and KaZaA [2].

Structured peer-to-peer networks are designed to overcome the problems of un-structured networks. Structured networks maintain a distributed hash table (DHT), which is used for both data queries and net-

work routing. Each peer is responsible for managing a given part of the network, corresponding to a range of hash values. Any request for data in this hash range is handled by the node responsible for that hash value. This idea stems from the influential paper by Karger et al. on consistent hashing and web caching [15]. Researchers have explored various designs for DHT's, but the fundamental idea of using a hash function to distribute data and nodes throughout the network has changed very little. While there are several different structured peer-to-peer networks, such as Chord [29], CAN [21], and Pastry [26], each of these networks follow the same general architecture. They each divide the hash space into a range of values that are assigned to participating nodes. Then data is mapped to a single hash value which is used to determine the node that is responsible for that data.

Most structured overlay networks separate the functions of routing information from the actual data storage. For example when a query is submitted to the DHT the network must first determine which node should answer the query, this process is known as routing. Routing messages to responsible nodes have applications outside of a DHT, as a result, the routing functions are usually built independently of the DHT. A DHT, combined with routing functions creates a distributed data storage. The typical interface [9] for the DHT is *put* and *get* functions that allow the storage and retrieval of data within the network. Data requests are routed to the responsible node using the routing layer, and the data storage is handled independently by each node.

Researchers have used DHT's as an underlying architecture to provide complete filesystems or object repositories. Systems such as the Cooperative File System (CFS) [10], uses Chord's DHT structure to manage a filesystem. Several other DHT based filesystems [6, 19] have sought to solve issues such as metadata storage, mutable storage, availability, and latency issues. Additionally, RAMA [18] a distributed file system that pre-dates DHT's, also uses a hash function to pseudo-randomly store filesystem data blocks. These systems assume that security would be handled by a central authority or data encryption. SCAR can be used to address these shortcomings as it provides the basis to implement a system that is secure and truly decentralized.

SCAR assumes a structured peer-to-peer network, and uses the associated DHT to provide distributed data storage. SCAR does not depend on any particular DHT implementation, but assumes that a *put/get* interface is available. SCAR's contribution is to design a framework to securely and reliably store data using the underlying DHT of the peer-to-peer network. In the rest of this section, we will discuss peer-to-peer design issues, including the current state of the art DHT security and availability.

2.1.1 DHT Security

A DHT relies upon two different components, peer-to-peer infrastructure and the storage facilities. The underling peer-to-peer infrastructure, determines the routing and address space responsibilities to be performed by each peer node. The second component is the storage service provided by each node, managing its assigned address space. The combination of these two components provides a distributed data storage within the peer-to-peer network, referred to as a DHT. Because the DHT involves both of these components, security of data within the DHT must consider both components.

The responsibility of nodes to manage their allocated address space and perform routing operations is assigned when a node joins the peer-to-peer network. This assignment can easily be achieved if all nodes were trusted, as it does not raise security risks. However, if the nodes joining the network are malicious, the routing and space management assignment become more difficult [5, 28]. The ability of nodes to join and then leave the network, referred to as node churn [22], can lead to a potential security attack, known as the join/leave attack. Such an attack occurs when nodes rapidly join and then leave the network, causing reassignment of the address space to each node. This rapid oscillation can lead to the collapse of the network. Another class of attacks, known as Sybil attacks [12], occur when one malicious user is controlling multiple nodes. If an attacker controls multiple nodes within the network then the attacker might be able to interfere with network routing, by dropping or mis-routing messages. Routing attacks, such as Sybil attacks, can occur because the network depends on all of the nodes for routing, so even a single malicious node can cause routing to fail. In addition, denial of service attacks are easily carried out by flooding the network with messages addressed to the same node. Many of these attacks become very hard to guard against and only pragmatic solutions exist that limit, but not completely disband, the threat. There have been several peer-to-peer frameworks developed to address these security issues [11, 3, 4].

Maintaining the DHT structure depends upon the reliability of the underlying peer-to-peer infrastructure; as a result, the security of the infrastructure components are a requirement for a secure DHT. However, data correctness and protection are issues that must be handled by the storage facilities. Data correctness is the ability to both ensure and verify that data retrieved from the DHT is the same as the data inserted. Fortunately, data correctness can be verified using various cryptographic techniques. Data protection is the requirement that stored data can only be accessed by authorized users. Data protection becomes a problem when untrusted nodes are made responsible for storing data that needs to be protected.

While there has been significant work in the area of peer-to-peer security, data protection in the DHT has been largely ignored. The assumption has been that if the data protection is needed, it will be protected

before insertion into the DHT. Typically this would involve the encryption of the data prior to insertion it into the DHT. This solution might not be adequate because attackers can perform cryptoanalysis to break the encryption. Such an attack can easily be carried out because all encrypted data is available to everyone in the DHT, including the attacker. Other solutions to this problem introduce a central authority that is used for authentication and authorization. A central authority imposes design constraints that are hard to securely implement and manage over time. Additionally, these approaches force a distributed system to rely upon a central service.

SCAR provides a data protection solution that is more secure than encrypting data, and does so without the use of a central authority. SCAR conceals data in the DHT using a password seeded hash chain to deterministically scatter data and prevent unauthorized users from locating the stored data.

2.2 DHT AVAILABILITY

The ability to secure data in the DHT does not guarantee that data will be available when requested. Data availability is a challenge because data is stored at nodes that may join or leave the network at anytime. This fluctuation in node availability is referred to as *node churn*, and has been the focus of research both for DHT-based [22] and other types of peer-to-peer networks. In addition to node churn, data availability is also concerned with malicious nodes not providing storage that they have agreed to provide, therefore making data unavailable.

The common solution to the node churn problem is to replicate the data so that when retrieving data, there are several available location for finding it. This solution, while simple, is not efficient in recovery time or storage requirements. Replication may cause recovery time to become excessive as it relies on a timeout mechanism to determine if nodes have failed. Because of these shortcomings one could use erasure coding to encode the data into multiple pieces such that only a fraction of the pieces is required to fully recover the data. Weathersppon and Kubiatowicz present the argument for using erasure codes instead of replication [30]. SCAR uses Rabin's Information Dispersal Algorithm (IDA) [20], which is also used by Freenet [7] and Mnemosyne [14]. Erasure coding is much harder to implement and can cause greater maintenance overhead [25]. All DHT implementations using erasure coding store data pieces at easily predictable locations, such as neighboring nodes or node ancestors. These locations are known to all users of the network and therefore provide no additional security. However, SCAR conceals the storage locations from everyone but authorized users, thereby providing greater data security.

Also related to availability is the fact that malicious nodes might not keep the data they have been requested to store. This problem has been addressed with random data challenges [8] to nodes in the system to verify they are storing the data they claim to have stored. This checking system is able to detect malicious nodes and enforce counter measures, such as high levels of replication or ejection of that node from the network.

2.3 DATA RELIABILITY

Reliability of data refers to the fact that the data that is retrieved should be identical to the data that was stored. For example, the data could have been corrupted while in the storage system or might have been modified by a malicious user, in either case the data was not reliably stored. This differs from availability because a storage system that is available 100% of the time but always produces the wrong data is available but not reliable. Most DHT's address the problem of data reliability by using a cryptographic message digest, such as a MD5 or SHA-1 checksum. This technique allows the user to detect data corruption because upon retrieval the stored checksum can be compared with the checksum of the retrieved data. If a malicious user is trying to corrupt the stored data this technique fails because the malicious user could update the stored checksum. This problem is addressed using digital signatures [14] which allows users to detect the corrupted checksum.

2.4 HASH CHAINING

Hash chains have their roots in providing user authentication [13] and secure micropayments [24]. To use hash chains for authentication, a series of passwords are calculated from a single password. Each login attempt requires the next password in the series. To create this series, the root password is hashed repeatedly creating unique passwords. SCAR uses this technique to generate storage locations for the pieces of data to be stored in the DHT. Because the hash chain is seeded with a secret password only users with that password can determine the storage locations. This property is what allows authorized users to track an apparent random distribution of data pieces.

3.0 SCATTER, CONCEAL AND RECOVER (SCAR) DESIGN

The main tenet of SCAR is based on the observation that it is harder to break encrypted information if the attacker has no way of obtaining the encrypted data. SCAR achieves this using a simple, yet powerful concept of concealment through random distribution. The goal is to break data into pieces and randomly distribute the data throughout the network so that only authorized users can locate the pieces and recover the original data. Hence, our approach is to scatter data such that its locations are concealed, and ensure that only authorized users can recover the data that was stored.

A system to randomly distribute data faces several crucial design challenges. First, how to track the untrackable, meaning how can we randomly distribute data so that the attacker can not find it, but yet make the data easily accessible to authorized users? Second, how do we embed such a system within the context of a DHT? Also, how can we ensure high availability in the face of node failures and potentially high node churn? Finally, how can we build a system that provides a simple interface such that the scattering, concealing, and recovering data is transparent to the user of the system?

3.1 DATA SCATTERING

Many structured DHT's apply a hash function to some unique node values (such as network address) to determine the node's location within the peer-to-peer network. The hash function not only guarantees a random node distribution but is also deterministic. Therefore, using a hash function to generate random but also reproducible storage locations was adopted in SCAR. The need to have storage locations unknown to an attacker prompted the idea to seed the hash function with a secret password to produce the storage location. This approach only leads to one storage location; hiding the data in one location, however, this does not ensure the same level of security as breaking the data into many pieces and hiding them in multiple locations. To generate multiple storage locations, SCAR uses the concept of hash chaining [13]. Based

on this concept, SCAR uses a given hash value and repeatedly runs the hash function to generate multiple random locations. The use of hash functions to generate storage locations, makes a DHT a natural fit for SCAR, as DHT storage locations are already defined as a hash value.

To address the problem of availability, SCAR makes use of the erasure encoding schema called information dispersal algorithm (IDA) [20]. IDA splits a block of data into multiple pieces such that only a fraction of those generated pieces are needed for data recovery. Formally, IDA generates f pieces of data such that only k pieces are needed for recovery, such that $k < f$.

Information dispersal algorithm (IDA) was chosen because of its ability to produce f distinct pieces of data from a data block of any size. In contract, other erasure codes such as Reed-Solomon only produce a small set of fragments for a given block size. Other codes, such as Tornado, are focused on reducing computation speed and do not guarantee distinct pieces.

Upon breaking the original data into multiple pieces using erasure coding, SCAR must scatter these pieces throughout the DHT. This is required for SCAR to meet security and high data availability as specified in it's design goals. As nodes leave, SCAR is still able to recover the stored data, since not all data pieces are required to recover the original data.

3.2 DATA STORAGE INTERFACE

Users of a distributed storage system should not be concerned about how their data is stored. Furthermore, this calls for a simple user interface to store and recover data. Driven by this requirement, SCAR provides support for a simple save and load (put/get) interface, which relieves the user from the burden of dealing with how their data is stored in the network. To meet it's design goals, SCAR automatically selects the system parameters to achieve the security and availability requirements.

SCAR first adds header information to the raw data, and then it breaks the data into a number of pieces called *storable units*. Each of these storable units are then placed into individual *storage bins* that contain additional header information used during the recovery process. To determine the storage location of each storage bin, a hash chain is started using the name of the data, a password, and the version number of the data. The hash chain is then produced by re-hashing the previous hash value along with the secret password. Each hash value produced by this hash chain becomes a series of storage locations for these storage bins. See Figure 1 for a overview of the encoding process. To recover the data, the user must provide the name of the data, a password, and a version number that SCAR will use to re-generate the hash chain, thus obtaining

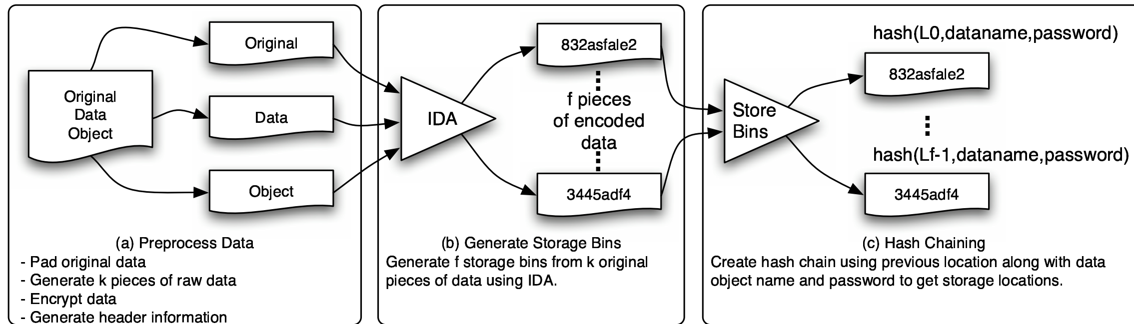

Figure 1: Overview of SCAR's Encoding Process

the storage locations.

3.3 PACKAGING THE RAW DATA

As can be seen in Figure 1 (a), there is some pre-processing that must occur before the raw data can be encoded into its storable units. Pre-processing will ensure that the data is large enough for encoding, perform encryption, and add a header for data verification upon recovery.

Since the encoding process divides the raw data into smaller pieces, we need to first make sure the raw data is of sufficient size to support this process. If the data is smaller than predefined size, then we need to pad the data making it that size. When padding the data, we want to require that all of the data, including the padding, is required for data recovery. To obtain this property, SCAR uses secret sharing techniques [27] to generate the padding.

Assume we need at least 100 bytes to encode and we have raw data of only 25 bytes. SCAR would generate three more 25 byte random pieces of data. The last piece of data would be the XOR of each of the random pieces and the original data. By stringing each of these generated pieces of data together, we will have the desired 100 bytes of data. To recover the original data, one must identify each generated piece of data and XOR each piece together. The result of this operation is the original data. Every generated piece is needed to recover the original data, meaning we need all 100 bytes to recover the 25 bytes of original data.

After we are assured the raw data is large enough, we construct a fixed size header depicted in Figure 2. After the header is added the data is encrypted using symmetrical encryption.

file size (64)	
SHA-1 Hash (40)	padding size (10)
... encrypted data ...	

Figure 2: Raw Data Header, numbers represent size in bytes

3.4 ENCODING DATA INTO STORABLE UNITS

Once we have the data ready for storage, SCAR must split the data into multiple pieces and encode the data to ensure data availability, as can be seen in Figure 1 (b). To do this, SCAR implements Rabin's Information Dispersal Algorithm (IDA) [20] to encode the data into storable units. This encoding produces f storable units and allows full data recovery with only k recovered units, where $k \leq f$. Values for f and k are dependent upon the availability requirements for the data being encoded. Chapter 4 discusses the selection of values for k and f based upon tradeoffs between availability and security. Once values of f and k are chosen, we execute IDA on the pre-processed raw data. Details on how our IDA implementation works can be found in the next section. Each piece that is generated from this encoding process is unique and of the same size.

3.4.1 IDA Explained

The main concept behind IDA is to split a given data object into f different pieces such that only k pieces are required for regenerating the original data object, $k \leq f$. To achieve this property, IDA uses principles of matrix multiplication to build a transform from k original pieces of data into f pieces of encoded data. The transform is built using a linearly independent matrix so that the inverse of that transform matrix is guaranteed to exist. Then given any k pieces of the encoded data and the inverse transform, it is possible to regenerate the original data.

Let the original data be represented as a series of bytes of the form D=b_1,b_2,b_3...b_n. Create a matrix of these bytes by slicing the data into n/k pieces and stacking them on top of one another. The next step is to generate the transform matrix of n/k columns and f rows. As mentioned earlier, this matrix must be linearly independent, meaning that none of the rows can be generated by linear manipulation of another row

in the matrix. This guarantees that the inverse matrix exists, allowing the reversal of matrix multiplication. There are several ways to generate this transform matrix. SCAR uses a simple algorithm and is guaranteed to generate the same transform matrix given the values of f, k, and n. By using the same transform matrix each time for these given values, the transform matrix does not have to be encoded within the storage bins.

Once we have the transform matrix, the original data matrix and the transform matrix are multiplied together. This will generate a new matrix with n/k columns and f rows, with each of these rows corresponding to one of f encoded pieces of data.

To recover the data given any k pieces of the encoded data, it is required that one knows the index of the k pieces within the context of all f encoded pieces. Then knowing the order of the encoded pieces, the decoder must also have the rows corresponding to this same index within the original transform matrix. As mentioned earlier, SCAR uses the same transform matrix for all given values of f, k, and n. The order of the recovered k pieces is also known to SCAR because the hash chain used to generate the storage locations are known. Once the proper subset of the transform matrix is known, the recovery process must generate the inverse transform. The recovered encoded data is then multiplied by the inverse transform matrix producing the original data.

One problem arises from the fact that the f encoded pieces might be larger than the original k pieces. To resolve this problem, we perform all math within a finite field. Specifically, this transform process uses a prime finite field, also known as a Galois field. Essentially, we perform all math modulo a prime number. This forces all of the cells in the encoded matrix to be less than the prime number. In our implementation we want all cells to be 1 byte, so we choose the prime number 257. This is described in detail in the next section.

3.4.2 Finite Field Arithmetic

A problem with the schema described above is that each of the f encoded pieces will not be the same size as an original k piece. This is because each cell in the original data matrix represented 1 byte, but because of the transform, the cells in the encoded matrix might be larger than 1 byte. This is due to repeated mathematical operations caused by the matrix manipulations resulting in numbers that might be greater than those input. The desired output is a series of bytes, and to accomplish this, all arithmetic operations must be bounded by the size of a byte. Because of this requirement, all arithmetic operations are performed using finite field arithmetic.

Finite field arithmetic is different than standard arithmetic. The finite field is a limited list of numbers

and all arithmetic operations result in a number in that finite list. By using finite arithmetic during all mathematical operations, we can be assured that the output from IDA will never exceed the size of the input. In particular, IDA uses a prime finite field, also called a Galois field (GF). A prime field is an example of a special type of field known as a ring in which mathematical operations are closed under that same field. Prime fields are denoted $GF(p)$, where p is a prime number and the field consists of the set of integers from 0 to $p-1$. An extension field is denoted $GF(p^r)$, where p is a prime number and $r > 1$. Prime fields and extension fields are both closed under arithmetic operations.

To make finite field operations efficient on a computer, let $p = 2$, so that mathematics can be achieved using bit operations. For implementation purposes we let $r = 8$, which makes the field size 256, allowing the computations to always produce a byte size integer. We need to define addition, subtraction, multiplication, and division in order to perform the matrix operations needed for IDA.

Addition and subtraction are defined as the XOR of the two values. Notice that properties of adding and subtracting still hold, but all applications of the operation always result in a value inside the finite field.

Multiplication is based on primitive polynomials. These are polynomials that cannot be expressed as the product of any two elements within the finite field. A primitive polynomial is also called an irreducible polynomial because it cannot be reduced by elements within the finite field. As an example, consider $GF(p^r)$, where $p = 2$ and $r = 8$, one irreducible polynomial is $x^8 + x^4 + x^3 + x^2 + 1$, which can be represented by the bit string 100011101, where the nth bit location corresponds the nth polynomial location. Multiplication is then defined as the polynomial multiplication of the two polynomials corresponding to the two values being multiplied modulo a primitive polynomial for that finite field.

By defining the *log* function within the context of the prime field, multiplication and division can be simplified to addition and subtraction, reducing them to bit operations. The definition of log is based on the property that for any prime field, there exists an element α that belongs to the finite field, and for every element n in the finite field, there exists a value i such that $\alpha^i = n$. We can then define $log(\alpha^i) = i$. With this definition, we can re-define multiplication as $x * y = \alpha^{(log(x)+log(y))mod(q-1)}$, where $q = p^r$. Armed with this definition, we can define the inverse within the finite field as $1/x = \alpha^{q-1-log(x)}$, where $q = p^r$. Using multiplication by the inverse element, we can define division within the finite field. In $GF(2^8)$ we can define $\alpha = 2$, for all of these operations [17].

3.4.3 An IDA Example

All matrix operations must be performed using the arithmetic operations defined in the previous section. As an example, let $p = 2$ and $r = 3$, which defines a finite field of size $2^3 = 8$. Further, assume that the IDA parameters are $k = 3$ and $f = 5$. This means we need a linearly independent transform matrix of size 3×5 ($k \times f$). An algorithm to generate such a matrix is discussed in Chapter 5.1. Notice, however, the process used to generate a encoding matrix is not important, as long as the matrix is linearly independent. The matrix generated by the proposed algorithm is:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 4 & 3 & 6 \\ 1 & 1 & 1 \end{pmatrix}$$

Let's assume the data we want to encode is defined by the matrix:

$$\begin{pmatrix} 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\ 4 & 3 & 6 & 5 & 0 & 7 & 2 & 1 \\ 5 & 6 & 3 & 4 & 1 & 2 & 7 & 0 \end{pmatrix}$$

Encoding the data is done by multiplying the transformation matrix by the data matrix resulting in f encoded pieces of data. The size of each encoded data piece will not exceed the size of the input data piece because all math is done within the finite field.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 4 & 3 & 6 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\ 4 & 3 & 6 & 5 & 0 & 7 & 2 & 1 \\ 5 & 6 & 3 & 4 & 1 & 2 & 7 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\ 4 & 3 & 6 & 5 & 0 & 7 & 2 & 1 \\ 5 & 6 & 3 & 4 & 1 & 2 & 7 & 0 \\ 3 & 1 & 2 & 4 & 7 & 5 & 6 & 0 \\ 2 & 1 & 0 & 7 & 6 & 5 & 4 & 3 \end{pmatrix}$$

If we only retrieve data pieces number 3, 4, and 2, then to recover the original data, we must build another transform matrix, called the decoding matrix, using the originally created transform matrix. As described previously, this transform matrix is made up of the rows of the original transform matrix corresponding to the data pieces gathered (rows 3, 4, 2). The inverse of this constructed matrix is then multiplied by the recovered data to produce the original data.

$$D = \begin{pmatrix} 1 & 1 & 1 \\ 4 & 3 & 6 \\ 0 & 0 & 1 \end{pmatrix}$$

$$D^{-1} = \begin{pmatrix} 7 & 4 & 2 \\ 6 & 4 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 7 & 4 & 2 \\ 6 & 4 & 3 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 1 & 0 & 7 & 6 & 5 & 4 & 3 \\ 3 & 1 & 2 & 4 & 7 & 5 & 6 & 0 \\ 5 & 6 & 3 & 4 & 1 & 2 & 7 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\ 4 & 3 & 6 & 5 & 0 & 7 & 2 & 1 \\ 5 & 6 & 3 & 4 & 1 & 2 & 7 & 0 \end{pmatrix}$$

3.5 PACKING THE STORAGE BINS

Each of the f storable units need to packed into storage bins before storing them in the DHT. This process adds header information (see Figure 3) that will allow us to verify the data during the recovery process. The header contains the hash of the concatenation of the previous storage location, the next storage location, and the user's secret password. This is critical during the recovery process because it allows SCAR to identify hash collisions. By including the password into the seed value of the hash, we ensure that an attacker could not determine the previous or next storage location in the hash chain. This piece of the header is not known until the storage bin is inserted into the DHT. SCAR also adds a checksum of the storable unit data to allow the detection of data errors during recovery before decoding the pieces.

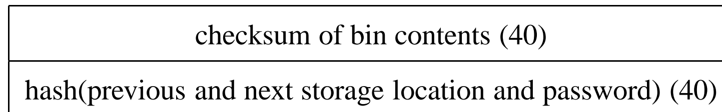

Figure 3: Storage Bin Header, numbers represent size in bytes

3.6 SCATTERING THE STORAGE BINS

We now have a series of storage bins ready for placement into the DHT. SCAR relies upon a hash chain seeded by a password to determine the storage locations within the DHT, refer to Figure 1 (c). The password is known only to those with authorized access to the data being stored. Let the locations produced be called L_0, L_1, L_2, ..., L_n, where L_0 is used as a base for the hash generation. As such, L_0 does not represent an actual storage location. The process to produce the locations is depicted in Figure 4.

L_0 = hash(cat(password, object id, revision number))

L_1 = hash(cat(L_0, password, object id, revision number))

L_2 = hash(cat(L_1, password, object id, revision number))

L_n = hash(cat(L_{n-1}, password, object id, revision number))

Figure 4: Storage Location Generation. Where *cat* is the string concatenation of all arguments.

The chain is seeded with three pieces of information: a name identifying the data object (object id), a revision number, and a secret password. This information is hashed, and the resulting value is then hashed again with the same seeding information (data name, revision, and password), to produce the first storage location (see Figure 4). The result of the previous hash, combined with the original seeding values, is then used to generate the next storage location. This hash chaining is continued until all the units have been assigned a storage location.

A closer look at the above process reveals that the recursive location procedure may result in storage location collisions. To prevent collisions, a request is made to determine if another bin has been previously stored at that location before inserting the storage bin. If so, the algorithm skips that value in the hash chain and attempts to use the next value in the chain. This process is continued until a vacant location in the DHT is found. During the recovery process, the storage bin header is used to determine when such a collision has occurred.

If a collision occurs while trying to store the first unit, SCAR checks to make sure that the hash of the reference name, revision number, and password does not match that of the existing block. If it does, then the recovery process would not be able to detect the error and would recover the wrong data. In this case, we must return an error to the user and ask them to select a different object identifier, revision, or password. This special case is discussed in detail in section 3.8.

As storage locations are determined, *put* requests can be issued to the DHT for each storage bin. This process can be done in parallel, and therefore the time to perform an insert is bounded by the slowest time to insert any one of the storage bins.

3.7 DATA RECOVERY

To recover data in SCAR, the user must provide a object identifier, revision number, and password. Once these pieces of information are determined, SCAR generates the hash chain used to store the original data. SCAR then issues *get* requests against the DHT to start recovering the storage bins. Each bin that is recovered is verified by looking at the hash value of the previous and next storage locations. This ensures that the data being recovered is in the correct sequence. If SCAR detects an error in this process, it assumes a hash collision occurred and skips that storage location.

Similar to the storage process, the data recovery process can occur in parallel. As errors are detected, storage bins recovered for one part of the sequence might be found to be a different part of the sequence, but this can be resolved during the recovery process. Again, the recovery time is bounded by the slowest time to recover one storage bin.

Once all the bins are recovered, they are assembled and decoded using the information dispersal algorithm. Once the data is recovered, the raw data header is used to verify the contents of the entire data object. Lastly, the raw data is decrypted producing the original data.

3.8 HASH COLLISION

There are two types of hash collisions that SCAR does not handle. The first occurs when the information provided by the user (data name, revision, and password), is identical to that of another user. This problem is detected by SCAR when it attempts to store the data, but simply returns a error stating the data already exists. One solution is to add some user identifying information to the meta-data used to generate the storage locations.

The second case occurs if two hash chains using different secret passwords converge to the same series of values. This is very unlikely, but if it did occur, SCAR would again report an error during the insertion phase and request the user to change the data name, revision, or password.

3.9 IMMUTABLE UNITS

Most DHT's implement their storage as immutable objects. To ensure compatibility with these DHT's, SCAR was design to handle immutable objects. To handle this objective SCAR adds a revision number as part of the hash chain. Every time a new revision is stored using SCAR, the revision number is incremented to the next available revision. This process can be hidden from the user by automatically querying the DHT to determine highest available revision number. The process of determining the maximum revision number requires that a single storage bin be retrieved to determine if that revision number already exists. It is worth noting that although designed to handle immutable data stores, SCAR does not require them and could just as easily operate with mutable storage.

4.0 ANALYSIS

This section presents the models developed to describe the availability and security of data stored using SCAR. These models are then used to analyze the inherent tradeoff between availability and security.

4.1 DATA AVAILABILITY MODEL

In SCAR, data availability is dependent upon the availability of nodes that are storing the data pieces. To understand data availability, we first explore the availability of a single node. Having an availability model of a single node, we can then develop a model that describes SCAR's data availability. With this model, it is possible to derive parameters for the information dispersal algorithm.

4.1.1 Node Availability

To model node availability we characterize users based on their behaviors in terms of:

- frequency of which they visit the network, and
- resulting time for each visit in the network.

Based on the above characterization, nodes alternate between two states: Online and Offline. When in the online state, the node is available and therefore is able to provide peer-to-peer services. While offline, however, the node is unavailable and can no longer assume its peer-to-peer responsibilities.

Notice that the reasons which drive a user to alternate between the online and offline states are only relevant to SCAR to the extent that they expose different patterns of behaviors for different classes of users. As such, SCAR is only concerned with the availability, or lack thereof, of the user. Consequently, node availability in the SCAR framework can be modeled as an ON/OFF process, where the average duration in the ON and OFF states are reflective of a particular pattern of behavior. The state diagram depicting

the ON/OFF process is shown in Figure 5. At any given point in time, the node can be either available or unavailable. To model this behavior, we define the following binary variable, which indicates if the node is available at a given time t, as follows:

$$S_i(t) = \begin{cases} 1 & \text{if node i is available at time t} \\ 0 & \text{otherwise} \end{cases}$$

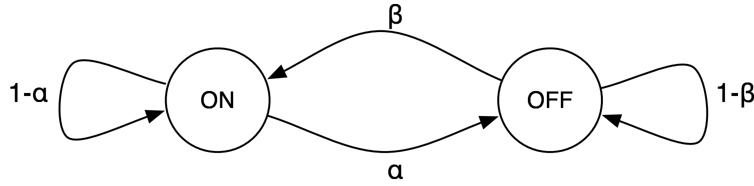

Figure 5: State Diagram depicting ON/OFF model where α and β is the probability of going between ON and OFF states respectively.

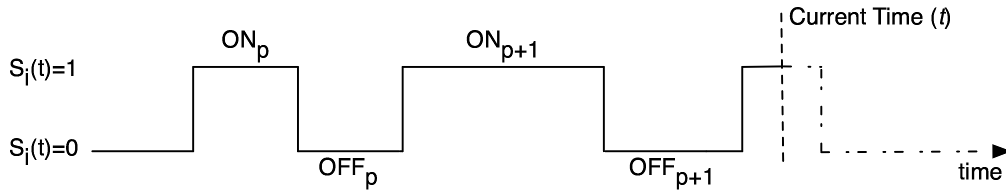

Figure 6: Binary Variable (S_i) over time.

The node's availability is therefore defined as the probability that the value of $S_i(t)$ is 1. The availability of a given node over time is illustrated in Figure 6.

Assuming that the nodes ON and OFF periods have a mean value, we can define the node's asymptotic availability using this binary variable. Because these ON and OFF cycles have a mean, they have an expected value for the amount of time that the node will be ON or OFF. Let $E[ON_i]$ and $E[OFF_i]$ be the expected ON and OFF time for node i respectively. Given this information, we can define A_i as the availability of node i.

$$A_i = \lim_{t \to \infty} Pr[S_i(t) = 1] = \frac{E[ON_i]}{E[ON_i] + E[OFF_i]}$$

The challenge is how to compute the values of $E[ON]$ and $E[OFF]$. This requires the analysis of the corresponding nodes behaviors. Assuming that the typical network users login to the network, perform some work, then leave the network only to return at some later time, the nodes availability can be modeled based on the user's activities and behaviors. Under this assumption, the amount of time a node is ON during a period of time is dependent upon the activity the user is performing. For example, if the user is downloading textual files, they might be present for a duration of time proportional to the file transfer time. If they are streaming a movie, however, they would remain in the network a longer periods of time. The OFF periods are not characterized by the activity of the user but by the user's typical behaviors. For example, a user might login once a day and check for new files whereas another user might login every hour. Therefore, the time a node spends in the ON state is driven by the user's activity while the OFF state is determined by the user's profile. This shows that the distribution model used for the ON and OFF time should be modeled using two separate distribution models.

Based on the above observation, one possible model would be to use an exponential distribution to model the ON state and use a heavy tailed pareto distribution for the OFF state. This is based upon the assumption that the ON state is driven by the user's activities which will vary over time but have no particular pattern. On the other hand, the OFF state is driven by the user's behaviors and the longer a user is away from the system the more likely it is that they will return. This would represent the case where a user logs into a system, leaves, then returns at some point later in time.

While the choice of distributions are important for simulating the actual network behavior our model does not rely on the model itself. Notice that the node's availability, as defined previously, depends only upon the expected ON and OFF time, not the distribution model. This allows us to simplify our analysis by using distributions which have a mean value, allowing us to easily determine the expected ON and OFF times.

4.1.2 Node Types

Based on the node availability model discussed previously, peer nodes are characterized based on the parameters of an ON/OFF process, namely the average duration of the ON and OFF periods. A closer look at a typical users behavior in a peer-to-peer network suggests three classes of users which correspond to three types of nodes: infrastructure, power, and peeper nodes.

Infrastructure nodes represent the core component of a peer-to-peer network. As such, these nodes are expected to remain connected to the network, barring physical failure. Power nodes represent a class of

nodes which are not dedicated resources, but remain connected to the network for extended periods of time. Peepers, on the other hand, are short-lived visitors of the network who are driven by the need to perform a "quick" task. Upon acquiring the needed resource or service, peeper nodes leave the network.

One may argue that a more refined categorization of the nodes may be more reflective of the diverse behaviors of a peer-to-peer network. The three types of nodes defined previously, however, is sufficient to characterize node availability in most frequently encountered scenarios in a typical peer-to-peer network. This node classification is used in the SCAR framework to model node behavior.

Let $E[ON_c]$ and $E[OFF_c]$ be the mean ON and OFF times for each class c. Then, we can define a single node's average availability by summing over all possible classes and their respective availability multiplied by the distribution of that node class.

$$\bar{A} = \sum_{c \in Classes} Pr[available|class = c] * Pr[n_{class} = c] = \sum_{c \in Classes} \frac{E[ON_c]}{E[ON_c] + E[OFF_c]} * Pr[class = c]$$

Conversely, let the node's average unavailability be defined as $\bar{U}$, which is the inverse of $\bar{A}$.

$$\bar{U} = 1 - \bar{A}$$

To determine the probability that a given node is in a particular class, the network is characterized by its nodes class distribution. For example, the network might be composed of 50% infrastructure nodes, 30% power nodes, and the remaining 20% are peepers. This allows us to vary the class distribution of our network and see how this affects the average node availability. This also provides a simple model that can be used to quickly classify nodes and calculate the networks average node availability. The network should monitor its members and provide to its users the current expected node availability so that its users can better utilize the network.

4.1.3 SCAR's Data Availability

Availability of data within the DHT is expressed as a probability that the data stored in the DHT will be available when requested. This probability is dependent upon the probability that a single node is available. This model assumes that the availability of any node is independent of the availability of any other node. Based upon the model of node availability described previously, the average network node un-availability ($\bar{U}$) is used to determine the actual data availability. The following is based upon the work described in [30].

Given the following definitions:

- k - number of fragments required,
- f - total number of fragments,

- $\bar{U}$ - average probability of node being unavailable see 4.1.1, and
- n - total number of nodes.

The probability that the original datum can be recovered is expressed as:

$$\sum_{i=0}^{f-k} \frac{\binom{\bar{U}*n}{i}\binom{(1-\bar{U})*n}{f-i}}{\binom{n}{f}}$$

In the above expression:

- $\binom{\bar{U}*n}{i}$ represents the number of ways i fragments can be stored at the unavailable nodes.
- $\binom{(1-\bar{U})*n}{f-i}$ represents the number of ways $f-i$ fragments can be stored at the available nodes.
- $\binom{n}{f}$ represents the number of ways f fragments can be stored in the n network nodes.

Multiplying the top two terms represents the total number of ways we could arrange the f fragments given that i fragments are unavailable. The last term is a normalization factor. Dividing the equation by this term produces the probability of i fragments being unavailable and the remaining $f-i$ fragments being available. Summing these probabilities over the number of fragments that could be lost, still allowing full data recovery, represents the data availability of SCAR.

The problem with this model is that it depends upon the number of nodes in the network. This is problematic because when we are determining values of f and k, the number of nodes in the network may be unknown. As a result the following model is used:

$$\sum_{i=k}^{f} \binom{f}{i} (1-\bar{U})^{i} (\bar{U})^{f-i}$$

As n increases, the model converges to the previously stated model, as depicted in Figure 7. Once the total number of nodes, n, becomes about 10 times larger than the number of fragments, f, the two models are virtually identical. This model was also used in [25].

An important observation is that using erasure encoding to reliably store data is viable only when the node's availability is sufficiently high. Looking at Figure 8, it is obvious that if the node's availability is less than 80%, then replication exhibits higher data availability than erasure coding based schemes. This is is because erasure coding relies upon a quorum of nodes to be available, and as the nodes in the network become unavailable, achieving the necessary quorum becomes less likely. The advantages of erasure coding, however, is that it uses less storage and increases security; but if the node's availability is low, its effectiveness as a data availability schema is significantly decreased.

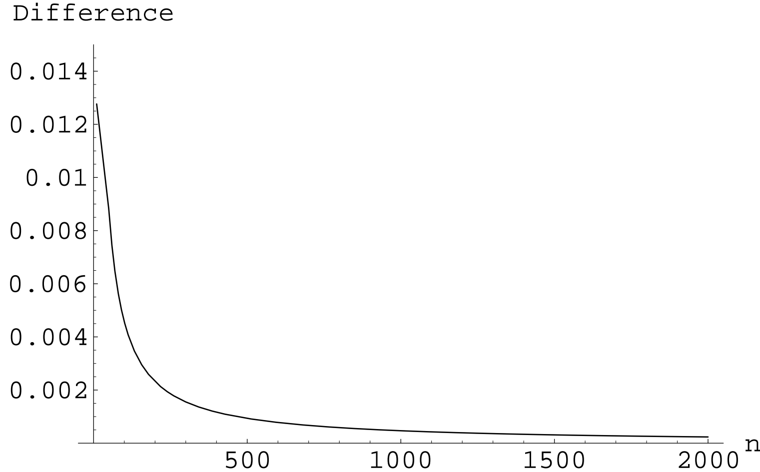

Figure 7: Difference between two availability models, shows that they converge as n increases. $k=7$, $f=10, \bar{U}=0.1$

4.2 SECURITY MODEL

Instead of modeling the security of the entire system, we model only the additional security provided by SCAR. Because symmetrical encryption would be used on the raw data before SCAR is executed, any security provided by SCAR is in addition to this encryption. Based upon this assumption, the model used to evaluate SCAR's security is the probability that an attacker could reassemble the scattered pieces. This probability is based upon the the number of ways an attacker would have to arrange the pieces to be able to obtain the original data. Knowing how many possible ways the attacker has to arrange the pieces gives us an indication of how *hard* it would be for the attacker to get the stored data.

The number of combinations an attacker would have to test is dependent upon the amount of information the attacker knows. An attacker needs three pieces of information to recover the data stored with SCAR. First, the attacker needs to know many how pieces are needed for recovery (this is the k value). Then, the attacker needs to find at least k of these pieces associated with the data in question. After having the k pieces, the attacker must know how to order these pieces. The security provided by SCAR is based upon the fact that the attacker could not easily obtain all three pieces of information.

For practicality reasons, the value of k must be bounded by some maximum value; hence bounding the attackers search space. This maximum is referred to as max_k. SCAR chose 20 to be the value of max_k. This

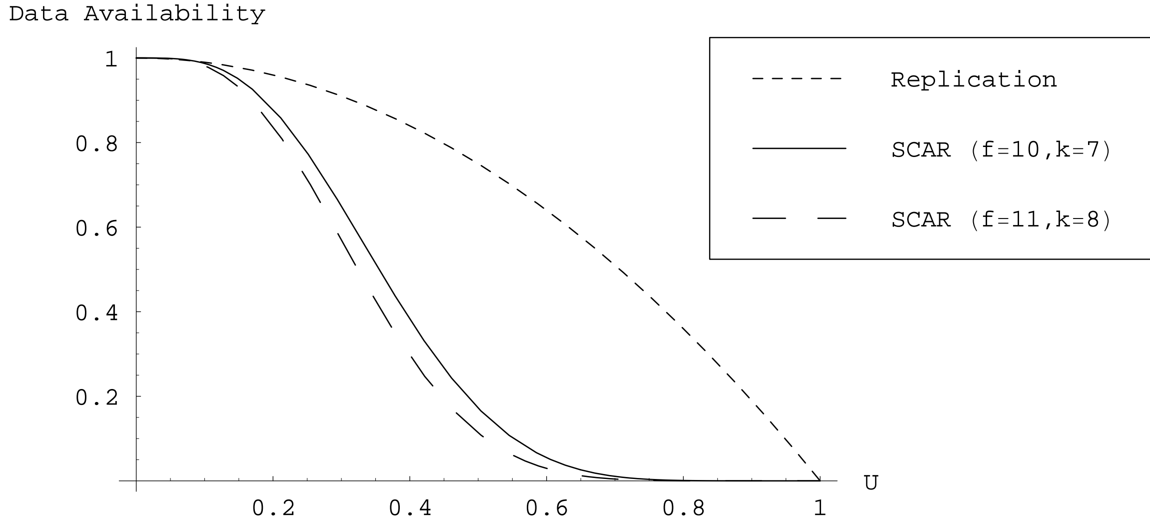

Figure 8: Comparison between replication and SCAR as the network nodes become increasingly unreliable.

security model assumes that the attacker does not know k, and must search for fragment groupings of size 1 to max_k.

Knowing where to find the k pieces is dependent upon the attackers *perceived capacity* of the network. The perceived capacity represents how large the storage capacity of the network appears to the attacker. For example, if the attacker has no information about where the data pieces are scattered, the perceived capacity of the network is the size of the entire network. Clearly searching the entire network for a small subset of pieces, requires that the attacker explores a large number of combinations. On the other hand, if the attacker can monitor the network traffic, he has a smaller perceived capacity. This is because he can refine his search space to a smaller subset of pieces accessed by the SCAR user. Also note that the value of k is bounded by the perceived capacity.

Once k pieces are selected, the attacker must find the correct ordering of these pieces. The order is important because IDA requires that the decoding matrix is built using the same properly ordered rows of the encoding matrix. Recall that we use the same encoding matrix for each value of k and f. As a result, this model assumes this encoding matrix is known to the attacker, but knowing the encoding matrix does not allow the attacker to know the order of the data pieces.

These observations allow us to develop a model that describes the number of combinations the attacker must check. The probability, P_c, that a given combination is the correct combination is defined by the

following expression.

$$P_c = \frac{1}{\sum_{i=1}^{min(c,max_k)} \binom{c}{i} i!}$$

- c - the attackers perceived capacity of the network
- max_k - the maximum value of k (number of a pieces)

Consider the case where the attacker knows the total number of data pieces, f. By knowing f, the attacker is effectively reducing the perceived capacity, c, of the network. This would be possible if the attacker monitored the network traffic between the SCAR user and the DHT.

From this analysis, we can conclude that to ensure SCAR's security, the attackers *perceived capacity* of the network must be increased. One possible way to achieve this is to introduce fake data, thereby increasing the data the attacker would have to check. This method would introduce data into the network that has no value, and would be a very high price to pay for security. A more efficient method would be to batch data accesses over time. This process would cause the data access of a data object A to overlap with the data access of object B, again increasing the attackers perceived capacity. By doing this, we introduce storage latency at the cost of increased security.

In order to determine the relationship between the perceived capacity and the value of k, we further assume that the attacker knows k. In addition to simplifying our model, we believe that in practice the value of k will be fixed or derived from a small range of values. Regardless of the size of the data being stored, the number of pieces being generated (k) will be the same. This allows us to define the probability of an attacker recovering the data from any given combination of pieces, as follows:

$$\frac{1}{\binom{c}{k} k!}$$

Considering Figure 9, observe that when $k = 8$, an increase in the perceived capacity causes the probability of a successful attack to decrease rapidly. By fixing $k = 8$, further consider the effect of perceived capacity on security. Figure 10 shows the effect of varying the perceived capacity; even a small increase in the capacity can cause a large increase in security.

Since increasing the perceived capacity is accomplished at a cost of either storage efficiency or latency cost, there is a desire to bound this value. To analyze this, consider the amount of time it might take an attacker to try each of the possibilities. We assume that each possibility would take 1 second to assemble and test a set of fragments. Table 1 illustrates the effect that increasing the capacity has on the time it would take to reassemble the pieces. The observation is with a value of $k = 8$ and a perceived capacity of 20, the time to try all possible combinations makes it practically infeasible for an attacker to reassemble the data.

It is therefore reasonable to select $k = 8$ and $c = 20$ in the analysis. This means that when we access data, SCAR should interleave that data access within another data access, assuming $n >= 10$. This effectively doubles the perceived capacity of the network, thus making $c >= 20$.

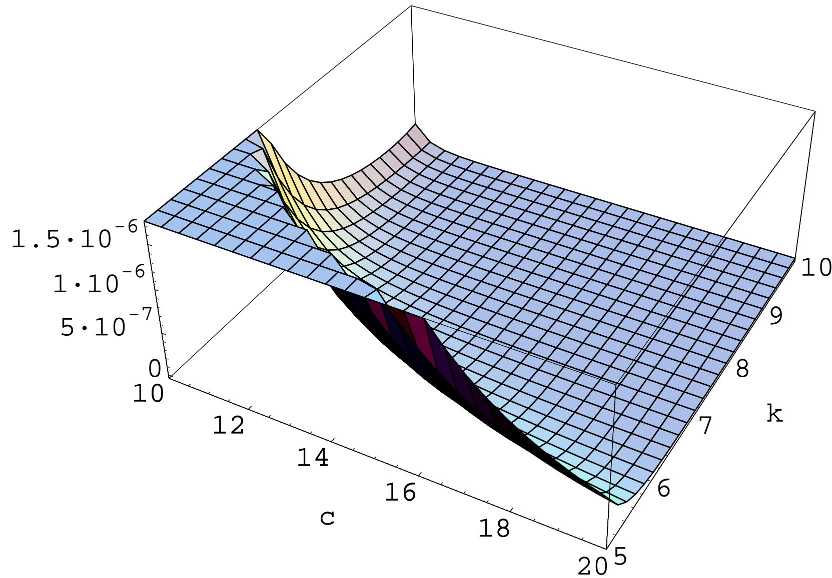

Figure 9: Security model varying the networks perceived capacity,c, and k; $f = 10$, $max_k = 20$

4.3 TRADEOFF

Using availability and security models, the goal is to find the values of k and f which "optimize" both availability and security. To accomplish this, both models must be integrated into a model that captures the tradeoff between availability and security. The availability metric is used as it was presented previously. The security metric, however, only reflects the probability of finding one given fragment combination. Because of the scale difference in the availability and security probabilities we look at the probability of a security failure within a given time frame, using the same computing power assumptions stated in the previous section. For this analysis we use a 5 year time frame. This then allows the combination of the two metrics into a single metric, called the tradeoff metric. The intuition is that users want to maximize both availability and security. Using the tradeoff metric one can see directly the balance between availability and security, as

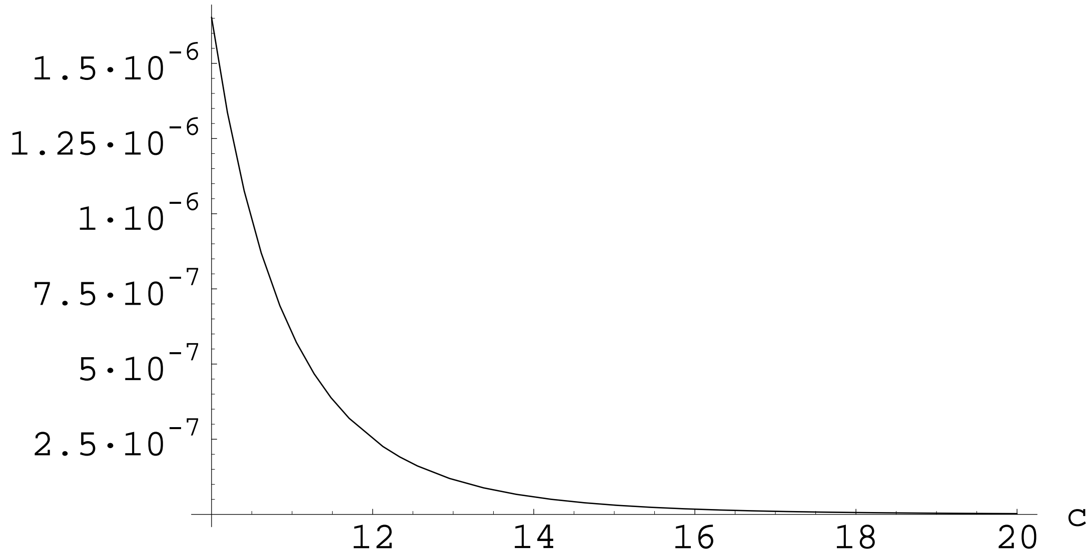

Figure 10: Security model varying the networks perceived capacity (c); $k = 8$, $f = 10$, $max_k = 20$

it relates to the number of pieces generated and required (f and k). We combine these two probabilities into a weighted summation, thereby enabling the user to set the importance of security relative to availability.

$$Tradeoff\,Metric = w_1 * Availability\,Model + w_2 * Probability\,Of\,Security\,Failure$$

In our analysis we assume that availability and security are equally important, and set the weights to 0.5. In Figure 11, we observe that the optimal value for k would be 8 when the value of f is set to 10 and the node's unavailability is 10%. Intuitively this makes sense: for maximum availability, a small number of pieces are required, but for maximum security a large number of pieces should be required. The tradeoff model shows that the balance point is not in the middle but shifted closer towards the total number of pieces generated (f). In our case we have $f = 10$ and $k = 8$.

Using this model enabled us to determine a value for k for a fixed value of $f = 10$ and $p = 0.10$. This model does not limit the value f, and we can observe in Figure 12 that by increasing f we can achieve better performance. Because we are increasing the perceived capacity of the network by batching multiple data accesses, increasing f also increases the perceived capacity. Using this model, we can see in Figure 13 that by fixing $k = 8$, we can increase the metric significantly by adding one extra data fragment.

capacity	combinations	calculation time
10	$1.8144 * 10^6$	21 days
12	$1.99584 * 10^7$	231 days
13	$5.18918 * 10^7$	1.64548 years
15	$2.59459 * 10^8$	8.2274 years
20	$5.07911 * 10^9$	161.058 years
25	$4.36091 * 10^{10}$	1382.84 years
30	$2.3599 * 10^{11}$	7483.19 years
40	$3.1008 * 10^{12}$	98325.6 years
50	$2.16469 * 10^{11}$	686420.2 years

Table 1: Effect of increasing the perceived capacity of the network. $k = 8$, $f = 10$, $max_k = 20$

It is important to note that the values, $k = 8$ and $f = 11$, were determined by setting the average node availability to $\bar{A} = 0.90$, or $\bar{U} = 0.1$. As the average node availability decreases, neither data security nor availability can be achieved using SCAR. As the network becomes unstable, any availability scheme must use multiple replicas of the data, causing data security to decrease. The tradeoff between availability and security can only be balanced when the availability of the underlying network nodes are between 80% and 100%. Figure 14 depicts this behavior using our tradeoff metric. When the nodes are available 90% of the time, there is a clear peak at $k = 8$, but once the node availability falls below 80% this peak becomes significantly less. This further highlights the interplay between the contending goals, security through randomness and availability through replication.

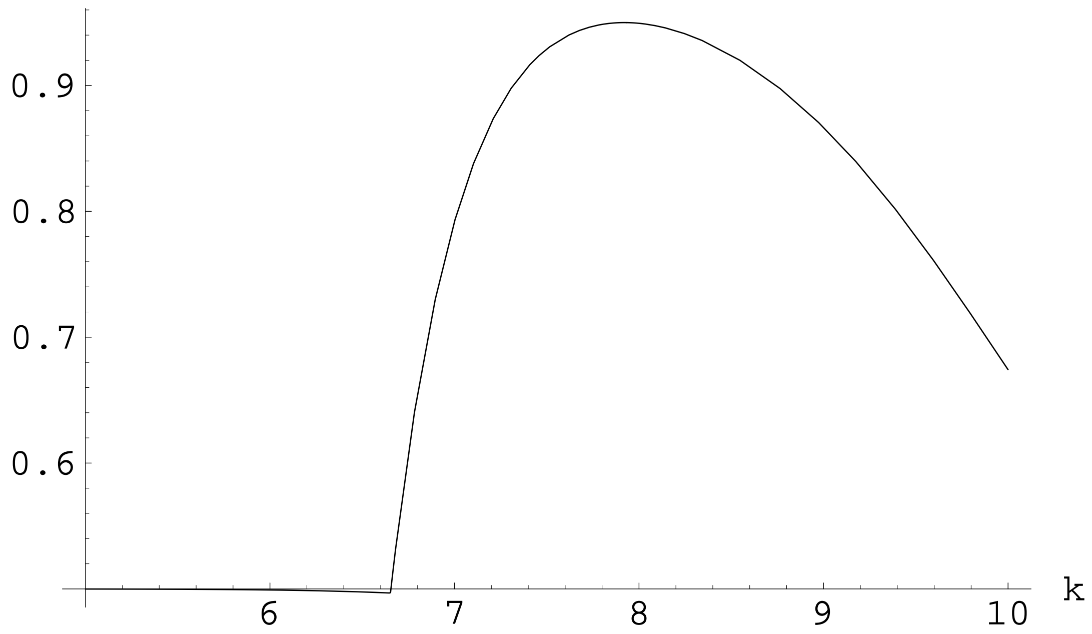

Figure 11: Tradeoff model varying k; $f = 10$, $max_k = 20$, $\bar{U} = 0.1$

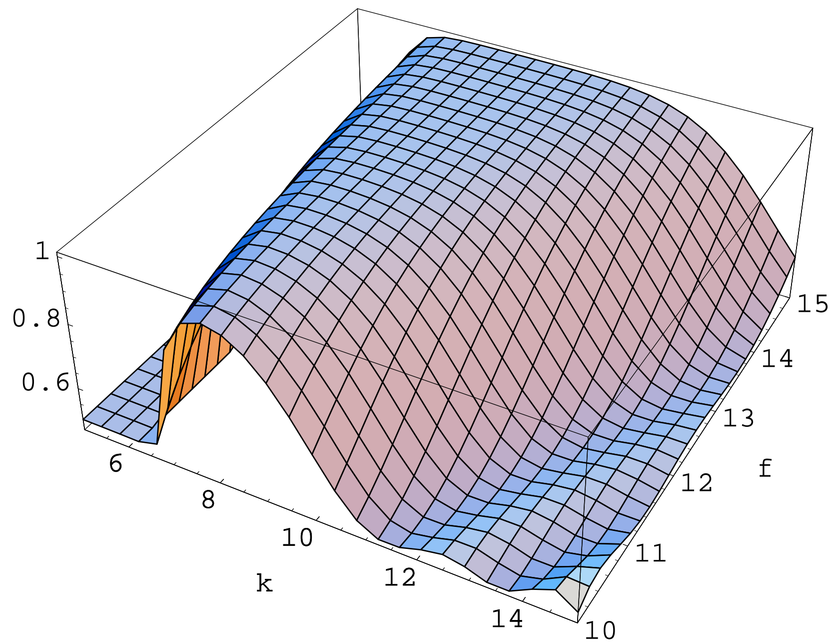

Figure 12: Tradeoff model varying k and f; $max_k = 20$, $\bar{U} = 0.1$

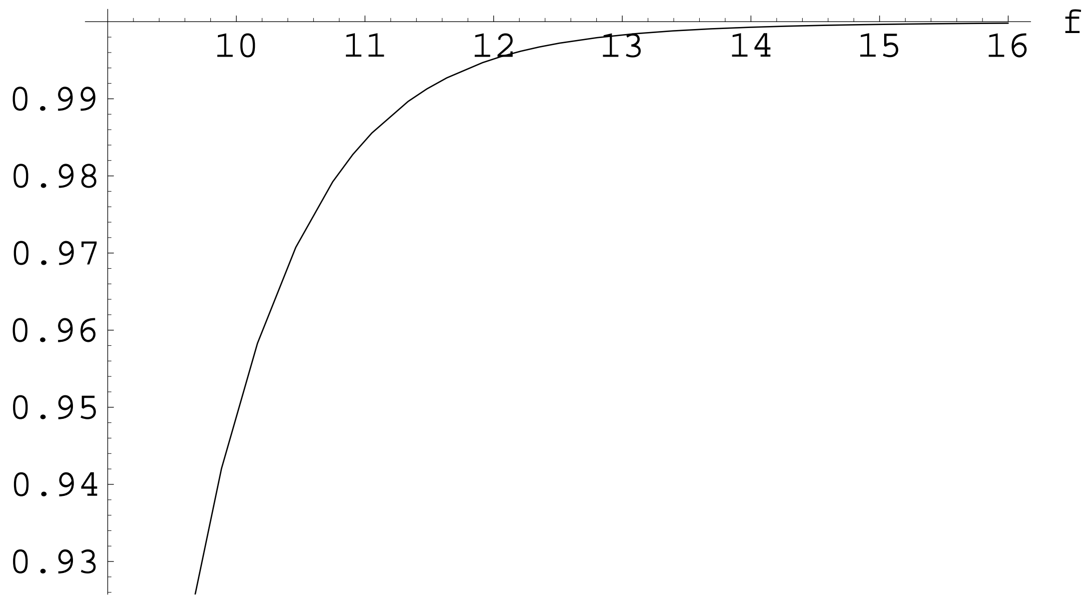

Figure 13: Tradeoff model varying f; $k = 8$, $max_k = 20$, $\bar{U} = 0.1$

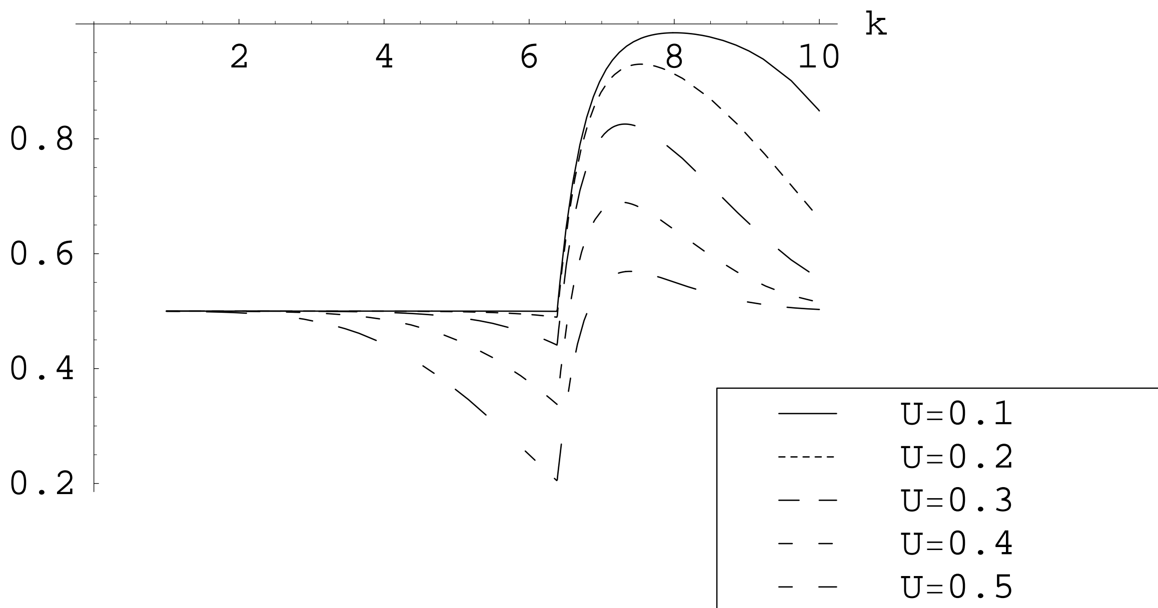

Figure 14: Tradeoff model varying $\bar{U}$; $f = 11$

5.0 IMPLEMENTATION AND EXPERIMENTS

5.1 IMPLEMENTATION

SCAR was implemented as an application outside the DHT, and therefore is not bound to any specific DHT implementation. To accomplish this, there is an interface layer between the generation of the storage bins and the actual data accesses. Currently there is only two storage interfaces defined. One makes use of OpenDHT's web services API [23] and the other is a file system interface used for development. SCAR's architecture consists of 7 different components:

- Data Preprocessor - Encrypts raw data, generate data padding if necessary, prepends checksum and other header information to the raw data.
- File Splitter - Splits a given file into k pieces.
- Encode - This transforms k into f encoded pieces, such that any k of those encoded pieces can be used to recover the entire data object.
- Storage Bin Generation - This processes the f encoded pieces into storage bins which contain additional header information.
- Storage Location Generator - Given identifying information generates a list of storage locations.
- Storage Manager - This component handles the request to store data. It accepts a list of storage bins and determines how they will be stored (ie. batching transaction). This component is also capable of retrieving data given a list of storage locations.
- Storage Interface - Interacts directly with the storage medium (ie. DHT, filesystem, etc).
- Decoder - Given k pieces of data, this process decodes the encoded data.

5.1.1 Details

SCAR was implemented using the Python programming language, and contains a command line interface. This command line utility allows the user to specify a file to store in the DHT. Here, the user is prompted for

a password, which is used to store the data in the DHT. The same command line can also be used to retrieve data from the DHT by specifying a filename and revision, again prompting for a password. In addition, values for f and k can be specified on the command line. Based upon the research presented in this thesis, our default values are $f = 11$ and $k = 8$.

5.1.2 Encoding and Decoding

The implementation of an encoding and decoding process uses two basic matrix operations under a finite prime field: multiplication and inversion. Once these two matrix operations are available, the next challenge is to generate a linearly independent transformation matrix. To do this, SCAR makes use of the algorithm listed in [17], and described in Algorithm 1.

The python language does not have native libraries for handling either matrix operations or finite fields. In addition, the matrix libraries available made various assumptions about the mathematical properties of the matrix contents making them difficult to use. Because of this, a custom matrix class was written which enabled finite field arithmetic, called *FiniteFieldMatrix*. To perform matrix inversion, this class uses Gaussian reduction. One method of improving performance is to use a more efficient matrix inversion algorithm.

5.2 EXPERIMENTS

In addition to the implementation described in section 5.1, experiments were run using a DHT simulator that replicated the properties of SCAR. In addition to emulating SCAR, the simulator also allows experimentation with the properties of the network including the node's availability, distribution of types of nodes, and probability of data access. In the following section, we will describe the simulator used, compare the theoretical models to simulations, analyze SCAR's sensitivity to network changes, and compare SCAR's availability to that of replication.

5.2.1 Simulator

The simulator was built to allow quick spawning of multiple nodes. Each node is assigned a unique key which was a MD5 hash value. This emulates the way nodes are created in a DHT. Next, f pieces of data were stored at locations within the simulated DHT by using hash chains to determine the storage locations of the data. The simulation of the hash chain process is the same used in the implementation. For each

Algorithm 1 Create Encoding Matrix

1: f=total number of fragments
2: k=number of required pieces
3: Construct a $(n-f)xk$ matrix A
4: **for** $i = 0$ to $(f-k)$ **do**
5:     **for** $j = 0$ to k **do**
6:         A[i][j]=$(2^{f-k+i})^j$
7:     **end for**
8: **end for**
9: Construct a kxk matrix B
10: B[0][0]=1
11: **for** $j = 0$ to $k-1$ **do**
12:     B[0][0]=0
13: **end for**
14: set B^{-1} to the inverse of B in GF
15: set $C = A * B^{-1}$, this is a $(n-k)xk$ matrix
16: construct a fxk matrix
17: construct E with the first k rows being the identity matrix and the remaining $(f-k)$ rows being C
18: return E

storage location, the node with the closest numerically matching id was selected to store the data. At any point, the simulated network can be queried for a stored data object given the current network state. This simulates the same erasure coding used in SCAR, and further allows the simulation of replication by setting $f = 2$ and $k = 1$.

To simulate nodes ON and OFF behavior, nodes alternate between exiting and entering the network based on their node type. This ON or OFF probability is based upon a distribution model specified, or can be set to fixed probability. At each simulation step, all nodes are given the opportunity to change their current state. Each node in the simulator is created as an independent object thus allowing the specification of any node properties independently of the other nodes. This in turn allows us to simulate a network composed of different types of nodes. During the simulations several standard network configurations were used, these are listed in Table 2.

Infrastructure	Power Users	Peepers	Average Node Availability
100%	0%	0%	98.39%
80%	10%	10%	86.78%
70%	20%	10%	83.44%
50%	30%	20%	71.66%
40%	30%	30%	63.36%
20%	30%	50%	46.70%
10%	20%	70%	33.41%

Table 2: Simulated average node availability for network configurations. Standard Deviation was less than 0.01%.

5.2.2 Data Availability

The goal of this experiment was to validate the theoretical models discussed in section 4. Using the simulator, we created a network with homogeneous nodes, all with the same fixed node availability. Data was inserted into the network using various values of f, k, n, and node availability. We compared the simulation results with the results obtained using the analytical models. As can be seen in Figures 15 and 16, the simulation results show that the analytical models offer accurate estimates of SCAR's behavior.

In a few cases the experimentation resulted in two pieces of data being stored on one node, causing the

data availability to be less than that predicted by the model. In the next experiment, we look at the likelihood that a single node might be assigned two pieces of data from the original data object.

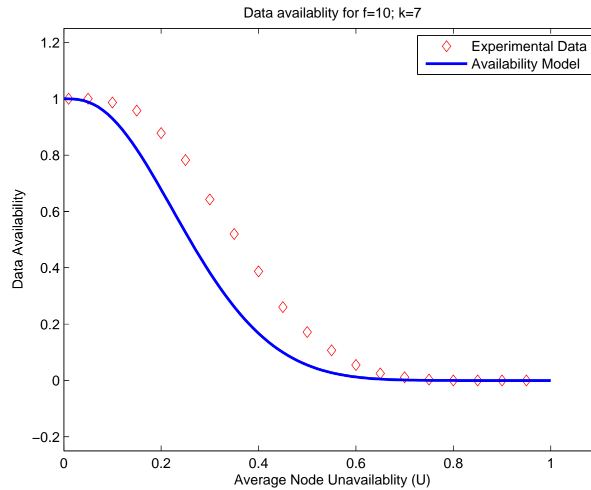

Figure 15: Comparing Model and Experimental Results varying $\bar{U}$ using $f = 10$, $k = 7$, $n = 1000$. Each experimental point represents mean recovery rate of 1000 samples.

5.2.3 Node Collision

SCAR's availability metric assumes that each piece of data is distributed to a unique node location. This property is required because the metric assumes that data piece failures are independent from one another; however, if two or more pieces of data were stored on the same node, those failures would no longer be independent. To confirm our assumption, we simulated the storage of a 1000 data objects using SCAR and monitored the network for node collisions within the same data object. The obvious observation is that as the total number of nodes in the network increases, the probability of node collision decreases. Inversely, as f increases the probability of a node collision increases. This can be seen in Figure 17.

As can be observed in Figure 18, the probability of node collisions are very low when using a reasonably large network. In Figure 18 the value of f was fixed to 11 and the total number of nodes was varied to determine how many collisions resulted. The simulation was run multiple times for all 1000 data objects using randomly generated network nodes of the specified size. During each experiment the likelihood of a node collision was calculated by looking at the number of actual collisions divided by the number of pieces that didn't result in a collision. To characterize the worst case behavior, Figure 18 shows the maximum ratio of collisions to non-collisions observed for a network of the specified size.

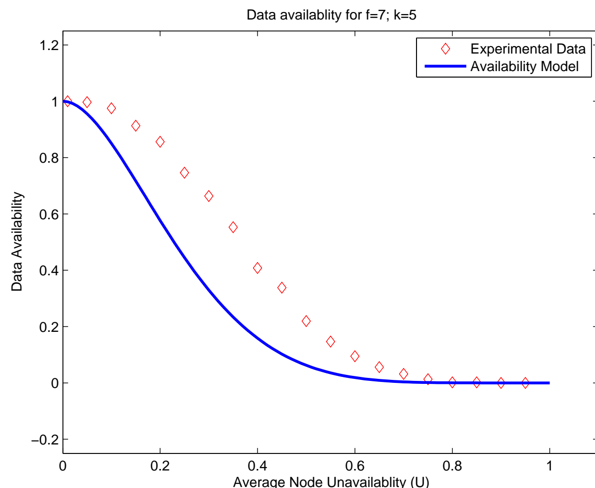

Figure 16: Comparing Model and Experimental Results varying $\bar{U}$ using $f = 7$, $k = 10$, $n = 1000$. Each experimental point represents mean recovery rate of 1000 samples.

5.2.4 Network Sensitivity Analysis

In this section, we focus on SCAR's sensitivity to network changes. In particular, we focus on the data availability metric as the nodes in the network become less available. As mentioned in chapter 4, erasure coding is only effective at increasing data availability when the nodes themselves are fairly reliable ($\sim 80\%$). To validate this assumption, we first define probabilities of nodes leaving and rejoining the network for the three defined node types, see section 4.1.2. The simulation was executed using an exponential distribution with mean probabilities set to the values listed in Table 3. The column labeled average node availability is the measured average amount of time the nodes of that type were available across the lifetime of the network. The variance in these numbers was less than 0.5 a percent; therefore, the variance is not listed.

The purpose of this experiment is to determine how SCAR performs as the network becomes unstable. To simulate this behavior, we start with a network of only *Infrastructure* nodes and begin adding *Peepers* to the network, thus increasing the instability of the network. The results are presented in Figure 19. This indicates that when the average node availability is above 80%, the data availability is comparable with replication; however when availability is below 80%, SCAR's data availability becomes un-acceptable. This is identical to the prediction made by our theoretical models, and confirms that SCAR is only effective when the network is made of mostly available nodes.

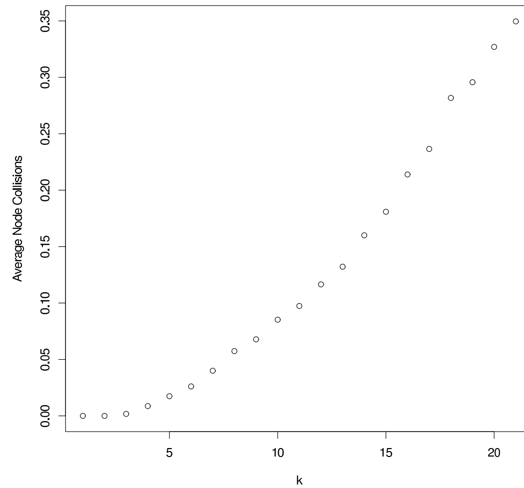

Figure 17: Average number of collisions varying number of fragments; $n = 1000$.

5.2.5 Likelihood of Recovery and Data Availability

During the analysis and experiments, we have seen that SCAR does not perform well when the nodes in the network are unreliable. Observe that availability is dependent upon the user. When the user requests the data it must be available, but that doesn't mean the data has to be available at all time points. This reasoning led us to investigate the way SCAR performs if data only needs to be available a fraction of the time. Instead of verifying data availability at each time-point, we modified our simulation to check only a percentage of the time. In Figure 20, it can be seen that varying the probability of data checking has no effect upon the data availability. This is explainable because the overall availability would not be effected by the probability of accessing data. Therefore, to solve the data availability problem, one must make the data available at all times in order to increase the data availability for particular user.

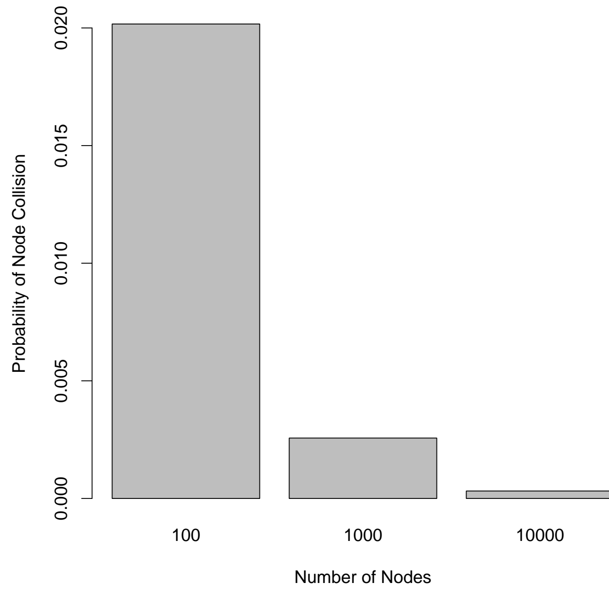

Figure 18: Maximum number of collisions to non-collisions varying number of nodes; $f = 11$.

type	probability of leaving ($\bar{\alpha}$)	probability of re-joining ($\bar{\beta}$)	$\bar{A}$
Infrastructure	1.0%	95.0%	98.0%
Power Users	20.0%	40.0%	65.0%
Peepers	80.0%	10.0%	15.0%

Table 3: Network Sensitivity Analysis Settings

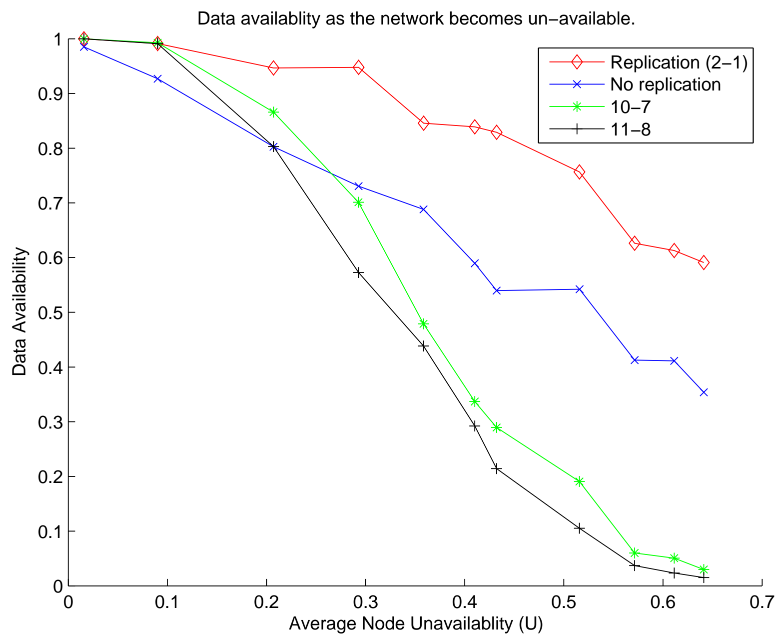

Figure 19: SCAR Simulation as network becomes unstable. Simulated by starting with a network of 1000 infrastructure nodes then adding peepers to the network, thereby making the network increasingly un-available. Each data point represents the mean data availability of 10 objects over 200 simulation runs.

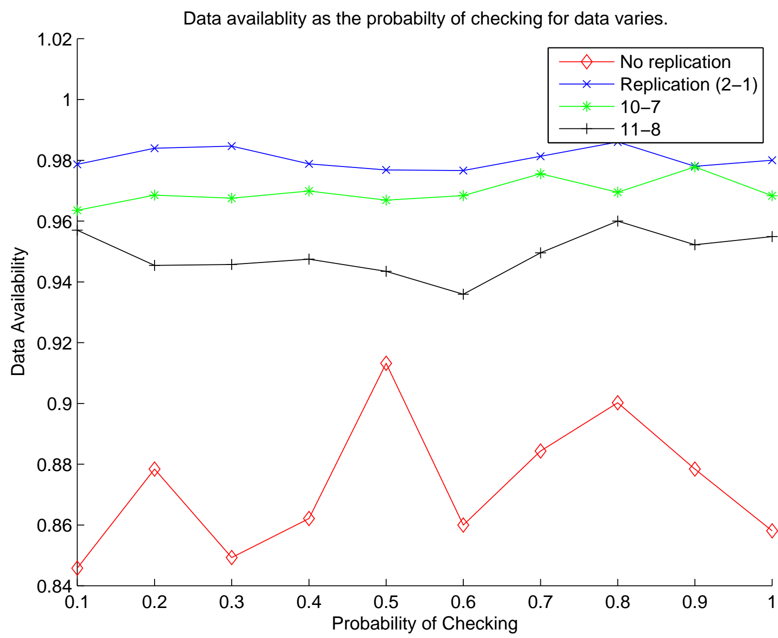

Figure 20: Data Availability as probability of checking for data varies. Network configuration (80% Infrastructure; 10% Power Users, 10% Peepers). Each data point represents the mean data availability for 10 data objects over 200 simulation runs.

6.0 CONCLUSION

Peer-to-peer technologies promise to be a solution to the distributed storage problem, but current research has failed to fully address the problem of distributed data security. This thesis presents a novel and elegant solution that provides both data security and availability using peer-to-peer networks. The framework presented combines hash chaining, erasure coding, and distributed hash tables to create a complete solution to a complex problem. A fully working implementation was developed thereby validating the protocols and design discussed. In addition, this thesis provides an analysis of the proposed framework.

Analytical models were developed to model SCAR's security and availability characteristics. The analysis of these models showed us that the erasure coding techniques used by SCAR were only effective when the peer-to-peer node's availability is sufficiently high, $> 80\%$. This was a disappointment, however, this thesis has provided the framework for evaluating new availability schemes. This work further emphasized the tradeoff between availability and security.

6.1 FUTURE WORK

This thesis provides a complete prototype and provides a platform for exploring other issues in using DHT's to provide distributed storage. One of the metrics needing further study is how SCAR's design effects system latency. The hypothesis is that SCAR will not hurt, and might even decrease system latency. This is because reading and writing data pieces using SCAR can be done in parallel.

Increasing an attackers perceived capacity, as described in section 4.2, needs more investigation. The goal here is to find other methods besides batching that will improve security, while not introducing access latency.

Using the SCAR framework to store user profile information promises to be a solution to a privacy concerns of millions of web users. One potential solution is to develop a web browser extension that allow

users to securely store their bookmarks, preferences, and saved passwords. The motivation is to enable secure data storage that is not controlled by a single organization and is only accessible to the information owner.

Another potential application is to store digital photos, which led to the idea of allowing multiple entry points into the data. The idea would modify the hash chain to allowing multiple data entry points. For instance, one staring hash value could be used to access a lower resolution image while another value is used for accessing high resolution images. If this could be accomplished using the same hash sequence, the data overlap could reduce the amount of redundant data being stored.

In both the analysis and simulations, we observed that as the average node availability goes down, the effectiveness of SCAR also decreases. There are two options for using SCAR: build a network with high node availability or select storage nodes based upon their availability. The selection process would require knowledge of the node's availability, and therefore require changes to the underlying DHT. Future work could look at what changes need to be made within the underlying DHT to improve SCAR's availability and security.

6.2 SUMMARY

This thesis presents a method for using peer-to-peer networks to securely and reliably store data using a novel combination of erasure coding and hash chaining. This thesis also provides implementation details based upon the prototype that was developed showing the feasibility of SCAR. Beyond developing SCAR, this thesis provides an analysis of the inherent tradeoff between security and availability. The exploration of this tradeoff lead to the conclusion that SCAR can effectively balance this tradeoff when the nodes of the network are sufficiently available.

BIBLIOGRAPHY

[1] *Gnutella*, http://www.gnutella.com/.

[2] *Kazaa*, http://www.kazaa.com.

[3] *The Peer-to-Peer Trusted Library*, http://sourceforge.net/projects/ptptl/.

[4] *Project JXTA*, http://www.jxta.org/.

[5] B. Awerbuch and C. Scheideler, "Towards a scalable and robust dht," in *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, (New York, NY, USA), pp. 318–327, ACM Press, 2006.

[6] J.-M. Busca, F. Picconi, and P. Sens., "Pastis: A highly-scalable multi-user peer-to-peer file system.," in *Euro-Par 2005*, (Lisbon, Portugal), 2005.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Lecture Notes in Computer Science*, vol. 2009, 2001.

[8] L. Cox and B. Noble, "Pastiche: Making backup cheap and easy," in *Fifth USENIX Symposium on Operating Systems Design and Implementation*, (Boston, MA), December 2002.

[9] F. Dabek, B. Zhao, P. Druschel, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," in *IPTPS '03*, February 2003.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, (Chateau Lake Louise, Banff, Canada), Oct. 2001.

[11] A. Detsch, L. P. Gaspary, M. P. Barcellos, and G. G. H. Cavalheiro, "Towards a flexible security framework for peer-to-peer based grid computing," in *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, (New York, NY, USA), pp. 52–56, ACM Press, 2004.

[12] J. R. Doueeur, "The sybil attack," 2002.

[13] N. M. Haller., "The s/key one-time password system.," in *In Proceedings of the Internet Society Symposium on Network and Distributed Systems*, 1994.

[14] S. Hand and T. Roscoe, "Mnemosyne: Peer-to-Peer Steganographic Storage," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Boston, MA, USA*, March 2002.

[15] D. Karger, A. Sherman, A. Berkhemier, B. Bogstad, R. Dhanidina, B. K. K. Iwamoto, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," in *Eighth International World Wide Web Conference*, May 1999.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ACM ASPLOS*, ACM, November 2000.

[17] G. H. Lynn, *ROMR: Robust Multicast Routing in Mobile Ad-hoc Networks*. PhD thesis, Univeristy of Pittsburgh, 2003.

[18] E. L. Miller and R. H. Katz, "RAMA: An easy-to-use, high-performance parallel file system," *Parallel Computing*, vol. 23, no. 4–5, pp. 419–446, 1997.

[19] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: a read/write peer-to-peer file system," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 31–44, 2002.

[20] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, vol. 36, no. 2, pp. 335–348, 1989.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 161–172, ACM Press, 2001.

[22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht," in *In Proc. of USENIX Technical Conference*, June 2004.

[23] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public dht service and its uses," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 73–84, ACM Press, 2005.

[24] R. Rivest and A. Shamir, "Payword and micromint: Two simple micropayment schemes.." manuscript, 1995.

[25] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *IPTPS '05: 4th International Workshop on Peer-To-Peer Systems.*, February 2005.

[26] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, 2001.

[27] A. Shamir, "How to share a secret," in *Communication of the ACM*, ACM Press, 1979.

[28] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash table," in *IPTPS '02*, 2002.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 149–160, ACM Press, 2001.

[30] H. Weatherspoon and J. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in *IPTPS ’02*, 2002.

[31] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *IEEE Journal on Selected Areas in Communications*, pp. 41–53, 2004.