

DESIGN AND EVALUATION OF
A FAULT TOLERANCE PROTOCOL IN BISTRO

by

Leslie Chi-Keung Cheung

A Thesis Presented to the
FACULTY OF THE SCHOOL OF ENGINEERING
UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the
Requirements for the Degree
MASTER OF SCIENCE
(COMPUTER SCIENCE)

May 2004

Copyright 2004

Leslie Chi-Keung Cheung

Acknowledgements

First of all, I would like to express my thanks to my advisor Leana Golubchik. Without her guidance I would not have completed this thesis. On technical matters, she gave me a lot of useful comments to improve this thesis. Personally, she gave me tremendous amount of support and encouragement.

I am grateful to Banu Ozden and Cyrus Shahabi for serving on my thesis committee, and for giving me comments to improve this thesis.

I also want to thank my co-worker, Yan Yang. He designed the protocol with me, and gave me a lot of useful feedback on improving this thesis.

Thanks also go to Cheng-Fu Chou for providing me with his previous work on this topic. His work provided me an excellent outline for this thesis.

I thank my colleagues at the Internet Multimedia Lab for participating in discussions on this work, and for providing me feedback on earlier versions of this thesis. Thank you for providing me a nice place to work, and for letting me occupy the conference table all the time when I was in the lab.

Finally, I wish to express my love to my parents. They provided a lot of emotional support for me, especially during my hard times.

Contents

Acknowledgements	ii
List of Tables	v
List of Figures	vi
Abstract	vii
1 Introduction	1
2 Related Work	4
2.1 Fault Tolerance in Distributed Systems	4
2.2 Erasure Codes in Computer Networking	6
3 Overview of Erasure Codes	8
4 Fault Tolerance Protocol	11
4.1 Timestamp Step	11
4.2 Data Transfer Step	15
4.3 Data Collection Step	16
5 Analytical Models	19
5.1 Modeling Reliability of Checksum Groups	19
5.1.1 Independent Packet Losses	20
5.1.2 Independent Bistro Failures	20
5.1.3 Gilbert Model	21
5.2 Overall Reliability Model	23
5.3 Performance Model	26
5.3.1 Server Performance in the Timestamp Step	26
5.3.2 Size of timestamp messages	28
5.4 Cost function	28

6	Results	30
6.1	Setting Weights	31
6.2	Varying the Number of Checksum Groups in a FEC Group	32
6.3	Varying the Number of Parity Packets in a FEC Group	34
6.4	Varying the Number of FEC Groups in a File	35
6.5	Varying the Probability of Losing a Packet	37
7	Future Work	38
8	Conclusions	40
	Bibliography	42

List of Tables

4.1	Meaning of Variables	13
5.1	Summary of Content of Checksum Groups	24

List of Figures

1.1	Depiction of upload processes with and without Bistro	2
3.1	Illustration of an Erasure Code	8
4.1	Timestamp Step	12
4.2	FEC Groups and Checksum Groups	12
4.3	Data Transfer Step	15
4.4	Data Collection Step	17
5.1	Gilbert Model	21
5.2	Average Time to Compute Hash of Different Number of Checksums	27
6.1	Cost Function - varying weights	31
6.2	Cost Function - varying Z	33
6.3	Cost Function - varying $n - k$	34
6.4	Cost Function - varying k	36
6.5	Cost Function - varying p	37

Abstract

This thesis investigates fault tolerance issues in Bistro, a wide area upload architecture. In Bistro, to achieve scalability and to avoid hot spots when deadlines approach, clients first upload their data to intermediaries, known as bistros. A destination server then pulls data from bistros as needed. However, during the server pull process, bistros can be unavailable due to failures, or they can be malicious, i.e., they might intentionally corrupt data. As a result, we need to provide a fault tolerance protocol within the Bistro architecture. Thus, in this thesis, we develop such a protocol which employs erasure codes in order to make the data uploading process more reliable. We develop analytical models to study reliability and performance characteristics of this protocol, and we derive a cost function to study the tradeoff between reliability and performance in this context.

Chapter 1

Introduction

High demand for some services or data creates hot spots, which is a major hurdle to achieving scalability in Internet-based applications. In many cases, hot spots are associated with real life events. For instances, releasing a new version of certain software may create a huge demand for it soon after the release, which may overload servers which distribute that software.

There has been a lot of research aiming at relieving hot spots in the Internet due to download applications, such that Internet applications can operate efficiently even under heavy loads. Examples include replication of services (e.g., replication of DNS servers), data replication (e.g., web caching and Akamai [1]), and data replacement (e.g., different streaming rates for audio and video streaming). Types of communicate mode which these research addresses are mostly: one-to-one such as email and instant messaging, one-to-many such as web downloads, and many-to-many such as chatrooms and video conferencing.

To the best of our knowledge, however, there are no research attempts to relieve hot spots in many-to-one applications, or upload applications, except Bistro [4]. Examples of upload applications are: interactive TV polling, online tax form submissions, homework submissions in distance education, and conference paper submissions. Current

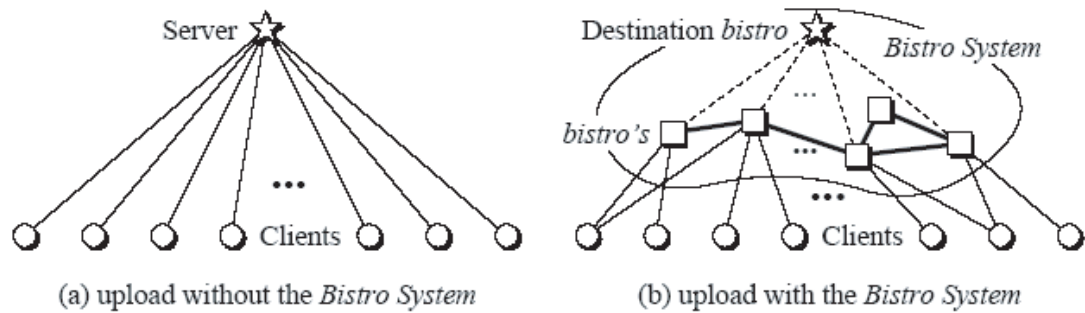


Figure 1.1: Depiction of upload processes with and without Bistro

upload applications usually make use of many individual one-to-one transfers. This is not a scalable solution because the server or the network around the server can be saturated. This problem is exacerbated when real life deadlines are approaching, during which there can be a large number of clients.

Bistro attempts to relieve hot spots in upload applications. In the design of Bistro, we take advantage of the fact that data are usually not consumed right after the deadline. Instead, the application does not want senders to change their data after the deadline, e.g., to achieve fairness. Therefore, as long as the data remains unchanged after the deadline, data transfer can take place later within a reasonable amount of time.

In Bistro, an upload process is broken down into three steps. Figure 1.1, taken from [4], depicts upload processes with and without Bistro. First, in the timestamp step, clients send hashes of their files to the server, and obtain timestamps. These timestamps clock clients' submission time. After this point, clients cannot change their data without the server detecting this. In the data transfer step, clients send their data to intermediaries called bistros. Note that bistros are not trusted, so clients encrypt their files to prevent unauthorized access. In the last step, called the data collection step, the server coordinates bistros to transfer clients' data to itself. The server then matches the hashes of the received files against the hashes it received directly from the clients. The server accepts

files that pass this test, and asks the clients to resubmit otherwise. This completes the upload procedure.

We are interested in developing and analyzing a fault tolerance protocol in this thesis in the context of the Bistro architecture. In the original implementation of Bistro, if bistros are not available at the data collection step due to, for examples, power failures, disk failures or network problems, all files on the unavailable bistros are lost, hence the destination server needs to ask clients to resubmit. In addition, malicious bistros can intentionally corrupt data. Although destination server can detect corrupted data from the hash check, it has no way of recovering the data; hence the destination server needs to ask clients to resubmit. In this work, we are interested in using forward error correction techniques to recover corrupted or lost data. The fault tolerance protocol, on the other hand, brings in additional storage and network transfer costs for the redundant data as a result of employing the protocol. The goal of this thesis is to provide better performance when intermediaries fails while minimizing the amount of redundant data brought on by the fault tolerance protocol.

We propose analytical models to evaluate our fault tolerance protocol. In particular, we develop reliability models to analyze the reliability characteristics of bistros. We also derive performance models to estimate the performance penalty of employing our protocol. We study the tradeoff between reliability and performance using a cost function.

This thesis is organized as follows. We discuss related work in Chapter 2. We give an overview of erasure code in Chapter 3. Chapter 4 describes our fault tolerance protocol. We derive analytical models for this protocol in Chapter 5. Chapter 6 presents some results showing the tradeoff between performance and reliability characteristics of our protocol. We describe future work in Chapter 7. Finally, we conclude in Chapter 8.

Chapter 2

Related Work

We focus on fault tolerance issues in Bistro framework, a wide area upload architecture, and we propose to use erasure codes to provide fault tolerance in such systems. This chapter describes fault tolerance in other large-scale data transfer applications, and discusses other uses of erasure codes in the context of computer networking.

2.1 Fault Tolerance in Distributed Systems

In the context of download applications, one approach to achieve fault tolerance is through service replication. Replication of DNS servers [22, 23] is one such example. The root directory servers are replicated, so if any root server fails, DNS service is still available. Each ISP is likely to host a number of DNS server, and most clients are configured with primary and alternate DNS servers. Therefore, even if some DNS servers fail, clients can contact an alternate DNS server to make DNS lookup requests. In Bistro, the service of intermediaries is replicated, where intermediaries provide interim storages of data until the destination server retrieves it.

In storage systems [31, 16, 15], data replication techniques, such as RAID techniques [26], are commonly used for providing better fault tolerance characteristics. In

case of disk failures, file servers are able to reconstruct data on the failed disk once the failed disk is replaced, and data is available even before replacing the failed disks. Although data replication can provide better fault tolerance characteristics, the storage overhead is high. For example, the storage overhead of mirroring is 100%. We are interested in providing fault tolerance with smaller storage overhead in this work.

Some caching techniques can be used to improve fault tolerance. Disconnected operations in Coda [17] makes use of clients' local cache to achieve better fault tolerance. When client accesses a file in Coda, the file server sends the whole file to the client, and allows the client to keep a local cache copy of the file in local storage. When the file server fails, the client can still access the files in local storage, and hence allow work to be performed in the event of file server failure. In upload applications, however, caching is not feasible because the destination server reads the data only once.

For fault tolerance reasons, recent research in distributed storage systems has been moving from centralized servers to serverless systems; [13, 3, 30, 18, 28] are examples of such systems. A centralized file server has a single point of failure, as the system cannot operate once the file server breaks down. Using a peer-to-peer communication model to provide file service can provide better fault tolerance because failure of one server is assumed to be independent from failures of other servers. By combining with this replication techniques such as RAID-type striping, these systems are able to provide high availability of data. The destination server is a single point of failure in the Bistro architecture. Once the destination server fails, the entire system breaks down. Future research should investigate how to eliminate this problem in the Bistro framework.

2.2 Erasure Codes in Computer Networking

There are a lot of uses of erasure codes in computer networking. A number of multicast applications employ erasure codes to protect against losses [19, 25]. When there are losses in multicast applications, clients can either tolerate the losses, or attempt to recover the packets. One way to recover lost packets is to ask for retransmissions, but this solution is not scalable when there are a large number of clients. Using forward error correction techniques allows clients to reconstruct lost packets without contacting the sender.

Erasure codes are also useful in bulk data distribution, e.g., [6] describes a way to use erasure codes where clients can reconstruct the data as long as a small fraction of erasure-encoded files are received. This scheme allows clients to choose from a large set of servers, resulting in good fault tolerance and better performance characteristics than traditional bulk data distribution techniques.

In wireless networking, using forward error correction techniques can reduce packet loss rates by recovering parts of lost packets [2, 10, 21]. Packet loss rates in wireless networks are much higher because propagation errors occur more frequently when the data is transmitted through air. Employing forward error correction techniques can improve reliability and reduce retransmissions.

Multimedia streaming also employs erasure codes [14, 24, 11]. Retransmitting packets may not be feasible because streaming applications are usually time-critical, so streaming applications must often operate under packet losses. However, when a lot of packets are lost, the quality of the video is often poor. Using erasure codes to recover parts of lost packets can improve the quality of the video.

These applications of erasure codes assume that packets are either received successfully or are lost. They assume that there are other ways to detect corrupted packets. e.g., using TCP checksums. In Bistro, however, this assumption is not valid because packets

can be intentionally corrupted by intermediate bistro. In Chapter 4, we describe one way to detect corrupted packets using checksums such that we treat corrupted packets as losses.

Chapter 3

Overview of Erasure Codes

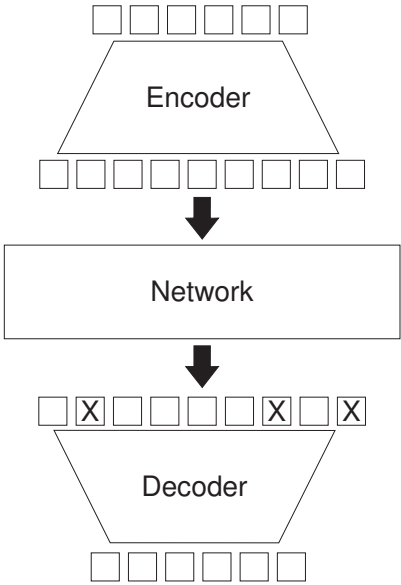


Figure 3.1: Illustration of an Erasure Code

This chapter provides an overview of erasure codes. An erasure code takes a file of k data packets, adds $(n - k)$ parity packets, and creates a n -packet encoded file. After encoding, a sender typically transmits the encoded file using a number of channels. A receiver can use an erasure code decoder to reproduce the original file as long as any k packets are received with no errors. Figure 3.1 illustrates this process.

Many erasure codes are based on linear algebra theory. Let x_i be a data packet, and let $x = x_0x_1 \dots x_{k-1}$ be the original data. An encoding procedure produces encoded data $y = y_0y_1 \dots y_{n-1}$ by multiplying a generator matrix $G = g_{ij}$ and x , i.e.,

$$y = Gx$$

At the receiver side, assume that some of the y_i s are lost. Let $z = z_0z_1 \dots z_{k-1}$ be the received data. Let $H = \{g_{ij} | y_i \text{ is received}\}$. That is, if y_r is received, we add the r th row in G to H . Hence, H is a $k \times k$ matrix. Note that

$$z = Hx$$

So,

$$x = H^{-1}z$$

Therefore, we can reconstruct the original data by multiplying H^{-1} and z . This procedure requires rows of H to be linearly independent, which is true when H is invertible. Note that an erasure code decoder assumes that all received packets are correct. That is, it does not correct corrupted data; it only recovers lost data.

Some erasure codes make use of Vandermonde matrices; this is based on finite field theory. Vandermonde matrices are used in traditional error-correcting codes such as Reed-Solomon Codes. However, these codes are not efficient since finite field operations are very expensive; [29] is one of such example.

Other erasure codes take advantage of the fact that finite field operations in erasure codes can be reduced to XOR operations. This can improve the performance of erasure codes because XOR operations are much more efficient than finite field operations; [5] is one such implementation. This code uses Cauchy matrices where there is an efficient

algorithm for inverting such matrices. Recall that we invert H in the decoding procedure, so making such operations efficient can improve the performance of the code. For instance, [12] develops a Reed-Solomon-like erasure code which uses only XOR operations. Instead of using Cauchy matrices, this code explores the structure of an RS-based erasure code and gives an improved algorithm to perform encoding and decoding.

Tornado Codes [20] are another example of erasure codes. Unlike other erasure codes that are based on linear algebra, Tornado Codes are based on bipartite graphs. Let $G(V, E)$ be a bipartite graph. Let S be a set of vertices representing the original data, and let P be a set of vertices representing parity data, where $V = S \cup P$. Let $R_j = \{s_i \in S \mid (s_i, p_j) \in E\}$. Encoding is done by performing XOR on every element in R_j and producing p_j for all j . Decoding is done in a similar fashion, in which XOR operations are performed on received data to find the lost data. This is an efficient implementation since it uses only XOR operations. It, however, requires slightly more than k packets to guarantee that the decoder is able to construct the original data.

Chapter 4

Fault Tolerance Protocol

This chapter provides details of our fault tolerance protocol. The protocol is broken down into three parts as in the original Bistro protocol described in [8]. The timestamp step verifies clients' submissions. Actual data transfer is done in the data transfer step, in which clients stripe their files across a number of bistros, instead of sending their files to one bistro as in the original protocol. In the data collection step, the destination server coordinates data transfers from intermediaries to itself. Note that only the timestamp step has to be done before the real life deadline, since our protocol can detect if the files have changed after the deadline. File transfer can be done later as any changes to the file after the issuing of timestamp can be detected. We are going to provide details about each step in this chapter with focus on fault tolerance aspects, and discuss related design decisions.

4.1 Timestamp Step

The timestamp step verifies clients' submissions. Clients generate hashes of their data and send them over to the destination server. The destination server replies to clients

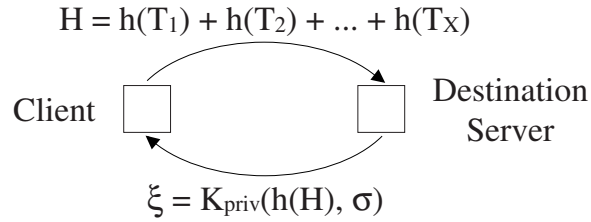


Figure 4.1: Timestamp Step

with tickets, which consist of timestamps and the hashes messages clients have just sent. Figure 4.1 depicts the timestamp step.

In the original protocol, clients send a checksum of the whole file to the destination server in the timestamp step. If any packets are lost or corrupted, the checksum check would fail, and the destination server would have to discard all packets that correspond to that checksum because it does not know which packets are corrupted. This would mean that losing any packet would result in retransmissions in the original protocol.

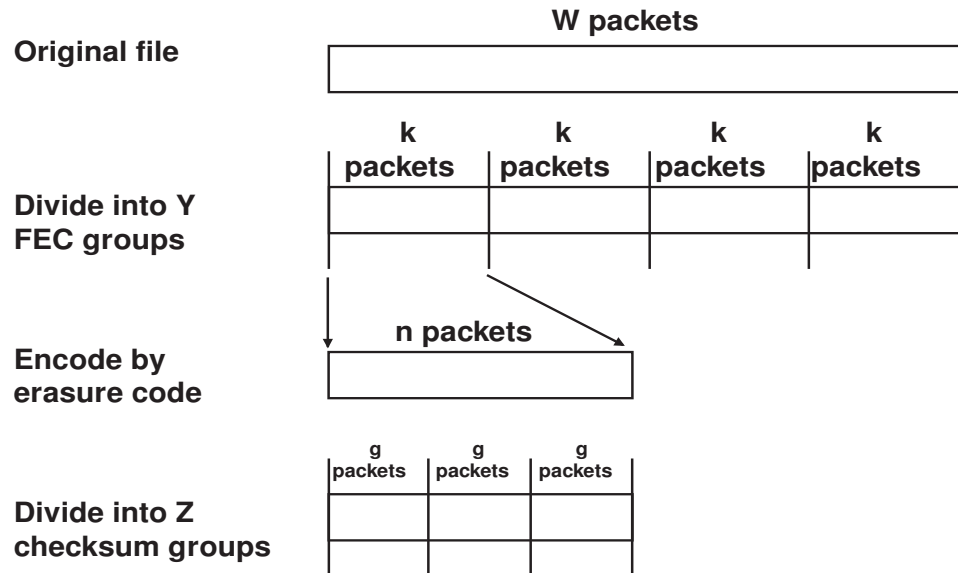


Figure 4.2: FEC Groups and Checksum Groups

Variable	Meaning
T_o	The original file
T	The original file encoded by erasure code
W	Total number of packet in the original file
X	Number of checksums in the timestamp request message
Y	Number of data packets in a FEC group
Z	Number of checksum groups in a FEC group
n	Number of data packets and parity packets in a FEC group
k	Number of data packets in a FEC group

Table 4.1: Meaning of Variables

To solve this problem, we send multiple checksums in the fault tolerance protocol. Assume that each client has W data packets to send. The data packets are divided into Y FEC (forward error correction) groups of k packets each. For each FEC group, a client encodes k data packets into n packets (data + parity), arranges the n packets into Z checksum groups each of size g , and generates one checksum for each checksum group using a message digest algorithm such as SHA1. We assume that Z is a factor of g , because we want the size of all checksum groups to be the same, which can simplify our reliability model in Chapter 5. There are altogether $X = YZ$ checksums, which are concatenated and sent in one message to the destination server. Figure 4.2 illustrates the relationship between FEC groups and checksum groups. Table 4.1 summarizes the meaning of variables we use in this context.

Note that the size of a checksum group has to be smaller than number of data packets per FEC group ($g < k$). Recall that erasure codes do not correct corrupted packets, so we drop all packets in a checksum group if any packet within the checksum group is lost or corrupted, and then we try to recover the dropped packets using an erasure code. If $g \geq k$ and if a checksum group is dropped, then we lose more than k packets in at least one FEC group, which we are not able to recover because less than k packets within that FEC group are received. Hence we have to ask for retransmissions if any packet in the

file is lost or corrupted. So, if $g \geq k$, we are back to the problem of the original protocol that losing any packet would result in retransmissions. The above argument also implies that there must be at least two checksum groups per FEC group.

This raises an interesting question. To provide better fault tolerance, we should choose Z to be n , i.e., we generate a checksum for every packet, so losing one packet does not affect other packets. However, in the timestamp step, if we generate a checksum for every packet, the size of the message to be sent to the destination server increases, and hence more network resources would be used. This problem is exacerbated when a large number of clients try to send messages to the destination server moments before the real life deadline, i.e., the original problem solved by Bistro. We derive a cost function to study this tradeoff in Chapter 5.

Also, note that erasure codes can keep replication as a special case when we set n to be multiples of k . For example, if we set $n = 2k$, we can consider this scheme to be sending two copies of the original file to intermediaries.

Timestamp Step Algorithm

1. Client divides his original file, T_o , into Y FEC groups, each of size k . Then client passes T_o to an erasure code encoder to get an encoded file, T , where size of each FEC group in T is n .
2. The encoded file T is divided into X parts ($T = T_1 + T_2 \cdots + T_X$), and client generates checksums $h(T_i)$ for all i using a message digest algorithm such as SHA1.
3. Client concatenates checksums he generated in the previous step, and send the result, H , to the destination server.

$$H = h(T_1) + h(T_2) + \cdots + h(T_X)$$

4. Upon receiving the message, destination server generates a timestamp σ .

5. Destination server stores information about the client, the received checksums, and timestamp it has just produced into a local database.
6. Destination server computes the hash value of the received message $h(H)$.
7. Destination server concatenates the timestamp σ and hash $h(H)$, digitally signs them with its private key, and sends this message, i.e., tickets ξ , to client.

$$\xi = K_{priv}(h(H), \sigma)$$

4.2 Data Transfer Step

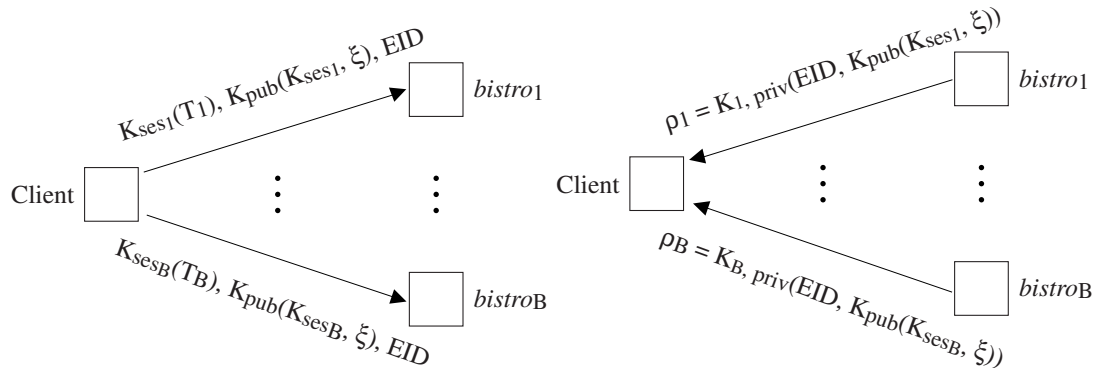


Figure 4.3: Data Transfer Step

In the data transfer step, clients send their files to intermediate bistros which are not trusted. Upon receiving the data from clients, bistros send receipts to clients and the destination server to verify their submissions. Figure 4.3 depicts the data transfer step.

In [9], the assignment problem is studied. That is, how a client should choose a bistro to which it sends its file to. However, in that case, only one bistro out of a pool of bistros is chosen. In the case of striping, a client needs to choose $B \geq 1$ bistros. We leave the choice of which B bistro clients should stripe its file to, and how clients determine the value of B to future work.

In our fault tolerance protocol, we stripe the data across a number of bistros instead of sending the whole file to one bistro, as in the original protocol. [27] suggests that data dispersal can provide better fault tolerance, if failure of one storage device is independent of failure of other storage devices in the system. Note, we treat failures and malicious behavior similarly.

Since we do not trust intermediate bistros, clients need to encrypt their data to protect it against unauthorized accesses or modifications. This property is inherited from the original protocol, except that we need to generate a number of session keys instead of just one since we are striping files across a number of bistros.

Data Transfer Step Algorithm

1. Client chooses B bistros to send their data to.
2. Client generates a session key K_{ses_i} for each bistro it has chosen.
3. Client divides the file into B parts. For each part of the file, client encrypts it with a session key K_{ses_i} , and sends that part to intermediate bistros i . Client also sends bistro i session key K_{ses_i} and ticket ξ encrypted with public key of destination server.

$$K_{ses_i}(T_i), K_{pub}(K_{ses_i}, \xi), EID$$

4. Each bistro i generates a receipt ρ_i and sends it to both client and destination server.

$$\rho_i = K_{i,priv}(K_{pub}(K_{ses_i}, \xi)), K_{i,pub}$$

4.3 Data Collection Step

In the data collection step, the destination server coordinates intermediate bistros to collect data. Figure 4.4 depicts this step.

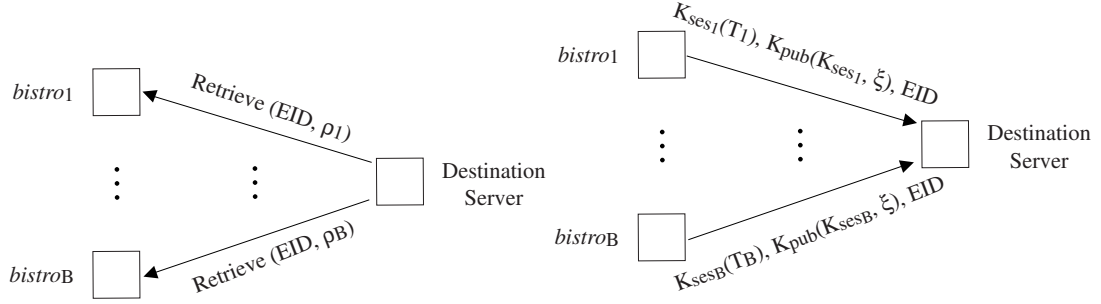


Figure 4.4: Data Collection Step

When an erasure code is used, we do not always need to collect all the data as some of it is redundant. We only need k out of n packets from each FEC group in a file. After receiving k packets for each FEC group, the destination server has two choices on reconstructing clients' data. First, the destination server can pass the received packets to an erasure code decoder. Second, the destination can ask intermediaries to transfer the remaining data. There is a tradeoff between computational costs of erasure code decoding and network resource requirements. The first scheme involves more computation costs while the second scheme requires more network bandwidth. We leave the study of this tradeoff as future work, and in this thesis we assume that the destination server collects all packets for every file.

Data Collection Step Algorithm

1. When destination server wants to retrieve data from bistro i , it sends a retrieval request along with receipt ρ_i .

$\text{Retrieve}(EID, \rho_i)$

2. Upon receiving retrieval requests from destination server, bistro i sends the file T_i along with the encrypted session key and ticket for decryption.

$K_{pub}(K_{ses_i}, \xi), K_{ses_i}(T_i), EID$

3. When the destination server receives a message, it retrieves the session key by decrypting the message using its private key. It then decrypts T_i using the session key.
4. When all packets within a checksum group are received, destination server computes a checksum of the received checksum group. It then matches this checksum with what it received during the timestamp step. If these two checksums match, the destination server accepts all packets in the checksum group, and discards them otherwise.
5. After the destination server has retrieved data from all intermediate bistros, it passes the packets that pass the checksum check to an erasure code decoder if it has received at least k packets from every FEC group, and the erasure code decoder reconstructs the original file T_o . If the destination server receives less than k packets from any FEC group, it contacts the clients and asks for resubmitting the lost data.

Chapter 5

Analytical Models

We propose analytical models to evaluate our fault tolerance protocol in this chapter. We derive reliability models to study how reliability characteristics of bistros affect system reliability. We also derive a performance model to estimate the performance penalty of employing our protocol. Lastly, we derive a cost function to study the tradeoff between reliability and performance.

5.1 Modeling Reliability of Checksum Groups

We begin the reliability models discussion by considering the reliability of checksum groups. Recall that if a checksum check fails, all packets within that checksum group are discarded because we have no way of determining which of the packets are corrupted.

Let p_g be the probability that there is no loss within a checksum group. Hence, the probability that at least one packet is lost within a checksum group is $1 - p_g$.

In the following sections, we are interested in deriving p_g using different reliability models. These models make different assumptions about the packet loss characteristics and corruption characteristics.

5.1.1 Independent Packet Losses

In the independent packet loss model, we assume that losing or corrupting one packet is independent of losing or corrupting other packets within the same checksum group. This is a good model to analyze packet losses and corruptions if we have no information about how striping is done. Let the probability of losing a packet, p , be the probability that a packet is lost or corrupted. Then,

$$p_g = (1 - p)^g$$

This model does not allow correlation between consecutive packet losses. For example, if a bistro is malicious, given that a packet is corrupted, the probability of the next packet from the same bistro being corrupted weight higher. We describe Gilbert Model in 5.1.3 to capture this correlation.

5.1.2 Independent Bistro Failures

This model assumes that all packets in the same checksum group are sent to the same bistro. We also assume that failure of one bistro is independent of failure of other bistros. So, if a bistro fails, all packets on that bistro are lost. This model attempts to illustrate this effect.

Let p_f be the probability that a bistro fails or is malicious. So, the probability that the whole checksum group is sent over successfully is

$$p_g = 1 - p_f$$

This model makes an assumption that malicious bistros always corrupt all packets, which is not the case since bistros do not have to corrupt all packets to be considered malicious.

5.1.3 Gilbert Model

The Gilbert Model takes the middle ground between the two models we have just described. This model allows correlations between lost or corrupted packets. Although it is typically used to model network losses, we believe it is also a good model for understanding reliability characteristics of bistros.

Gilbert Model is a two-state Markov chain. Being in state 0 means that the previous packet is not lost, while being in state 1 means that the previous packet is lost. Figure 5.1 depicts the discrete time version of the Gilbert Model.

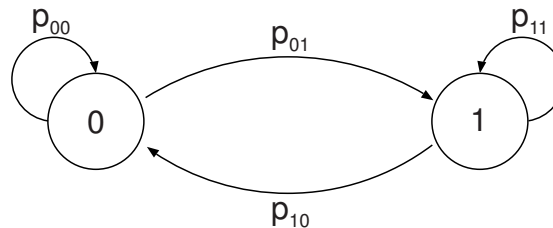


Figure 5.1: Gilbert Model

Let π_0 be the steady state probability of state 0, and π_1 be the steady state probability of state 1. Solving the Markov chain we have

$$\begin{aligned}\pi_0 &= \frac{p_{10}}{p_{10} + p_{01}} \\ \pi_1 &= \frac{p_{01}}{p_{10} + p_{01}}\end{aligned}$$

Let $P(a, b)$ be the probability that a packets are sent by the clients to the destination server via bistro i , and b of them are lost or corrupted. We can define $P(a, b)$ recursively as follows.

$$P(a, b) = \pi_0 P(a - 1, b) + \pi_1 P(a - 1, b - 1)$$

The boundary conditions are given as follow.

$$P(a, b) = 0 \quad \text{if } a < b \text{ or } a \leq 0 \text{ or } b \leq 0$$

$$P(1, 0) = \pi_0 p_{00} + \pi_1 p_{10}$$

$$P(1, 1) = \pi_0 p_{01} + \pi_1 p_{11}$$

Assuming we send all packets in a checksum group to a bistro, the probability of no loss or corruption within a checksum group is given by

$$p_g = P(g, 0)$$

The Gilbert Model degenerates to the independent packet loss model in Section 5.1.1 when $p_{00} = 1 - p$, $p_{01} = p$, $p_{10} = 1 - p$, and $p_{11} = p$. This says no matter which state the system is in, we have same probability of losing the current packet.

If we add an additional state s that has probability of $1 - p_f$ to go to state 0, and probability of p_f to go to state 1, and we set $p_{00} = 1$, $p_{01} = 0$, $p_{10} = 0$, and $p_{11} = 1$, we can represent the bistro fails model in Section 5.1.2.

5.2 Overall Reliability Model

Let V be a random variable that represents the number of checksum groups that pass the checksum check within a FEC group, then

$$P(V = v) = \binom{Z}{v} p_g^v (1 - p_g)^{Z-v}$$

In other words, V is a binomial random variable with parameters Z and p_g . The minimum number of checksum groups required to reconstruct a FEC group is given by $\lceil \frac{k}{g} \rceil$ because we need at least k packets and the packets are organized in groups of size g . As a result, the probability that the destination server is able to reconstruct a FEC group is given by

$$\begin{aligned} P(V \geq \lceil \frac{k}{g} \rceil) &= \sum_{i=\lceil \frac{k}{g} \rceil}^Z P(V = i) \\ &= \sum_{i=\lceil \frac{k}{g} \rceil}^Z \binom{Z}{i} p_g^i (1 - p_g)^{Z-i} \end{aligned}$$

We are also interested in finding the expected data packets lost within a FEC group, $L(j)$, where j is the number of checksum groups lost. In other words, $L(j)$ is the expected number of data packets that cannot be recovered by erasure codes. If k or more packets are received from a FEC group, then $L(j)$ is 0 because erasure codes can recover all data packets.

The content of checksum groups can be classified as follows.

- Contain only data packets. There are $\lfloor \frac{k}{g} \rfloor$ such checksum groups.

Content	Quantity	# data packets
Data Packets only	$\lfloor \frac{k}{g} \rfloor$	g
Data Packets and Parity Packets	0, if k is divisible by g 1, otherwise	$k \bmod g$
Parity Packets Only	$\lfloor \frac{n-k}{g} \rfloor$	0

Table 5.1: Summary of Content of Checksum Groups

- Contain both data packets and parity packets. If k is divisible by g , there are no such checksum groups. Otherwise, there is exactly one such checksum group.
- Contain only parity packets. There are $\lfloor \frac{n-k}{g} \rfloor$ such checksum groups.

In the first case, the number of data packets per FEC group is g . In the second case, the number of data packets per FEC group is $k \bmod g$. In the last case, the number of data packets per FEC group is 0 as there are only parity packets in that FEC group. Table 5.1 summarizes this information.

Assume that k is a multiple of g . As a result, there is no checksum group that contains both data and parity packets. Let $N(j)$ be the number of different ways to distribute checksum group losses among the two classes of checksum group listed in Table 5.1 given that j out of Z checksum groups are lost. Hence,

$$N(j) = \sum_{x=0}^{\lfloor \frac{k}{g} \rfloor} \binom{\lfloor \frac{k}{g} \rfloor}{x} \binom{\lfloor \frac{n-k}{g} \rfloor}{j-x}$$

So,

$$L(j) = \sum_{x=0}^{\lfloor \frac{k}{g} \rfloor} \binom{\lfloor \frac{k}{g} \rfloor}{x} \binom{\lfloor \frac{n-k}{g} \rfloor}{j-x} xg(1-p_g)^j p_g^{(Z-j)}$$

Now assume that k is not a multiple of g , and we are given that j checksum groups are lost. Therefore, we need to distribute j losses among all three classes of checksum group. Let x be the number of checksum groups lost that contains only data packets, and let y be the number of checksum groups lost that contains both data packets and parity packets. So, the number of ways to distribute j checksum groups lost is

$$\begin{aligned} N(j) &= \sum_{x=0}^{\lfloor \frac{k}{g} \rfloor} \sum_{y=0}^1 \binom{\lfloor \frac{k}{g} \rfloor}{x} \binom{1}{y} \binom{\lfloor \frac{n-k}{g} \rfloor}{j-x-y} \\ &= \sum_{x=0}^{\lfloor \frac{k}{g} \rfloor} \sum_{y=0}^1 \binom{\lfloor \frac{k}{g} \rfloor}{x} \binom{\lfloor \frac{n-k}{g} \rfloor}{j-x-y} \end{aligned}$$

So,

$$L(j) = \sum_{x=0}^{\lfloor \frac{k}{g} \rfloor} \sum_{y=0}^1 \binom{\lfloor \frac{k}{g} \rfloor}{x} \binom{\lfloor \frac{n-k}{g} \rfloor}{j-x-y} (xg + y(k \bmod g))(1 - p_g)^j p_g^{(Z-j)}$$

Expected data packet losses per FEC group L is given by

$$L = \sum_{j=\lfloor \frac{n-k}{g} \rfloor + 1}^Z L(j)$$

Hence, the average data packet loss rate per FEC group L_{rate} is given by

$$L_{rate} = \frac{L}{k}$$

The whole file is transferred successfully if all FEC groups can be reconstructed. So, we need at least k packets from each of Y FEC group. Therefore, the probability that the file is transferred successfully is $(P(V \geq \lceil \frac{k}{g} \rceil))^Y$, and the probability that part of a

file is lost is given by $1 - (P(V \geq \lceil \frac{k}{g} \rceil))^Y$. The average data packets loss rate for the whole file is L_{rate} because the quantity is normalized.

5.3 Performance Model

This section describes performance models to evaluate our fault tolerance protocol. We consider the performance penalty in the timestamp step only. If the performance of timestamp step is poor, then we are back to the original problem where many clients are trying to send large files to a server at the same time. In the data transfer step, performance is not as critical because clients are less likely to overload intermediate bistros with striping. In the data collection step, the destination server is able to coordinate bistros, so performance is not as critical as in the timestamp step as there is no hard deadline.

5.3.1 Server Performance in the Timestamp Step

In this section, we are interested in the performance of the computation needed at the destination server in the timestamp step. When the destination server receives a timestamp request message from a client, it computes a hash value of the checksums generated by that client, and digitally signs the reply message, which we term as the ticket.

Let t_{ds} be the average time the destination server takes to digitally sign a ticket, and let $t_h(x)$ be the average time to compute a hash value of a timestamp request message consisting of x checksums. Notice that t_{ds} is independent of the number of checksums we generate, because the size of the return message is fixed, which is the size of the hash of the original message plus the size of the timestamp. Total time a destination server needs to compute a reply to a message is $t(x) = t_{ds} + t_h(x)$. Results from a

previous work [7] suggest that t_{ds} is approximately 0.0041s on an 800 MHZ Pentium-III PC running Linux, and we are interested in looking at how $t_h(x)$ changes when a client sends different number of checksums.

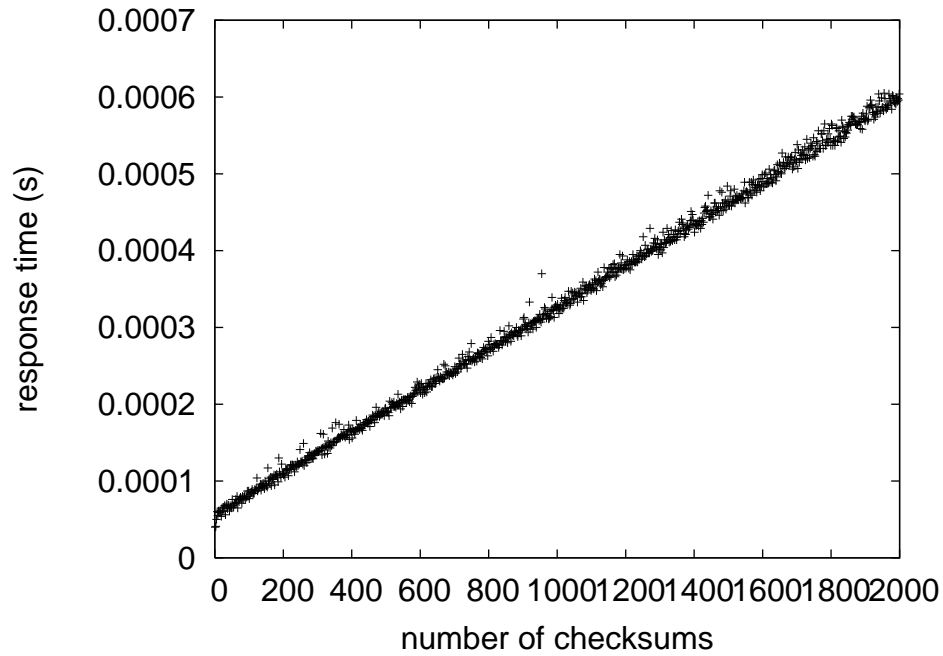


Figure 5.2: Average Time to Compute Hash of Different Number of Checksums

In order to estimate $t_h(x)$, we emulated the destination server by running OpenSSL on different number of checksums on an 800 MHZ Pentium-III PC running Linux. Figure 5.2 shows the time taken to compute SHA1 hashes in our simulation. From Figure 5.2, even if clients send 2000 checksums to the destination server, it takes about 0.0006s to computer a hash of the checksums, which is an order of magnitude faster than producing a digital signature. Thus we believe that the generation of a hash of multiple checksums should not overload the destination server.

5.3.2 Size of timestamp messages

It is likely that overloading of network resources, due to sending a greater number of checksums, is more important than the additional computations needed on the server. If every client sends 2000 checksums to the destination server, and clients use the SHA1 message digest algorithm which produces a 20-byte checksum, every client will send about 40KB of data in the timestamp step. This might overload network resources around the deadline time, i.e., we would be back to the original problem of a large number of clients trying to send large amounts of data to the server in a short period of time. As a result, we use the size of the timestamp message as a metric for determining the performance drawbacks of our scheme.

The total number of checksums a client sends is given by $X = YZ$. Now, we want to find a normalized metric to measure the size of a timestamp message, because we do not want to penalize large files for sending more checksums than small files.

One possible metric is the number of checksums normalized by the file size. This quantity is given by $\frac{YZ}{Yk}$, and hence, $\frac{Z}{k}$. This is the number of checksum groups per data packet. In what follows, we use this quantity as a metric for evaluating the performance penalties of our fault tolerance protocol.

5.4 Cost function

Now that we have a reliability model and a performance model, the question is how to combine the effects of from both in order to study the tradeoff between reliability and performance. This section describes a cost function which we propose to use to achieve this goal.

Let C_1 be the cost computed using the reliability model, and let C_2 be the cost computed using the performance model in the timestamp step. Thus, our cost function is

$$C = w_1 C_1 + w_2 C_2$$

where w_1 and w_2 are weights of each factor.

We derive two different costs from the reliability model, namely the probability of losing part of a file and the average data packet loss rate. We can use either for C_1 and evaluate the differences between these two metrics. The performance cost in the timestamp step is given by the number of checksum groups per data packet. In the next chapter, we will study how each parameter affects the overall cost function.

Chapter 6

Results

This chapter provides results on varying different parameters of the cost function discussed in the last chapter. Parameters of interest are as follows.

1. Number of checksum groups per FEC group, Z . We mentioned this tradeoff in Chapter 4. Setting Z to be large can provide better reliability because losing a packet affects fewer packets, as we drop the entire checksum group whenever any packets from that group is lost or corrupted. On the other hand, large values of Z result in large timestamp messages, which can have adverse effects on network resources.
2. Number of parity packets per FEC group ($n - k$). For reliability reasons, we want to send a large number of parity packets, but this increases the number of checksums we send as we are interested in adding parity checksum groups.
3. Number of data packets per FEC group k . Given a file of W packets, we want to study differences in dividing the file into few large FEC groups or many small FEC groups. Dividing files into many small FEC groups has better fault tolerance, but clients send more checksums which makes the timestamp request message

large because we require that clients produce at least two checksums per FEC group.

4. Probability of losing a packet, p . We want to see how sensitive the cost function is to p .

All results presented in this chapter use the independent packet loss model because of its simplicity. Other reliability models give similar results. We use both reliability metrics, which are probability of losing part of a file and the average data packet loss rate. The performance metric used is the number of checksums per data packet. In each of the results, we show graphs on each reliability metrics, the performance metric, and the cost function computed using each of the reliability metrics.

6.1 Setting Weights

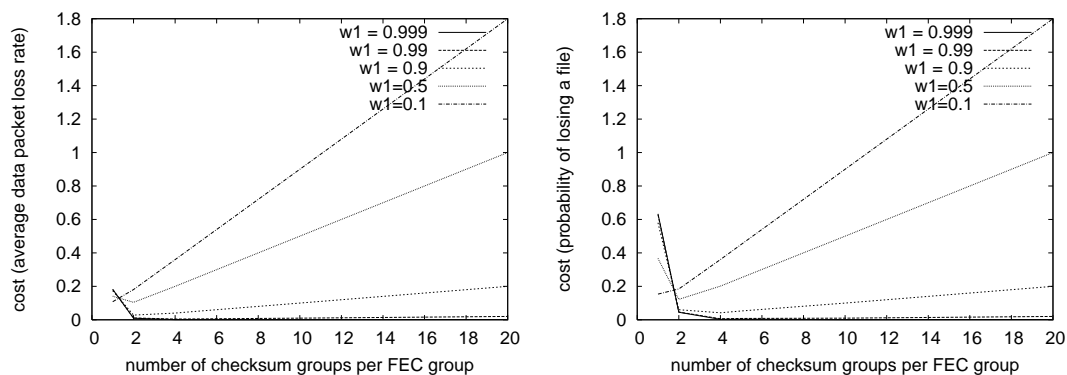


Figure 6.1: Cost Function - varying weights

In the section, we study how to set the weights in the cost function in order to obtain a convex curve to study the tradeoff. Recall that w_1 is the weight corresponding to a reliability metric, and w_2 is the weight corresponding to the performance metric, and

$w_1 + w_2 = 1$. We depict the results as a function of Z ; we plot these results by setting w_1 to 0.1, 0.5, 0.9, 0.99, and 0.999. Figure 6.1 illustrates these results.

From the graphs, we can see that when $w_1 = 0.1$, the cost strictly increasing. When $w_1 = 0.5$, the cost is very close to a strictly increasing line. We observe that the cost is a convex curve when $w_1 = 0.9$, and the cost becomes strictly decreasing when w_1 is 0.99 and 0.999.

Values of both reliability metrics is between 0 and 1, and these values approach 0 when Z increases. On the other hand, values of checksum per data packet ranging from 0 to 2, and are able to go further when we keep increasing Z . As a result, in order to obtain a convex curve, we need to set w_1 to be around 0.9.

6.2 Varying the Number of Checksum Groups in a FEC Group

We now study the effect of Z . Setting Z to be large can provide good reliability, because losing one packet can lead to drop fewer packets. On the other hand, large values of Z would make the timestamp message large, hence we are more likely to overload network resources.

Our results are plotted in Figure 6.2, with $Y = 5$, $n = 20$, $k = 10$, $p = 0.01$, $w_1 = 0.9$, and $w_2 = 0.1$. The two upper graphs show how reliability metrics change with Z . Both reliability metrics drop dramatically when Z is between 1 and 2, and are close to 0 when $Z \geq 4$. This is because when $Z \geq 2$, clients are able to reconstruct a FEC group even if some checksum groups are dropped. Checksums per data packet increase linearly, because Z increase linearly while k is fixed. The cost is high when Z is small because the probability of losing part of a file is high. The cost decreases when

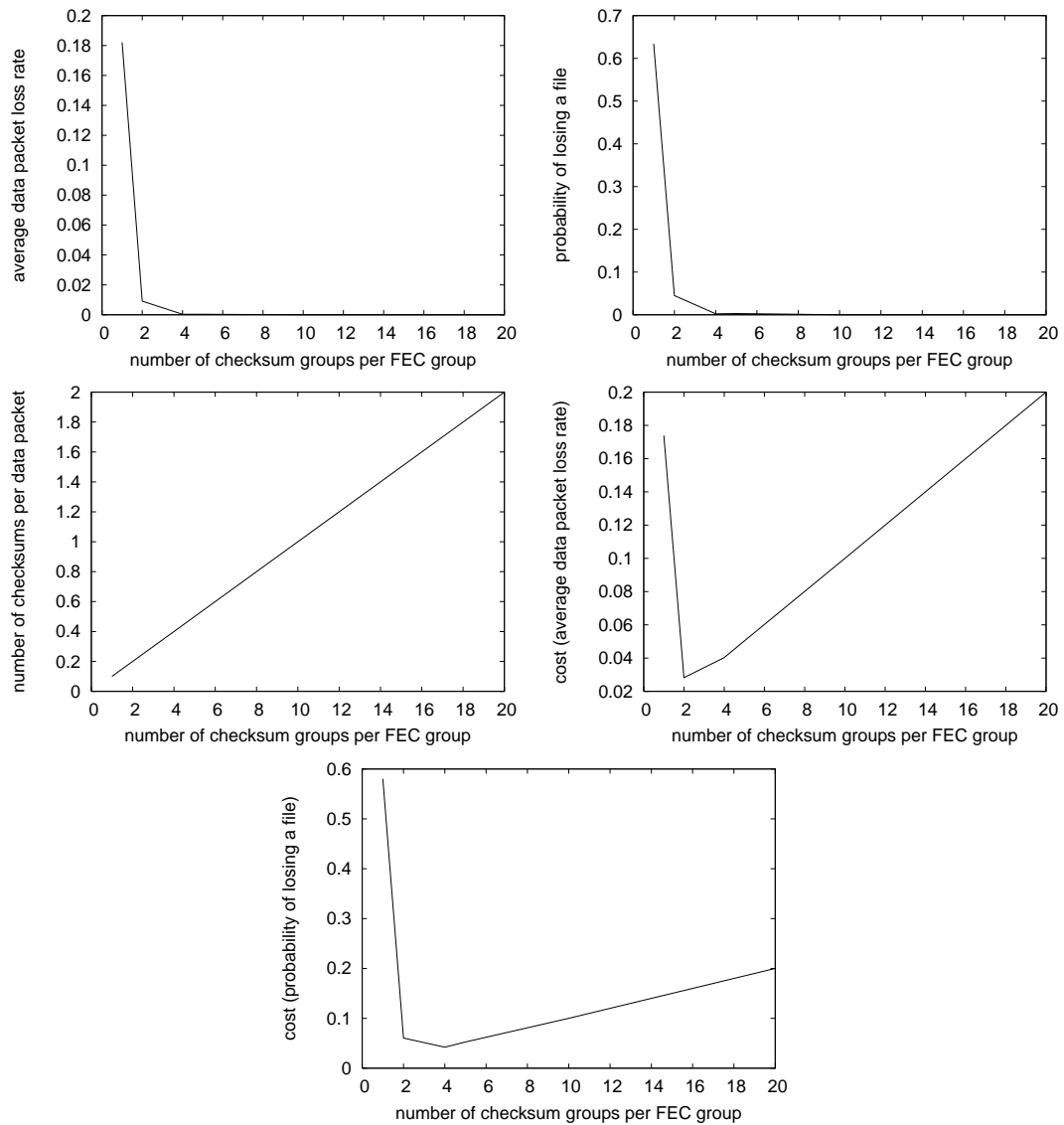


Figure 6.2: Cost Function - varying Z

Z is between 1 and 2 because the probability of losing part of a file is improving. At $Z \geq 2$, the cost goes up again because the size of the message is too large.

6.3 Varying the Number of Parity Packets in a FEC Group

Group

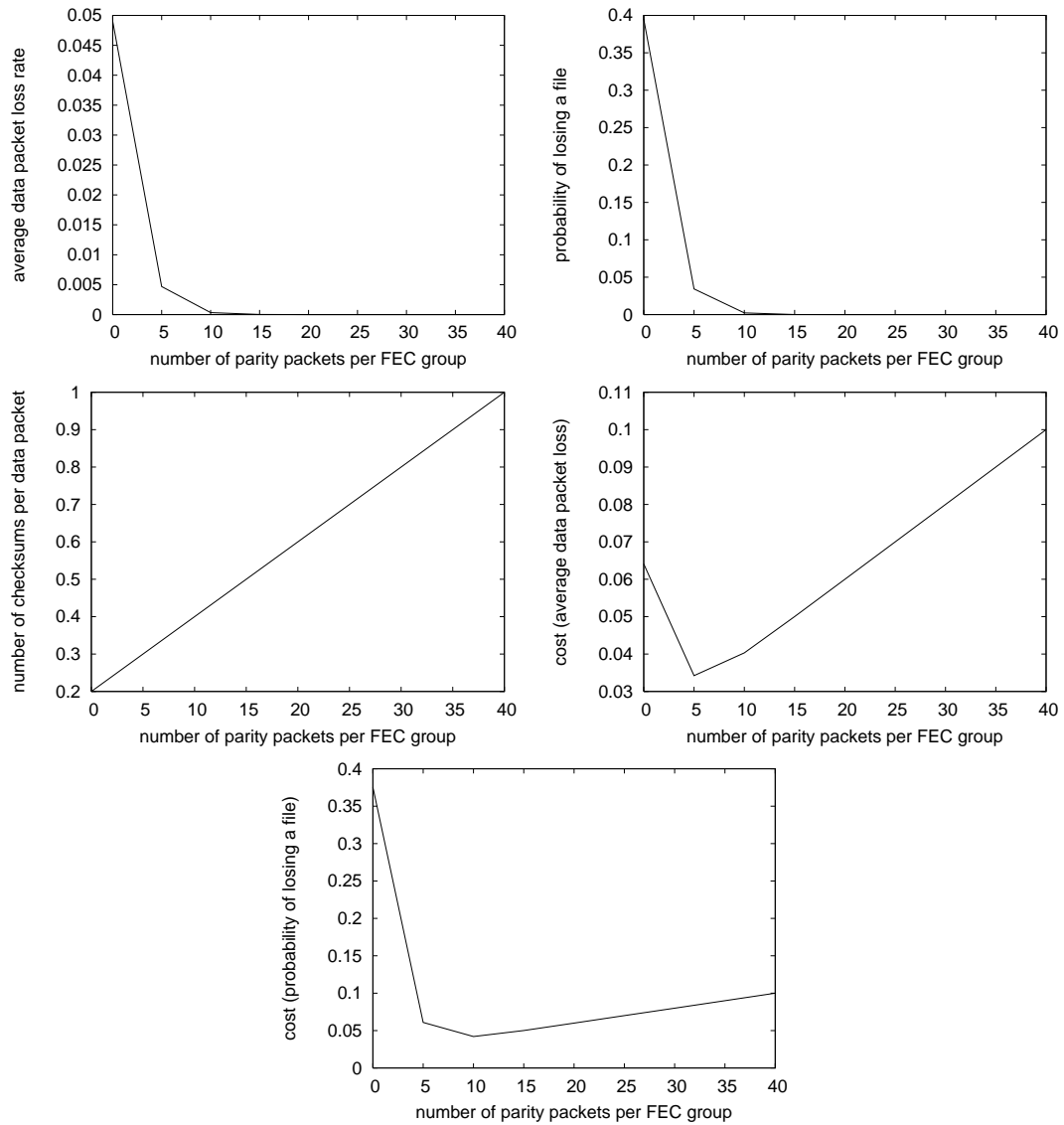


Figure 6.3: Cost Function - varying $n - k$

We now consider the effects of adding parity packets to our system. Intuitively, adding more parity packets would lead to better reliability. Here we vary $n - k$, the

number of parity packets, as well as Z . We are actually interested in the effect of adding ‘parity checksum groups’. If we fix Z , that means that the new parity packets are squeezed into the existing checksum groups, and this does not provide better reliability as might be expected.

Figure 6.3 shows the results, with $k = 10$, $p = 0.01$, $Z = 2$, $w_1 = 0.9$, and $w_2 = 0.1$. The two upper graphs show that both reliability metrics drop as the number of parity packets increase. This make intuitive sense because adding more parity packets can provide better reliability. Checksum per data packet increase linearly with number of parity packets per FEC group, because Z increase linearly as we add more parity packets, and k is fixed. In the cost function graphs, when $n - k \leq 5$, the cost decreases because the reliability metrics decrease. At $n - k = 5$ in the middle right graph and $n - k = 10$ in the bottom graph, the cost increases because $\frac{Z}{k}$ increases while both reliability metrics approaching 0.

6.4 Varying the Number of FEC Groups in a File

We study how we should choose k , the number of data packets in each FEC group, in this section. We are interested in the following question. Should we group the data packets into few large FEC groups, or should we group them into many smaller FEC groups?

The results are plotted in Figure 6.4, with $W = 100$, $n = 2k$, $Z = 2$, $p = 0.01$, $w_1 = 0.9$, and $w_2 = 0.1$. The two upper graphs show that reliability drops as k increases. That is, large FEC groups do not tolerate faults as well as small FEC groups. However, using smaller FEC groups means that we are sending more checksums, as we need at least two checksums for each FEC group. Since Z is fixed while k increases, checksum per data packets, given by $\frac{Z}{k}$, decreases exponentially.

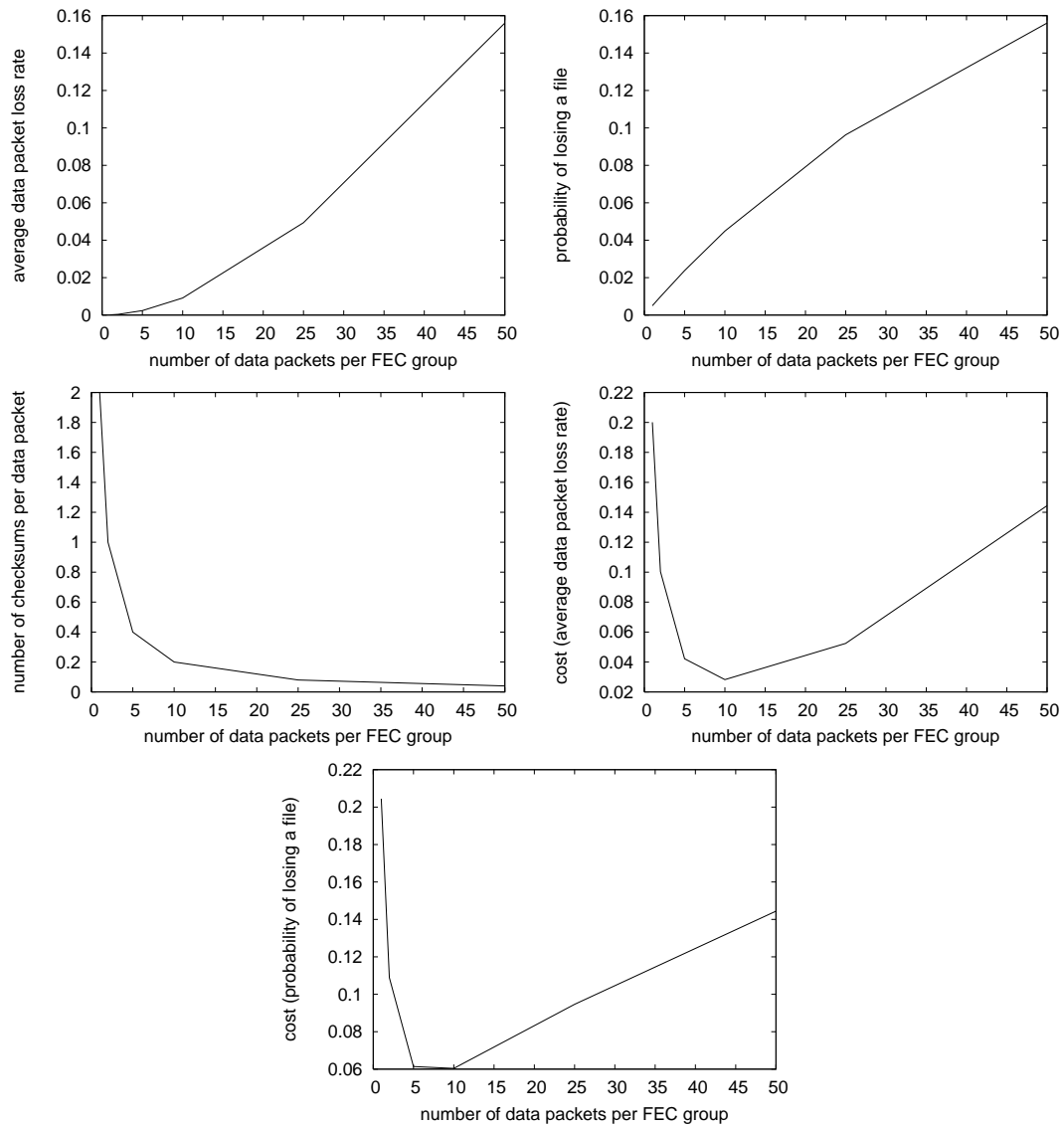


Figure 6.4: Cost Function - varying k

In the cost function graphs, cost is high when k is small, because this results in a lot of checksums. The cost drops when k is between 1 and 10, as we send fewer checksums and the corresponding reliability penalty does not increase as fast. Eventually, when $k \geq 10$, since larger FEC groups are not as fault tolerant, cost goes up as k increases.

6.5 Varying the Probability of Losing a Packet

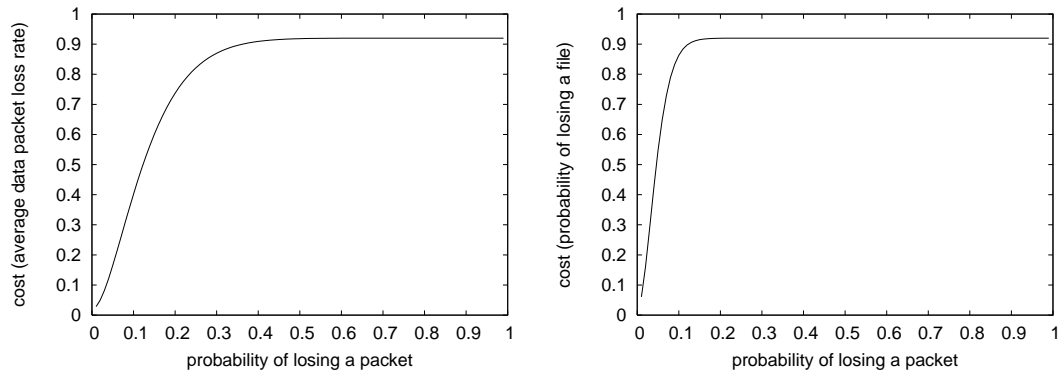


Figure 6.5: Cost Function - varying p

We are interested in looking at how the cost function changes with the probability of losing a packet, p . Figure 6.5 depicts our results, with $Y = 5$, $n = 20$, $k = 10$, $Z = 2$, $w_1 = 0.9$, and $w_2 = 0.1$.

Since both Z and k are fixed, changes in cost reflects changes in the reliability metrics. As p increases from 0.1 to 0.6, cost increases rapidly. As $p > 0.6$, the cost increases at a decreasing rate. This is because both the probability of losing part of a file and the average data packet loss rate approaches 1 as p increases. This result makes intuitive sense as p increases, reliability metric should increase.

Chapter 7

Future Work

In this chapter, we outline possible directions for future work.

In the data transfer step, a client needs to choose B bistros for striping their data. Clients also need to choose how much data to send to each of the B bistros according to their reliability and performance characteristics. For example, clients may want to send their files to faster bistros to minimize their response time, and they may want to send files to more reliable bistros to minimize the chance of losing part of their files. We believe that this problem can be formulated as a variant of the transportation problem, which is a NP-complete problem studied extensively in the area of operations research.

After the destination server has collected k packets from each FEC group, it has a choice of reconstructing the data without retrieving the remaining $n - k$ packets. This approach can reduce network bandwidth requirement, but it can increase the computational cost at the destination server for decoding. The remaining packets, on the other hand, could be on malicious bistros and hence may be unavailable. Since some erasure code decoders do not benefit from more than k packets, attempting to retrieve the remaining packets could be a waste if they are unlikely to pass the checksum check.

In this work we investigated data loss and data corruption. However, recall that we rely on bistros to send session keys to the destination server. Although the session

keys are encrypted with the public key of the destination server, malicious bistros can corrupt session keys to prevent data from properly reaching the destination server. Even though the destination server can treat such data as corrupted since it will not pass the checksum check, we can look for ways to prevent this from happening, hence making the system even more fault tolerant. One way to achieve this goal is to have the clients also send the session keys to the destination server in the timestamp step. This would make the timestamp message even larger, especially when a client stripes files across a large number of bistros.

Blacklisting of malicious bistros can provide better fault tolerance as we can eliminate unreliable bistros from future upload events. If data from a particular bistro does not pass the checksum checks frequently, we can assume that bistro is malicious, or has some software or hardware problems (i.e., unreliable). We can blacklist that bistro and notify its administrators, and remove that bistro from future upload events.

Another fault tolerance issue in Bistro is the failure of a destination server. If the destination server fails, an event owner can receive no data since all information about clients and files is unavailable. We need to figure out what is needed to reconstruct data on the destination server, then we can derive an algorithm to reconstruct the database of the failed destination server from data reside intermediate bistros.

Chapter 8

Conclusions

Hot spots are a major hurdle to making Internet applications scalable. Many researches have been addressing this problem in one-to-one applications, one-to-many applications, and many-to-many applications. Yet, to the best of our knowledge, there is no work on relieving hot spots for many-to-one, or upload, applications except for Bistro, which have been shown to have a scalable and secure design.

The goal of this thesis is to develop a fault tolerance protocol that improves performance in the face of failures or malicious behavior of intermediaries. In the original protocol, if any bistro is not available during data collection step, files on that bistro are lost, and the destination server has to ask for resubmissions. Also, if bistros are malicious, i.e., they intentionally corrupt data, the destination server can detect this, but it cannot recover the corrupted files, hence we have to request them from the clients again. Our goal is to provide redundancy using erasure codes to tolerate failures or corruption of some intermediate bistros, while minimizing the amount of storage and network transfer costs as a result of employing the protocol.

We developed a fault tolerance protocol in this thesis. As in the original protocol, the proposed fault tolerance mechanism is scalable and can provide data security. We encode files with erasure codes, divide them into checksum groups, and generate

a checksum for each checksum group. We concatenate the checksums and send them to the destination server in the timestamp step. We stripe the data across a number of bistros in the data transfer step. Finally, the destination server collects data residing on intermediaries in the data collection step.

We evaluated our protocol using proposed analytical models. We provided a reliability model that for computing the probability of losing part of a file as well as the average data packet loss rate. Our performance model uses number of checksums per data packet as its metric. Furthermore, we use a cost function to combine the reliability and performance metrics in order to study the combined effects and the resulting tradeoff.

We studied the resulting cost, as a function of a number of parameters, including number of data packets per FEC group, the number of parity packets, and the number of checksum groups per FEC group. We also studied the sensitivity of the cost function to the probability of losing a packet and to the weights of the cost function.

Conclusively, we believe fault tolerance is important in wide area data transfer applications. We developed a fault tolerance protocol in the Bistro architecture. We believe our protocol can provide fault tolerance while minimizing additional cost as a result of employing our protocol. Better fault tolerance also leads to fewer retransmissions due to packet losses or corruptions, resulting in better system performance.

Bibliography

- [1] Akamai, <http://www.akamai.com>.
- [2] Jong-Suk Ahn and John Heidemann. An adaptive FEC algorithm for mobile wireless networks. Technical Report ISI-TR-555, USC/Information Sciences Institute, March 2002.
- [3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Symposium on Operating Systems Principles (SOSP) 15*, December 1995.
- [4] Samrat Bhattacharjee, William C. Cheng, Cheng-Fu Chou, Leana Golubchik, and Samir Khuller. Bistro: a framework for building scalable wide-area upload applications. *ACM SIGMETRICS Performance Evaluation Review*, 28(2):29–35, September 2000.
- [5] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, International Computer Science Institute, Berkeley, California, 1995.
- [6] John Byes, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM*, September 1998.
- [7] William C. Cheng, Cheng-Fu Chou, and Leana Golubchik. Performance of batch-based digital signatures. In *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 291–302, October 2002.
- [8] William C. Cheng, Cheng-Fu Chou, Leana Golubchik, and Samir Khuller. A secure and scalable wide-area upload service. In *Proceedings of 2nd International Conference on Internet Computing*, volume 2, pages 733–739, June 2001.
- [9] William C. Cheng, Cheng-Fu Chou, Leana Golubchik, and Samir Khuller. A performance study of bistro, a scalable upload architecture. *ACM SIGMETRICS Performance Evaluation Review*, 29(4):31–39, 2002.

- [10] G. Ding, H. Ghafoor, and B. Bhargava. Resilient video transmission over wireless networks. In *6th IEEE International Conf. on Object-oriented Real-time Distributed Computing*, May 2003.
- [11] Nick Feamster and Harl Balakrishnan. Packet loss recovery for streaming video. In *IEEE Packet Video Workshop 2002*, April 2002.
- [12] Gui-Liang Feng, Yan Yang, Robert Deng, and Feng Bao. A novel reed-solomon-like code scheme with only XOR operations. Technical report, Center of Advanced Computer Studies, University of Louisiana Lafayette, 2003.
- [13] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobiuff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284, June 1997.
- [14] Leana Golubchik, John C.S.Lui, Tak Fu Tung, Lik Hang Chow, W.J. Lee, G. Franceschinis, and C. Anglano. Multi-path continuous media streaming: What are the benefits? *Performance Evaluation Journal*, 39:429–449, September 2002.
- [15] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3), August 1995.
- [16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [17] J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *13th Symposium on Operating Systems Principles (SOSP '91)*, pages 213–225.
- [18] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gum-madi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS 9*, November 2000.
- [19] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. RFC3453 : The use of forward error correction (FEC) in reliable multicast, December 2002.
- [20] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, February 2001.
- [21] Philip McKinley and Arun Mani. An experimental study of adaptive forward error correction for wireless collaborative computing. In *IEEE Symposium on Applications and the Internet (SAINT 2001)*, January 2001.

- [22] P. V. Mockapetris. RFC 1034: Domain names — concepts and facilities, November 1987.
- [23] P. V. Mockapetris. RFC 1035: Domain names — implementation and specification, November 1987.
- [24] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *IEEE Packet Video Workshop 2002*, April 2002.
- [25] Jorg Nonnenmacher and Ernst Biersack. Reliable multicast: where to use FEC. In *Protocols for High-Speed Networks*, pages 134–148, 1996.
- [26] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116. ACM Press, 1988.
- [27] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.
- [28] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, March 2003.
- [29] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, April 1997.
- [30] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent, peer-to-peer storage utility. In *SOSP 18*, October 2001.
- [31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Summer 1985 USENIX Conference*, pages 119–130, June 1985.