

# A Fault Tolerance Protocol for Uploads: Design and Evaluation\*

L. Cheung<sup>1</sup>, C.-F. Chou<sup>2</sup>, L. Golubchik<sup>3</sup>, and Y. Yang<sup>1</sup>

<sup>1</sup> Computer Science Department, University of Southern California, Los Angeles, CA  
{lccheung, yangyan}@usc.edu

<sup>2</sup> Department of Computer Science and Information Engineering,  
National Taiwan University  
ccf@csie.ntu.edu.tw

<sup>3</sup> Computer Science Department, EE-Systems Department, IMSC, and ISI,  
University of Southern California, Los Angeles, CA  
leana@cs.usc.edu

**Abstract.** This paper investigates fault tolerance issues in Bistro, a wide area upload architecture. In Bistro, 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. This degrades system performance since the destination server may need to ask for retransmissions. As a result, a fault tolerance protocol is needed within the Bistro architecture. Thus, in this paper, we develop such a protocol which employs erasure codes in order to improve the reliability of the data uploading process. 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. We also present numerical results to illustrate this tradeoff.

## 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. There are also real life deadlines associated with some events, such as submissions of papers to conferences. The demand of applications with deadlines is potentially higher when the deadlines are approaching.

---

\* This work is supported in part by the NSF Digital Government Grant 0091474. It has also been funded in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation. More information about the Bistro project can be found at <http://bourbon.usc.edu/iml/bistro>.

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 for Bistro [1]. Bistro is a wide-area upload architecture built at the application layer, and previous work [2] has shown that it is scalable and secure.

In Bistro, an upload process is broken down into three steps (see Sect. 3 for details) [1]. First, in the timestamp step, clients send hashes of their files,  $h(T)$ , to the server, and obtain timestamps,  $\sigma$ . These timestamps clock clients' submission time. In the data transfer step, clients send their data,  $T$ , to intermediaries called bistros. 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 in the original Bistro architecture [1, 2].

We are interested in developing and analyzing a fault tolerance protocol in this paper, in the context of the Bistro architecture. The original Bistro does not make any additional provisions for cases when bistros are not available during the data collection step. In addition, malicious bistros can intentionally corrupt data. Although a destination server can detect corrupted data from the hash check, it has no way of recovering the data. Hence, unavailable bistros and malicious behavior can result in the destination server having to request client resubmissions. In this work, we are interested in using forward error correction techniques to recover corrupted or lost data in order to improve the overall system performance. The fault tolerance protocol, on the other hand, brings in additional storage and network transfer costs due to redundant data. The goal of this paper is to (a) provide better performance when intermediaries fail while reducing the amount of redundant data needed to accomplish this, and (b) to evaluate the resulting tradeoff between performance and reliability.

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. Moreover, we study the tradeoff between reliability and performance.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes our fault tolerance protocol. We derive analytical models for this protocol in Sect. 4. Section 5 presents numerical results showing the tradeoff between performance and reliability characteristics of our protocol. Finally, we conclude in Sect. 6.

## 2 Related Work

This section briefly describes fault tolerance considerations in other large-scale data transfer applications, and discusses other uses of erasure codes in the context of computer networking.

One approach to achieving fault tolerance is through service replication. Replication of DNS servers 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 servers, 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, data redundancy techniques, such as RAID techniques [3], 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 redundancy can provide better fault tolerance characteristics, the storage overhead can be high. We are interested in providing fault tolerance with small storage overhead in this work.

Erasure codes are useful in bulk data distribution, e.g., in [4] 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 approaches.

In wireless networking, using forward error correction techniques can reduce packet loss rates by recovering parts of lost packets [5, 6]. 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.

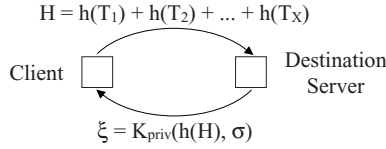
These applications of erasure codes assume that packets are either received successfully or are lost. They assume that there are other ways to detect corrupted packets. e.g., using TCP checksums. In Bistro, however, this assumption is not valid because packets can be intentionally corrupted by intermediate bistros. In Sect. 3, we describe one way to detect corrupted packets using checksums so that we can treat corrupted packets as losses.

### 3 Fault Tolerance Protocol

This section provides details of our fault tolerance protocol. The protocol is broken down into three parts as in the original Bistro protocol described in [2]. We provide details of each step in this section with focus on the fault tolerance aspects proposed in this paper. We also discuss related design decisions.

#### 3.1 Timestamp Step

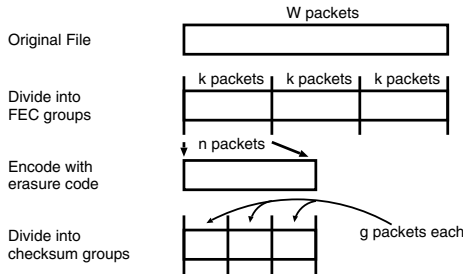
The timestamp step verifies clients' submissions. Clients first pass their files,  $T_o$ , to erasure code encoders to get the encoded files  $T = T_1 + T_2 + \dots + T_x$ . Then, clients generate hashes of each part of their data, concatenate the hashes and send the results,  $H$ , to the destination server. The destination server replies to clients with tickets,  $\xi$ , which consist of timestamps,  $\sigma$ , and the hash messages clients have just sent,  $h(H)$ . Tickets are digitally signed by the destination server, so clients can authenticate the destination server. Fig. 1 depicts the timestamp step.



**Fig. 1.** Timestamp Step

In the original protocol, clients send a checksum (or a hash) 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 of entire files, in the original protocol.

To solve this problem, we send multiple checksums in the fault tolerance protocol,  $h(T_1) + h(T_2) + \dots + h(T_x)$ . 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 simplifies our reliability evaluation in Sect. 4. There are altogether  $X = YZ$  checksums, which are concatenated and sent in one message to the destination server. Figure 2 illustrates the relationship between FEC groups and checksum groups.



**Fig. 2.** FEC Groups and Checksum Groups

Note that the size of a checksum group has to be smaller than the 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 would not be able to recover because less than  $k$  packets within that FEC group are received, i.e., we would 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 where 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 also explains the order in which FEC groups and checksum groups are constructed.

### 3.2 Data Transfer Step

In the data transfer step, clients send their files to intermediate bistros which are not trusted. Clients first choose  $B$  bistros to send their data to and then generate a session key  $K_{ses_i}$  for each chosen bistro,  $1 \leq i \leq B$ . After that, clients divide their files into  $B$  parts. For each part  $i$ , clients encrypt it with a session key  $K_{ses_i}$  and send that part to an intermediate bistro  $i$ . Clients also send to bistro  $i$  the session key,  $K_{ses_i}$ , and ticket,  $\xi$ , encrypted with the public key of the destination server. In addition, clients send event IDs,  $EID$ , so as to identify that the data is for a particular upload event whose event ID is  $EID$ . Each bistro  $i$  generates a receipt,  $\rho_i$ , and sends it to both an appropriate client and the destination server. The receipts contain the public key of bistro  $i$ ,  $K_{i,pub}$ , so that both clients and the destination server can decrypt and verify the receipt<sup>1</sup>. Figure 3 depicts the data transfer step.

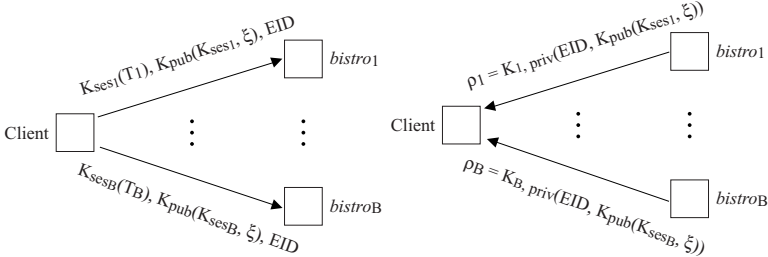


Fig. 3. Data Transfer Step

In [7], the so-called assignment problem is studied, i.e., how a client should choose a bistro to which it sends its file. However, in that case, only one bistro out of a pool of bistros is chosen. In the case of striping (our case), a client needs to choose  $B \geq 1$  bistros. As shown in [7], this is a difficult problem even for  $B = 1$ . Hence, we leave the choice of which  $B$  bistros a client should stripe its file to, and how clients determine the value of  $B$  to future work. In the remainder of this paper, we assume that the  $B$  bistros are known.

<sup>1</sup> Note that whether the public key of an intermediate bistro is correct or not does not affect the correctness of the protocol, as in the original Bistro system, as intermediate bistros are not trusted in any case.

### 3.3 Data Collection Step

In the data collection step, the destination server coordinates intermediate bistros to collect data. When the destination server wants to retrieve data from bistro  $i$ , it sends a retrieval request along with the receipt  $\rho_i$  and the event ID  $EID$ . Upon receiving retrieval requests from the destination server, bistro  $i$  sends the file  $T_i$  along with the encrypted session key and ticket for decryption. Figure 4 depicts the data collection step.

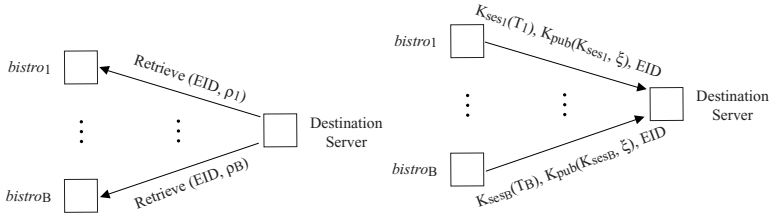


Fig. 4. Data Collection Step

When all packets within a checksum group are received, the destination server computes the 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.

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. The erasure code decoder then reconstructs the original file  $T_o$ . If the destination server receives less than  $k$  packets from any FEC group, it contacts the appropriate clients and requests retransmissions of specific FEC group(s).

## 4 Analytical Models

We propose analytical models to evaluate our fault tolerance protocol in this section. We develop a reliability model to study how reliability characteristics of bistros affect system reliability. We also develop 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.

### 4.1 Reliability Model

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

Let us assume that losing or corrupting one packet is independent of losing or corrupting other packets within the same checksum group. Let  $p$  be the probability that a packet is lost or corrupted. Then,

$$p_g = (1 - p)^g. \quad (1)$$

Due to the lack of space, we omit the derivation of  $P_{retrans}$ , the probability that retransmission is needed, and simply state the result as follows:

$$P_{retrans} = 1 - \left( \sum_{i=\lceil \frac{k}{g} \rceil}^Z \binom{Z}{i} p_g^i (1 - p_g)^{Z-i} \right)^Y. \quad (2)$$

The derivation of (2) can be found in [8]. We have also developed other reliability models to evaluate the reliability of our protocol, which are omitted here due to lack of space. They can be found in [8].

## 4.2 Performance Model

This section describes the performance model used for evaluating our fault tolerance protocol. We limit the evaluation in this paper to the performance penalty in the timestamp step only. This is motivated by the fact that if the performance of timestamp step is poor, then we are back to the original problems where many clients are trying to send large amounts of data to a server at the same time.

We believe that it is more important to consider the potential overloading of network resources, due to sending a greater number of checksums, than the additional computational needs on the server for producing digital signatures of larger timestamp messages. This is again due to the consideration that clients sending large messages to the destination server around the deadline time would take us 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.

Hence, we use the number of checksum groups per data packet,  $\frac{Z}{k}$ , as our performance metric. This is derived by considering the total number of checksums,  $YZ$ , normalized by the file size,  $Yk$ .

## 4.3 Cost Function

Now that we have a reliability model and a performance model, the question is how to combine the effects of 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

$$C = w_1 C_1 + w_2 C_2 \quad (3)$$

where  $w_1$  and  $w_2$  are weights of each factor.

Earlier we derived the probability that retransmission is needed,  $P_{retrans}$ , as a reliability metric. We can use this metric as our reliability cost. The performance cost in the timestamp step is given by the number of checksum groups per data packet,  $\frac{Z}{k}$ . In the next section, we study how reliability and performance metrics affect the overall cost function.

## 5 Numerical Results

This section provides numeric results on varying different parameters of our protocol and their effect on the cost function discussed above. Due to lack of space, we present only a subset of our experiments. Other results, which also support our findings, can be found in [8].

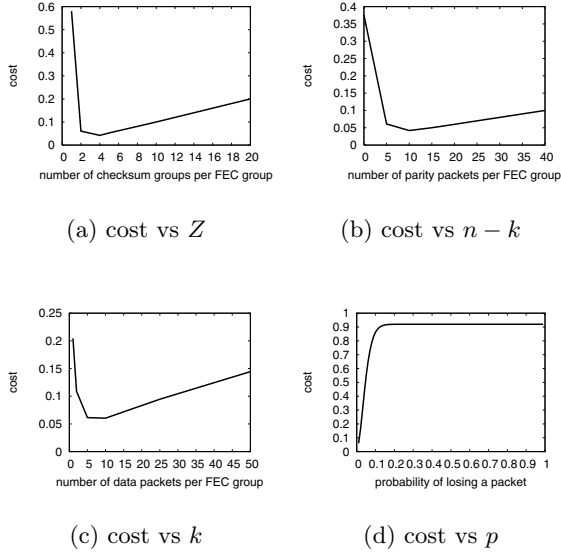
The parameters of interest to our system and its corresponding reliability and performance characteristics include the following.

1. Number of checksum groups per FEC group,  $Z$ . Setting  $Z$  to be large can provide better reliability because loss of a packet affects fewer other packets, as we drop the entire checksum group whenever any packet 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 the differences in dividing the file into few large FEC groups or many small FEC groups.
4. Probability of losing a packet,  $p$ . We want to see how sensitive the cost function is to  $p$ .
5. Weights  $w_1$  and  $w_2$ . We are interested in how sensitive the cost function is to the chosen weight values. We performed a number of experiments with different weight values (please refer to [8] for the results). Due to lack of space, we omit these results here and only use representative weight values in the remainder of the paper.

The tradeoff between reliability and performance in the context of varying the number of checksum groups per FEC group,  $Z$ , is illustrated in Fig. 5(a), where  $Y = 5$ ,  $n = 20$ ,  $k = 10$ ,  $p = 0.01$ ,  $w_1 = 0.9$ , and  $w_2 = 0.1$ . In Fig. 5(a) the cost is high when  $Z$  is small because  $P_{retrans}$  is high. The cost decreases when  $Z$  is between 1 and 4 because  $P_{retrans}$  is improving. At  $Z \geq 2$ , the cost goes up again because the size of the message becomes too large.

We study the tradeoff between reliability and performance of adding parity packets,  $n - k$  in Fig. 5(b), where  $k = 10$ ,  $p = 0.01$ ,  $Z = 2$ ,  $w_1 = 0.9$ , and  $w_2 = 0.1$ . In Fig. 5(b), when  $n - k \leq 10$ , the cost decreases because  $P_{retrans}$  decreases. At  $n - k \geq 10$  in Fig. 5(b), the cost increases because  $\frac{Z}{k}$  increases while  $P_{retrans}$  approaches 0.

We study how we should choose  $k$ , the number of data packets in each FEC group, in Fig. 5(c). For this experiment we set  $W = 100$ ,  $n = 2k$ ,  $Z = 2$ ,  $p = 0.01$ ,



**Fig. 5.** Results for varying different parameters in the cost function

$w_1 = 0.9$ , and  $w_2 = 0.1$ . Figure 5(c) shows that 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  $k \geq 10$ , cost goes up as  $k$  increases since larger FEC groups are not as fault tolerant.

We are interested in looking at how the cost function changes with the probability of losing a packet,  $p$ . We set  $Y = 5$ ,  $n = 20$ ,  $k = 10$ ,  $Z = 2$ ,  $w_1 = 0.9$ , and  $w_2 = 0.1$  in Fig. 5(d). Since both  $Z$  and  $k$  are fixed, changes in cost reflect changes in  $P_{retrans}$ . Cost increases rapidly when  $p$  is between 0 and 0.1. When  $p > 0.1$ , since  $P_{retrans}$  approaches 1, cost remains fairly constant.

## 6 Conclusions

Bistro is a scalable and secure wide-area upload architecture that can provide an efficient upload service. The goal of this paper was to develop a fault tolerance protocol that improves performance in the face of failures or malicious behavior of intermediaries in the context of the Bistro architecture. We developed such a protocol using a forward error correction technique. We also evaluated this protocol using proposed analytical models to study the reliability and performance characteristics. We studied the resulting cost, as a function of a number of parameters, including the number of data packets per FEC group, the number of parity packets, and the number of checksum groups per FEC group. In conclusion, we believe that fault tolerance is important in wide area data up-

load applications. We believe that the proposed protocol is a step in the right direction, leading to better fault tolerance characteristics with fewer retransmissions due to packet losses or corruptions, resulting in better overall system performance.

## References

1. Bhattacharjee, S., Cheng, W.C., Chou, C.F., Golubchik, L., Khuller, S.: Bistro: a framework for building scalable wide-area upload applications. *ACM SIGMETRICS Performance Evaluation Review* **28** (2000) 29–35
2. Cheng, W.C., Chou, C.F., Golubchik, L., Khuller, S.: A secure and scalable wide-area upload service. In: *Proceedings of 2nd International Conference on Internet Computing*, Volume 2. (2001) 733–739
3. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). In: *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, ACM Press (1988) 109–116
4. Byes, J., Luby, M., Mitzenmacher, M., Rege, A.: A digital fountain approach to reliable distribution of bulk data. In: *ACM SIGCOMM*. (1998)
5. Ding, G., Ghafoor, H., Bhargava, B.: Resilient video transmission over wireless networks. In: *6th IEEE International Conf. on Object-oriented Real-time Distributed Computing*. (2003)
6. McKinley, P., Mani, A.: An experimental study of adaptive forward error correction for wireless collaborative computing. In: *IEEE Symposium on Applications and the Internet (SAINT 2001)*. (2001)
7. Cheng, W.C., Chou, C.F., Golubchik, L., Khuller, S.: A performance study of bistro, a scalable upload architecture. *ACM SIGMETRICS Performance Evaluation Review* **29** (2002) 31–39
8. Cheung, L., Chou, C.F., Golubchik, L., Yang, Y.: A fault tolerance for uploads: Design and evaluation. Technical Report 04-834, Computer Science Department, University of Southern California (2004)