Multi-Version Coding - An Information Theoretic Perspective of Consistent Distributed Storage

Zhiying Wang, Viveck R. Cadambe*

Abstract

In applications of distributed storage systems to distributed computing and implementation of key-value stores, the following property, usually referred to as consistency in computer science and engineering, is an important requirement: as the data stored changes, the latest version of the data must be accessible to a client that connects to the storage system. An information theoretic formulation called multi-version coding is introduced in the paper, in order to study storage costs of consistent distributed storage systems. Multi-version coding is characterized by ν totally ordered versions of a message, and a storage system with n servers. At each server, values corresponding to an arbitrary subset of the ν versions are received and encoded. For any subset of c servers in the storage system, the value corresponding to the latest common version, or a later version as per the total ordering, among the c servers is required to be decodable. An achievable multi-version code construction via linear coding and a converse result that shows that the construction is approximately tight, are provided. An implication of the converse is that there is an inevitable price, in terms of storage cost, to ensure consistency in distributed storage systems.

I. Introduction

There is an enormous interest in recent times to understand the role of erasure coding in distributed storage systems. In this paper, we formulate a new information theoretic problem, the *multi-version coding* problem, motivated by applications of distributed storage systems to distributed computing and implementation of key-value stores. The multi-version coding problem captures two aspects that are not considered previously in information theoretic studies of distributed storage systems:

i) In several applications, the message (data) changes, and the user wants to get the latest version of the message. In computer science literature [1], the notion of obtaining the latest version of the data is known as *consistency*[1]

ii) There is an inherent *asynchrony* in storage systems due to the distributed nature of the system. As a consequence, the new version of the message may not arrive at all servers in the system at the same time.

The design of a consistent data storage service over an asynchronous distributed storage system has been studied carefully in distributed computing theory literature [1], [4], and forms an integral part of several data storage products used in practice, such as Amazon Dynamo [5], Apache Cassandra [6], and CouchDB [7]. The main objective of the multi-version coding problem is to understand the

*Zhiying Wang is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, and her email is zhiying@uci.edu. Viveck R. Cadambe is with Department of Electrical Engineering, Pennsylvania State University, and his email is viveck@engr.psu.edu.

This work is published in part, in the Proceedings of the 2014 IEEE International Symposium on Information Theory (ISIT), June 2014 and in the Proceedings of the 2014 IEEE Annual Allerton Conference on Communications, Control and Signal Processing, Oct 2014.

[1]There are several formal models of consistency studied in distributed systems literature (See for example [1], [2], [3]). In this paper, we use the term consistency to loosely mean that a user wants the most recent version of the data.

storage costs of consistent distributed storage systems from an information theoretic perspective. We begin with an informal description of the problem. We discuss the background and motivation of our problem formulation in Section I-B.

A. Informal Problem Description

Our problem formulation is pictorially depicted in Fig. 1. Consider a distributed storage system with a set of n servers. Suppose that it stores message W_1 using an n length code, such that a decoder can connect to *any* subset of c servers and decode W_1. Suppose an updated version of the message W_2 enters the system. For reasons that may be related to network delays or failures, W_2 arrives at some subset of servers, but not others. We assume that each server is unaware of which servers have W_2 and which do not. The question of interest here is to design a storage strategy for the servers so that, a decoder can connect to any c servers and decode the *latest common version* among the c servers, or *some version later* than the latest common version. That is, W_2 must be decodable from every set of c servers where each server in the set has received both W_1 and W_2. For every set of c servers where there is at least one server which has not received W_2, we require that either W_1 or W_2 is decodable. We intend our storage strategy to be applicable to every possible message arrival scenario, and every possible subset of servers of size c. A possible scenario is depicted in Fig. 1 for $n = 3, c = 2$.

Notice that in the storage strategy, a server with both W_1, W_2 stores a function of W_1 and W_2, whereas a server with only W_1 stores a function of W_1. We now describe two simple approaches, *replication* and *simple erasure coding*, that solve this problem. We assume that the *size* of both versions are equal, that is, the number of bits used to represent W_1 is equal to W_2. We refer to the size of one version as one unit.

- *Replication:* In this strategy, we assume that each server stores the latest version it receives, that is, servers with both versions store W_2, and servers with the first version store W_1. Notice that the storage cost of this strategy is 1 unit per server, or a total of n units. See Table I for an example.
- *Simple Erasure Coding:* In this strategy, we use two (n, c) MDS (maximum distance separable) codes, one for each version separately. A server stores one codeword symbol corresponding to every version it receives. So, a server with both versions stores two codeword symbols resulting in a storage cost of $\frac{2}{c}$ units, whereas, a server with only the first version stores $\frac{1}{c}$ unit. Notice that for the worst case where all servers have both versions, the total storage cost per server is $\frac{2}{c}$ unit. See Table II for an example.

We use *worst-case* storage costs to measure the performance of our codes for simplicity. Therefore, the per server storage cost of replication is equal to 1 unit, and that of the simple erasure coding strategy is equal to $\frac{2}{c}$ units. The singleton bound provides a natural information theoretic lower bound on the storage cost. In particular, the singleton bound implies that each node has to store at least $\frac{1}{c}$ units, even for storing a single version. A natural question of interest is whether we can achieve a storage cost of $\frac{1}{c}$ or whether a new information theoretic lower bound can be found. It is useful to note that asynchrony makes the problem non-trivial. In a synchronous setting, where all the servers receive all the versions at the same time, an MDS code-based strategy where each server stores a codeword symbol corresponding to the latest version received suffices. So, in a synchronous setting, the singleton bound would be tight.

Our main achievability result provides a code construction, which shows that replication and simple MDS codes are both sub-optimal. It is worth noting that we do not make any assumptions on the correlation between the two versions. Even with our conservative modeling assumption which ignores possible correlation among the versions, we can construct achievable coding schemes that, albeit mildly, improve upon simple erasure coding and replication.

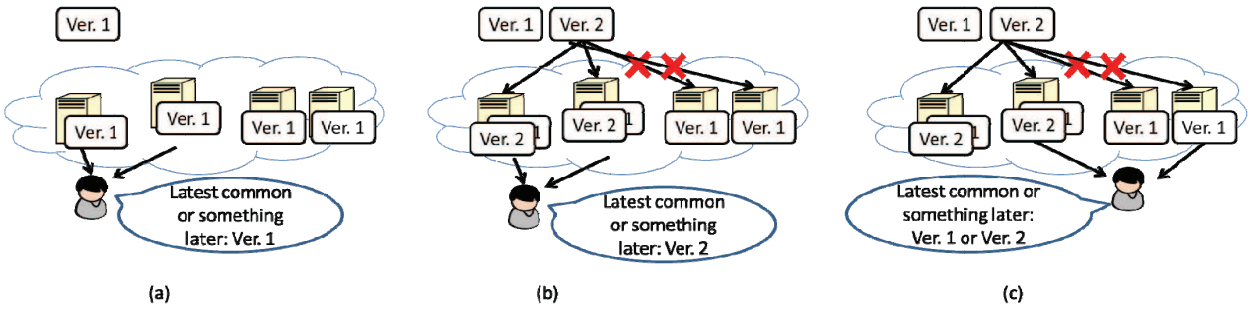

Fig. 1. Storing a file with 2 version in $n = 4$ nodes. From any $c = 2$ nodes, the code should recover the latest common version or something later. We denote the old and new versions as Version 1 and Version 2 respectively.

		Server 1	Server 2	Server 3
Initially Ver. 1 available at all servers	Ver. 1	W_1	W_1	W_1
Then, Ver. 2 reaches Servers 1 and 2	Ver. 1 Ver. 2	 W_2	 W_2	W_1

TABLE I

Replication for $n = 3, c = 2$ with two versions. A server stores W_1 if it receives only Version 1; it stores W_2 if it receives both versions. Two possible scenarios are shown in the table. Note that the latest version is decodable from every 2 servers in both scenarios. In general, it can be verified that the latest common version or a later version is decodable from every $c = 2$ servers, in every possible scenario. The storage cost is 1 unit.

Our main converse result shows that in the asynchronous setting that we study, the singleton bound is not tight and that the storage cost for any multi-version code cannot be close to $\frac{1}{c}$. Our converse implies that there is an inherent, unavoidable cost of ensuring a consistent storage service because of the asynchrony in the system.

For the setting described where there are two versions, we provide in this paper a code construction that achieves a per server storage cost of $\frac{2}{c+1}$ for odd c. When c is even, we achieve a storage cost of $\frac{2(c+1)}{c(c+2)}$. Table III provides an example of our construction with storage cost of $3/4$ unit, for $n = 3, c = 2$. Note that our construction outperforms replication and simple MDS codes. We provide in this paper a converse that shows that the worst case storage cost cannot be smaller than $\frac{2}{c+1}$, under some mild assumptions. The converse implies that our code construction is essentially optimal for odd values of c.

In this paper, we study a generalization of the above problem. In a system with n servers and ν versions, a *multi-version code* allows every server to receive any subset of the ν versions. Every server encodes according to the versions that it received. The decoder takes as input, codeword symbols of an arbitrary set of c servers, $c \leq n$, and recovers the latest common version among these servers, or some version later. The storage cost is the worst-case storage size per server over all possible scenarios, that is, over all possible subsets of versions corresponding to the servers. In this paper, we provide an information-theoretic characterization of the storage cost of such codes, including code constructions and lower bounds for given parameters n, c, ν.

B. Background and Motivation

The multi-version coding problem is characterized by two new aspects in its formulation: (i) the idea of consistency in the decoder, and (ii) the asynchrony in the distributed storage system. We describe some motivating applications and related background literature that inspire our formulation here.

Storing multiple versions of the same message consistently is important in several applications. For instance, the idea of requiring the latest version of the object is important in *shared memory*

		Server 1	Server 2	Server 3
Initially Ver. 1 available at all servers	Ver. 1	p_1, p_2	p_3, p_4	$p_1 \oplus p_3, p_2 \oplus p_4$
Then, Ver. 2 reaches Servers 1 and 2	Ver. 1 Ver. 2	p_1, p_2 q_1, q_2	p_3, p_4 q_3, q_4	$p_1 \oplus p_3, p_2 \oplus p_4$

TABLE II

Simple erasure coding for $n = 3, c = 2$ with two versions. Assume each unit is 4 bits, and the bits of the two versions are $W_1 = (p_1, p_2, p_3, p_4)$, $W_2 = (q_1, q_2, q_3, q_4)$. Every version is coded with a $(3, 2)$ MDS code where each codeword symbol is a 2-bit vector. The 3 codeword symbols for the 3 servers are $(p_1, p_2), (p_3, p_4), (p_1 \oplus p_3, p_2 \oplus p_4)$ for Version 1, and $(q_1, q_2), (q_3, q_4), (q_1 \oplus q_3, q_2 \oplus q_4)$ for Version 2. A server stores its corresponding codeword symbol of Version 1 if it has only Version 1; it stores codeword symbols of both Version 1 and Version 2 if it receives both versions. Two possible scenarios are shown in the table. It can be verified that the latest common version is decodable from every $c = 2$ servers, in every possible scenario. The storage cost is 4 bits, or equivalently, 1 unit.

		Server 1	Server 2	Server 3
Initially Ver. 1 available all servers	Ver. 1	p_1, p_2, p_3	p_1, p_2, p_4	p_1, p_2, p_5
Then, Ver. 2 reaches Servers 1 and 2	Ver. 1 Ver. 2	p_3 q_1, q_2	p_4 q_3, q_4	p_1, p_2, p_5

TABLE III

Proposed code for $n = 3, c = 2$ and two versions. Assume each unit is 4 bits, and the two versions are $W_1 = (p_1, p_2, p_3, p_4)$, $W_2 = (q_1, q_2, q_3, q_4)$. Here $p_5 = p_1 \oplus p_2 \oplus p_3 \oplus p_4$. Server 1 stores (p_1, p_2, p_3) if it receives only Version 1; it stores (p_3, q_1, q_2) if it receives both versions. Server 2 stores (p_1, p_2, p_4) if it receives only Version 1; it stores (p_4, q_3, q_4) if it receives both versions. Server 3 stores (p_1, p_2, p_5) if it receives only Version 1; it stores $(p_5, q_1 \oplus q_3, q_2 \oplus q_4)$ if it receives both versions. Two possible scenarios are shown in the table. It can be verified that the latest common version or a later version is decodable from every 2 servers, in every possible scenario. The storage cost is 3 bits, or equivalently, $3/4$ unit.

systems [1] that form the cornerstone of theory and practice of multiprocessor programming [8]. In particular, when multiple threads access the same variable, it is important that the changes made by one thread to the variable are reflected when another thread reads this variable. Another natural example comes from key value stores, for instance, applied to storing data in a stock market, where acquiring the latest stock value is of significant importance.

Asynchrony is inherent to the distributed nature of the storage systems used in practice. In particular, asynchrony occurs due to temporary or permanent failures of servers, or of transmission between the decoders and the servers. Indeed, the default model of study in storage systems in the distributed algorithms literature assumes that communication links can have arbitrarily large delays [1]. Since it is more difficult to achieve synchronization in larger systems, asynchrony is an arguably justified modeling choice for distributed storage systems which are expected to scale in practice to cope with rising demands.

The problem of storing multiple versions of the data consistently in distributed asynchronous storage systems forms the basis of celebrated results in distributed computing theory [4]. From a practical perspective, algorithms designed to ensure consistency in asynchronous environments form the basis of several commercial storage products [5], [9], [7], [6]. We refer the reader to [5] for a detailed description of the Amazon Dynamo key value store, which describes a replication-based data storage solution. While [4], [5] use replication-based techniques for fault tolerance, the idea of using erasure coding for consistency has been used in recent distributed computing literature [10], [11], [12], [13]. In fact, these references use the idea of simple erasure coding that we referred to in Section I-A.

We note that the idea of storing versioned data has acquired some recent interest in information theory literature. In particular, some of the challenges of updating data in distributed storage systems have been studied in [14], [15], [16], [17]. These works complement our paper, and their ideas can perhaps be adapted to our framework to build efficient consistent data storage implementations.

C. Contributions and Organizations

Multi-version coding provides an information theoretic perspective to the problem of ensuring a consistent data storage service over an asynchronous distributed storage system. Our problem formulation is geared towards optimizing the storage cost per server node. We describe the multi-version coding problem formally in Section II. In Section III, we formally state our main results: a multi-version code construction that has a lower cost compared to replication and naive erasure coding, and an information theoretic lower bound on the storage cost. The proofs of the main results are provided in Sections IV, V and VI. In Section VII, we demonstrate the utility of multi-version codes using a toy model of asynchronous distributed storage systems. We discuss related areas of future work in our concluding section, Section VIII.

We describe our achievable multi-version code constructions in Section IV. The construction use a simple linear coding scheme without coding across versions. Moreover, our code construction satisfies a *causality* property (defined in Section II) that enables easier implementation, because our encoding strategy is agnostic to the order of arrival of the various message versions at the servers.

In Section V and VI we prove lower bounds on the storage cost for $\nu = 2$ versions and arbitrary ν, respectively. Our lower bounds imply that our code constructions are essentially optimal for certain families of parameters, and are close to optimal in general. It is worth noting that our problem formulation allows for all possible methods of encoding the versions. In particular, servers can encode multiple versions together, and use possibly non-linear methods of encoding the data. The tightness of our converse shows that, perhaps surprisingly, encoding each version *separately* using linear codes is close to optimal.

From a technical standpoint, the lower bound argument is interesting and challenging, especially when the number of versions ν is larger than 2. This is because, in commonly studied settings in multi-user information theory, the decoder has a specific set of messages that it wants to recover reliably. In contrast, in multi-version coding, a decoder is allowed to recover any one of a subset of messages correctly. As a consequence of the relatively unusual decoding constraint, commonly used methods of deriving converses need to be modified appropriately to obtain our lower bound. We provide a more detailed discussion on the technical aspects of the converse in Section III.

In Section VII, we describe a toy model of distributed storage that explicitly includes an arrival model for new versions and channels models for the links between the encoders, servers and decoders. We demonstrate the utility of multi-version codes in understanding the storage costs over the described toy model. Our study in Section VII provides a more refined understanding of the parameters of the multi-version coding problem in terms of the characteristics of a distributed storage system. Readers who are interested understanding the applications of multi-version codes, but not the technical details of the construction and the converse, can skip Sections IV-VI and read Section VII.

II. System Model: Multi-version Codes

We begin with some notations. For integers $i < j$, we use $[i, j]$ to represent the set $\{i, i+1, \ldots, j\}$. For integers $i > j$, we define $[i, j]$ as the empty set. We use $[j]$ to represent the set $[1, j]$. And $[j]$ is an empty set if $j < 0$. For any set of indices $S = \{s_1, s_2, \ldots, s_{|S|}\} \subseteq \mathbb{Z}$ where $s_1 < s_2 < \ldots < s_{|S|}$, and for any ensemble of variables $\{X_i : i \in S\}$, we denote the tuple $(X_{s_1}, X_{s_2}, \ldots, X_{s_{|S|}})$ by X_S. For a set $\{v_1, \ldots, v_n\}$ of elements, we use v_S to denote the set $\{v_i : i \in S\}$. If S is empty, then v_S is defined to be the empty set. For sets $S \subseteq T$, we write $T - S$ to be the set difference $\{i : i \in T, i \notin S\}$. We use log to represent log base 2.

We now define the multi-version coding problem. We begin with an informal definition, and present the formal definition in Definition 1. The *multi-version coding* problem is parameterized

by positive integers n, c, ν, M and q. We consider a setup with n servers. Our goal is to store ν independent versions of the message, where each version of the message is drawn from the set $[M]$. We denote the value of the ith version of the message by $W_i \in [M]$ for $i \in [\nu]$. Each server stores a symbol from $[q]$. Therefore, $\log q$ can be interpreted as the number of bits stored in a server. Every server receives an arbitrary subset of the versions. We denote $\mathbf{S}(i) \subseteq [\nu]$ to be the set of versions received by the ith server. We refer to the set $\mathbf{S}(i)$ as *the state* of the ith server. We refer to $\mathbf{S} = (\mathbf{S}(1), \ldots, \mathbf{S}(n)) \in \mathcal{P}([\nu])^n$ as the *system state*, where $\mathcal{P}([\nu])$ denotes the power set of $[\nu]$. For the ith server, denoting its state $\mathbf{S}(i)$ as $S = \mathbf{S}(i) = \{s_1, s_2, \ldots, s_{|S|}\}$ where $s_1 < s_2 < \ldots < s_{|S|}$, the ith symbol of the codeword is generated by an encoding function $\varphi_S^{(i)}$ that takes an input, $W_S = (W_{s_1}, W_{s_2}, \ldots, W_{s_{|S|}})$, and outputs an element in $[q]$.

We assume that there is a total ordering on the versions: if $i < j$, then W_j is interpreted as a later version of the message as compared with W_i. For any set of servers $T \subseteq [n]$, we refer to $\max \cap_{i \in T} \mathbf{S}(i)$ as the *latest common version* in the set of servers T. The purpose of multi-version code design is to generate encoding functions such that, for every subset $T \subseteq [n]$ of c servers, a message W_m should be decodable from the set T, where $m \geq \max \cap_{i \in T} \mathbf{S}(i)$ for every possible system state. The goal of the problem is to find the smallest possible storage cost per bit stored, or more precisely, to find the smallest possible value of $\frac{\log q}{\log M}$ over all possible multi-version codes with parameters n, c, v, M, q.

We present a formal definition next.

Definition 1 (Multi-version code) *An* (n, c, ν, M, q) multi-version code *consists of*

- *encoding functions*
$$\varphi_S^{(i)} : [M]^{|S|} \to [q],$$
for every $i \in [n]$ *and every* $S \subseteq [\nu]$*, and*
- *decoding functions*
$$\psi_{\mathbf{S}}^{(T)} : [q]^c \to [M] \cup \{NULL\},$$
for every set $\mathbf{S} \in \mathcal{P}([\nu])^n$ *and set* $T \subseteq [n]$ *where* $|T| = c$*,*

that satisfy

$$\psi_{\mathbf{S}}^{(T)} \left(\varphi_{\mathbf{S}(t_1)}^{(t_1)}(W_{\mathbf{S}(t_1)}), \ldots, \varphi_{\mathbf{S}(t_c)}^{(t_c)}(W_{\mathbf{S}(t_c)}) \right) = \begin{cases} W_m & \text{for some } m \geq \max \cap_{i \in T} \mathbf{S}(i), \text{if } \cap_{i \in T} \mathbf{S}(i) \neq \emptyset, \\ NULL, & \text{o.w.,} \end{cases} . \tag{1}$$

for every $W_{[\nu]} \in [M]^\nu$*, where* $T = \{t_1, t_2, \ldots, t_c\}$*,* $t_1 < \cdots < t_c$*.*

Remark 1 *Suppose* $M \geq \nu$*, and let* $\mathbf{S}$ *be the* n*-tuple server state. Consider servers* $T \subseteq [n], |T| = c$*, and the union of their states* $S' = \cup_{t \in T} \mathbf{S}(t)$*. Then for any given tuple* $W_{[\nu]}$*, the decoding function* $\psi_{\mathbf{S}}^{(T)}$ *decodes either NULL, or a value that is equal to* W_j*, for some version* $j \in S'$*.*

We normalize the storage cost by the size of one version, that is $\log M$.

Definition 2 (Storage cost of an (n, c, ν, M, q) **multi-version code)** *The* storage cost *of an* (n, c, ν, M, q) *multi-version code is defined to be equal to* $\frac{\log q}{\log M}$*.*

As mentioned in the introduction, replication, where the latest version is stored in every server, i.e., $\varphi_{\mathbf{S}_i}^{(i)}(W_{\mathbf{S}_i}) = W_{\max(\mathbf{S}(i))}$ incurs a storage cost of 1. An alternate strategy would be to separately encode every version using an MDS code of length n and dimension c, with each server storing an MDS codeword symbol corresponding to every version that it has received. Such a coding scheme would achieve a storage cost of ν/c, for sufficiently large q.

For parameters n, c, ν, the goal of the multi-version coding problem is to find the infimum, taken over the set of all (n, c, ν, M, q) codes, of the quantity: $\frac{\log q}{\log M}$.

It is useful to understand the connection of the parameters of the multi-version coding problem and the physical characteristics of a distributed storage system. The parameter n naturally represents the number of servers across which we intend to encode the data of the storage system. The parameter c is connected to the failure tolerance; in particular, an (n, c, ν, M) multi-version code can protect against $n - c$ server failures since the latest common version is recoverable among any c nodes. In Section VII, we show through a toy model of distributed storage, that the parameter ν is related to the degree of asynchrony in the system.

Notice that in our definition, the encoding function of each server depends only on the subset of versions that has arrived at the server, but not on the order of the arrival of the versions. From a practical standpoint, it could be useful to modify the definition of multi-version codes to let the encoding function depend on the order of arrival of the versions. However, in this paper, we use a different approach. We introduce the notion of *causal* multi-version codes that obviates the need for incorporating the order of arrival in the definition.

Definition 3 (Causal codes) *A multi-version code is called* causal *if the encoding function satisfies: for all $S \subseteq [\nu], j \in S, i \in [n]$, there exists a function*

$$\hat{\varphi}^{(i)}_{S,j} : [q] \times [M] \to [q],$$

such that

$$\varphi^{(i)}_S(W_S) = \hat{\varphi}^{(i)}_{S,j}(\varphi^{(i)}_{S\setminus\{j\}}(W_{S\setminus\{j\}}), W_j).$$

To understand the notion of causal codes, imagine that a sequence of versions arrive at a server in an arbitrary order. If a casual multi-version code is used, then the encoding function at the server is only a function of its stored information and the value of the arriving version. We anticipate causal multi-version codes to be more relevant to practical distributed storage systems than non-causal codes. In fact, we demonstrate the utility of causal multi-version codes in storage systems through our toy model of distributed storage in Section VII. All the code constructions that we present in this paper are causal.

III. Main Results

In this section, we formally present the main results of this paper: Theorem 1, which states the storage cost of an achievable code construction, and Theorem 2, which states the result of a converse that lower bounds the storage cost of an arbitrary multi-version code. We present and discuss Theorem 1 in Section III-A. We present and discuss Theorem 2 in Section III-B.

A. Achievability

Theorem 1 *Given parameters (n, c, ν), there exists a causal (n, c, ν, M, q) multi-version code with a storage cost that is equal to*

$$\max\left\{\frac{\nu}{c} - \frac{(\nu-1)}{tc}, \frac{1}{t}\right\},$$

where

$$t = \begin{cases} \left\lceil \frac{c-1}{\nu} \right\rceil + 1, & \text{if } c \geq (\nu-1)^2, \\ \left\lfloor \frac{c}{\nu-1} \right\rfloor, & \text{if } c < (\nu-1)^2. \end{cases}$$

The achievable scheme of Theorem 1 has a strictly smaller storage cost as compared with replication and simple MDS codes. In particular, if ν is comparable to c, our achievable code

constructions could improve significantly upon replication and simple MDS codes. If $\nu = c-1$, our storage cost is approximately half the storage cost of the minimum of replication and simple MDS codes for large values of c. It is instructive to note that if $\nu|(c-1)$, the storage cost is $\frac{\nu}{c+\nu-1}$.

Our code constructions are quite simple since we do not code across versions. The main idea of our approach is to carefully allocate the storage "budget" of $\log q$ among the various versions in a server's state, and for each version, store an encoded value of the allocated size.

In [18], we studied a special case of the multi-version code that decodes *only* the latest common version. Here, we allow the decoder to return a version later than the latest common version. It is interesting to note that, under the relaxed definition of multi-version coding presented here, the converse of [18] is not applicable. In fact, the achievable scheme of Theorem 1 achieves a storage cost that is lower than the storage cost lower bound of [18] by exploiting the fact that a version that is later than the latest common version can be recovered. We plot the performance of Theorem 1 in Fig. 2.

B. Converse

Theorem 2 *A (n, c, ν, M, q) multi-version code with $n \geq c+\nu-1$ and $M \geq \nu$ must satisfy*

$$\frac{\log q}{\log M} \geq \frac{\nu}{c+\nu-1} - \frac{\log(\nu^{\nu}\binom{c+\nu-1}{\nu})}{(c+\nu-1)\log M}.$$

In the lower bound expression of Theorem 2, the second term on the right hand side vanishes as $\log M$ grows. For the case of $\nu = 2$ versions, we show a somewhat stronger result in section V. In particular, for $\nu = 2$, we show that the second term in the theorem can be improved to be $\frac{\log c}{(c+1)\log M}$.

The lower bound of Theorem 2 indicates that the storage cost, as a function of M, is at least $\nu/(c+\nu-1) + o(1)$. When $\nu|(c-1)$, the storage cost of Theorem 1 approaches the lower bound of Theorem 2 as $\log M$ grows, and is therefore asymptotically optimal. The multi-version coding problem remains open when $\nu \nmid (c-1)$. We establish a connection between the parameter ν and the degree of asynchrony in a storage system in Section VII. The converse of Theorem 2 in combination with the achievable scheme of Theorem 1 therefore implies that the greater the degree of asynchrony in a storage system, the higher the storage cost. In particular, as ν tends to infinity, the storage cost is one. Therefore, in the limit of infinite asynchrony, the gains of erasure coding vanish, and replication is essentially optimal.

The assumption that $\log M$ grows while c, ν are kept fixed is a reasonable first order assumption in our study of storage costs because, in systems where storage cost is large, the file size is typically large. The study of multi-version codes for finite M is, nonetheless, an interesting open problem.

In the lower bound proofs, we develop an algorithm that finds a system state that requires a large storage cost per server to ensure correct decoding. Our approach to deriving the converse has some interesting conceptual aspects. The standard approach to derive converses for a noiseless multi-user information theory problem is as follows: (i) express the encoder and decoder constraints using conditions on the entropy of the symbols, (ii) use Shannon information inequalities to constrain the region spanned by the entropies of the variables, and (iii) eliminate the intrinsic variables of the system to get bounds that must be satisfied by the extrinsic random variables. Usually performing steps (i),(ii) and (iii) requires ingenuity because they tend to be computationally intractable for many problems of interest (see [19] for example). For the multi-version coding problem, we face some additional challenges since we cannot use steps (i),(ii) and (iii) directly.

To understand the challenges, we re-examine our approach to deriving a converse in [18], where the decoder was restricted to recovering the latest common version. For the problem in [18], the standard approach to deriving converses in multi-user information theory was applicable. In the

multi-version coding problem, note that for state $\mathbf{S}$ we may express the constraint at the encoder as

$$\log q \geq H(\varphi^{(i)}_{\mathbf{S}(i)}(W_{[\nu]})), \quad H(\varphi^{(i)}_{\mathbf{S}(i)}(W_{[\nu]})|W_{\mathbf{S}(i)}) = 0, \tag{2}$$

for every $i \in [n]$ and every possible state $\mathbf{S}(i) \in \mathcal{P}([\nu])$, and for any distribution on the messages in the system.

In [18], where we constrained the decoder to decode the latest common version, we were able to similarly express the constraint at the decoder. For example, consider the first c servers and a state $\mathbf{S}$ where the latest common version is $k \in [\nu]$ for the servers $[c]$, we expressed the decoding constraint as

$$H(W_k|\varphi^{(1)}_{\mathbf{S}(1)}(W_{[\nu]}), \varphi^{(2)}_{\mathbf{S}(2)}(W_{[\nu]}), \ldots, \varphi^{(c)}_{\mathbf{S}(c)}(W_{[\nu]})) = 0. \tag{3}$$

Note that the above equation can be written for every possible state $\mathbf{S}$. If we assume a uniform distribution for all the messages, we have $H(W_{[\nu]}) = \nu \log M$. Combining this with (2) and (3), and using Shannon information inequalities, we obtained a bound on $\frac{\log q}{\log M}$.

In the problem we consider here, we can similarly write the constraints (2). However, in the multi-version coding problem, the constraint at the decoder cannot be expressed in a manner analogous to (3) because the decoder does not have a *specific* message to decode. At any given state of the system, a decoder that connects to c servers is allowed to decode one of several messages. In particular, imagine that version k is the latest common version for the servers $[c]$, when the system state is $\mathbf{S}$. Then the decoder is allowed to decode any one of $W_k, W_{k+1}, \ldots, W_\nu$ for state $\mathbf{S}$. In fact, one can conceive of a decoder that may return different message versions for the same state, depending on the message realization. For instance, one can conceive of multi-version code where, for a given state $\mathbf{S}$, when the encoded message tuple is $W_{[\nu]} = (W_1, W_2, \ldots, W_\nu)$, the decoder $\psi^T_{\mathbf{S}}$ outputs W_k, and when the encoded message tuple is $\overline{W}_{[\nu]} = (\overline{W}_1, \overline{W}_2, \ldots, \overline{W}_\nu)$, the decoder $\psi^{[c]}_{\mathbf{S}}$ outputs $\overline{W}_{k+1}$. As a consequence of the unusual nature of the decoding constraint, the converse proofs in Sections V and VI has an unusual structure. In particular, we carefully construct some auxilliary variables and write constraints on the entropies of the constructed variables to replace (3). Our approach to deriving converses is potentially useful for understanding *pliable index coding problem* and other recently formulated content-type coding problems [20], [21], where the decoder does not have a unique message, but is satisfied with reliably obtaining one of a given subset of messages.

IV. Code Construction

We describe our construction in this section. We start with code construction for $\nu = 2$ versions, and then generalize the construction for arbitrary ν. In the end, we show that our construction is a multi-version code in Theorem 4.

In our construction, each server encodes different versions separately. So that the total number of bits stored at a server is the sum of the storage costs of each of the versions in the server state. The encoding strategy at the servers satisfies the following property: Suppose that Server i is in state $S \subseteq [\nu]$ and stores $\alpha^{(S)}_{i,v} \log M$ bits of Version v, then Version v can be recovered from the c servers $i_1, i_2, \ldots i_c$, so long as

$$\sum_{j=1}^{c} \alpha^{(S)}_{i_j,v} \geq 1. \tag{4}$$

Note that such an encoding function can be found for a sufficiently large value of q using standard coding techniques. In fact, suppose that the message W_v is interpreted as a vector over some finite

field. We let Server i store $\alpha_{i,v}^{(S)} \log M$ random linear combinations of elements in the vector W_v. Then Version v can be recovered from any subset of c servers satisfying (4) with a non-zero probability so long as the field size is sufficiently large. As a result, there exists a deterministic code that decodes Version v if (4) is satisfied. We also note that, in our approach, the storage allocation $\alpha_{i,v}^{(S)}$ only depends on the server state but not on the server index. Therefore, we can write $\alpha_{i,v}^{(S)} = \alpha_v^{(S)}$ for any Server i at a nonempty state $S \subseteq [\nu]$.

As a result, to describe our construction, we only need to specify the parameter $\alpha_v^{(S)}$ for every possible server state $S \subseteq [\nu]$ and every $v \in S$. That is, we only need to specify the information amount corresponding to Version v stored at a server in state S. We denote $\alpha = \frac{\log q}{\log M}$ as the storage cost. Note that we have $\alpha = \max_{S \subseteq [\nu]} \sum_{v \in S} \alpha_v^{(S)}$.

Definition 4 (Partition of the servers) *For every system state* $\mathbf{S}$ *where* $\mathbf{S}(i) \neq \emptyset$ *for all* $i \in [n]$*, we define a partition of the* n *servers into* ν *groups as follows. For* $i \in [\nu]$*, Group* i *has the set of servers which have Version* i *as the latest version.*

For instance, if $\nu = 2$, Group 1 has the servers in state $\{1\}$, and Group 2 contains the servers in states $\{2\}$ and $\{1, 2\}$.

A. Code Construction for $\nu = 2$

We start by describing our construction for the case of $\nu = 2$ versions shown in Table IV. In Theorem 3, we show that our construction is a multi-version code.

Construction 1 *Define*

$$t = \lceil \frac{c-1}{2} \rceil + 1,$$

We construct a code for $\nu = 2$ *with storage cost* $\alpha = \frac{2t-1}{tc}$*. More specifically, we assign*

$$\alpha_1^{(\{1,2\})} = \alpha - \frac{1}{t}, \alpha_1^{(\{1\})} = \alpha, \alpha_2^{(\{1,2\})} = \alpha_2^{(\{2\})} = \frac{1}{t}.$$

One can see that the code in Table III is an example that follows the above storage allocation. It is instructive to note that if c is odd, then $\alpha_1^{(\{1,2\})} = 0, \alpha_2^{(\{1,2\})} = \alpha = \frac{2}{c+1}$. This means that if c is odd, each server simply stores $\frac{2}{c+1} \log_2 M$ bits of the latest version. That is, servers in Group 1 store $\frac{2}{c+1} \log_2 M$ bits of Version 1, and servers in Group 2 store $\frac{2}{c+1} \log_2 M$ bits of Version 2. By the pigeon-hole principle, there are at least $\frac{c+1}{2}$ servers either in Group 1 or Group 2; therefore a decoder can connect to any c servers and decode either version 1 or version 2. Furthermore, the decoder always obtains the latest common version, or a later version. As a consequence, our storage strategy forms a multi-version code. We next provide a formal proof next handling odd and even values of c together.

Theorem 3 *Construction* 1 *is an* $(n, c, \nu = 2, M, q)$ *multi-version code with storage cost of* $\frac{2}{c+1}$ *for odd* c*, and* $\frac{2(c+1)}{c(c+2)}$ *for even* c*.*

Proof: Consider any set of c servers. We argue that the latest common version or a later version is decodable for every possible state.

Case I. If the latest common version is Version 2, then all the c servers are in Group 2. Since we have $c \geq t$ servers, and each server contains $1/t$ amount of Version 2, Version 2 is recoverable.

Case II. If the latest common version is Version 1, then the c servers may be in state $\{1\}$ or state $\{1, 2\}$. If there are at least t servers in state $\{1, 2\}$, then we can recover Version 2. Otherwise, there

State	$\{1,2\}$	$\{1\}$	$\{2\}$
Ver 1	$\alpha - 1/t$	α	
Ver 2	$1/t$		$1/t$

TABLE IV
Storage allocations for code construction with $\nu = 2$ versions. Note that $t = \lceil \frac{c-1}{2} \rceil + 1$, and the storage cost is $\alpha = \frac{2t-1}{tc}$. More specifically, $\alpha = 1/t$ for odd values of c and $\alpha = \frac{2(c+1)}{c(c+2)}$ for even values of c.

are at most $t-1$ servers in state $\{1,2\}$, and at least $c-t+1$ servers in state $\{1\}$. Thus the total amount of Version 1 in these servers is at least

$$(c-t+1)\alpha + (t-1)(\alpha - 1/t) = 1,$$

so we can recover Version 1. ■

B. Code Construction for an Arbitrary ν

We generalize our constructions to arbitrary values of ν. We first provide our constructions in 2 and then prove in Theorem 4 that our construction is a multi-version code.

Construction 2 *Define a parameter t as follows.*

$$t = \begin{cases} \lceil \frac{c-1}{\nu} \rceil + 1, & c > (\nu-1)^2, \\ \lceil \frac{c}{\nu-1} \rceil, & c \leq (\nu-1)^2. \end{cases} \tag{5}$$

We construct the (n, c, ν, M, q) code with storage cost

$$\alpha = \frac{\log q}{\log M} = \max\left\{ \frac{\nu t - \nu + 1}{tc}, \frac{1}{t} \right\}. \tag{6}$$

For state S, the parameter $\alpha_v^{(S)}$ is set as follows:

- *If Version j, $j \geq 2$, is the latest version in state S, then $\alpha_j^{(S)} = \frac{1}{t}$, that is, store $\frac{1}{t}\log_2 M$ bits of Version j.*
- *If Version j, $j \geq 2$, is the latest version in state S, and $\{1\} \in S$, then, $\alpha_1^{(S)} = \alpha - \frac{1}{t}$, that is, store $\left(\alpha - \frac{1}{t}\right)\log_2 M$ bits of Version 1.*
- *If Version 1 is the latest version, namely $S = \{1\}$, then $\alpha_1^{(S)} = \alpha$. That is, store $\alpha \log M$ bits of Version 1.*

Note that in our construction, a server in Group j only stores encoded symbols of Version j and possibly Version 1.

It is useful to note that if $c > (\nu-1)^2$, then $t \geq \nu$ and if $c \leq (\nu-1)^2$, then $t < \nu$.

Remark 2 *It can be readily verified that the storage cost of Construction 2 can be expressed more explicitly as follows:*

$$\alpha = \begin{cases} \frac{1}{t}, & \text{if } c \in [(t-1)\nu + 1, (\nu-1)t], t < \nu, \\ \frac{\nu t - \nu + 1}{tc}, & \text{otherwise.} \end{cases}$$

where t is defined as in (5)*.*

Remark 3 *We note that when $\nu|(c-1)$, we have*

$$t = \frac{c + \nu - 1}{\nu}$$

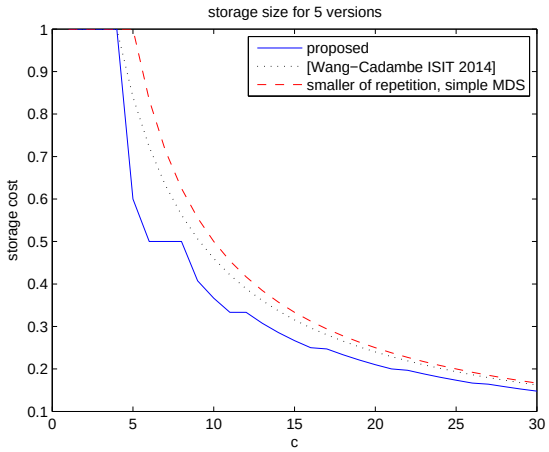

Fig. 2. Comparison between the construction for the code in Construction 2, the code in [18], and the smaller of replication and the simple MDS code. We fix $\nu = 5$ versions and plot results for different number of connected servers, c.

	Group 1	Group 2		Group 3			
State	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{3\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$
Version 1	$1/3$		0		0		0
Version 2		$1/3$	$1/3$			0	0
Version 3				$1/3$	$1/3$	$1/3$	$1/3$

TABLE V
Storage allocations for code with $c = 7, \nu = 3, \alpha = 1/3$.

irrespective of whether c is bigger then $(\nu-1)^2$ or not. As a result, we have $\alpha = \frac{\nu}{c+\nu-1}$ when $\nu|(c-1)$. In this case, as per Construction 2, a server in Group i stores $\frac{\nu}{c+\nu-1}\log_2 M$ bits of version i, and does not store any of the older versions. A simple pigeon-hole principle based argument suffices to ensure that any decoder that connects to c servers decodes the latest common version among the servers, or a later version.

In Figure 2 we show the storage cost of the construction with $\nu = 5$ versions, we can see the advantage of the proposed code compared to previous results.

Table V is an example with $c = 7, \nu = 3, \alpha = 1/3$. Notice in this case, $\nu|(c-1)$, each server only stores information about the latest version it receives, and does not store any information about any of the older versions. It is easy to see that when connected to $c = 7$ servers with a common version, at least one version, say Version i, can be decoded from 3 servers in Group i using similar arguments as the proof of Theorem 3.

Table VI is an example for $t = 3, c = 5, \nu = 3, \alpha = 7/15$. In this example, the storage cost of the states are not equal, but one can simply treat the worst-case size as α. One can check that the above code recovers the latest common version, or a version that is later than the latest common version. For example, suppose the latest common version is Version 1.

	Group 1	Group 2		Group 3			
State	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{3\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$
Version 1	$7/15$		$2/15$		$2/15$		$2/15$
Version 2		$1/3$	$1/3$			0	0
Version 3				$1/3$	$1/3$	$1/3$	$1/3$

TABLE VI
Server storage allocations for $c = 5, \nu = 3, \alpha = 7/15$.

- If at least three of the c servers are in Group 2, then Version 2 is recoverable.
- If at least three of the c servers are in Group 3, then Version 3 is decodable.
- Otherwise, among the c servers, at most two servers are in Group 2, at most two servers are in Group 3, and at least one server is in state $\{1\}$. The amount of information of Version 1 in these c servers is at least $7/15 + 2/15 \times 4 = 1$, which implies that Version 1 is recoverable.

Theorem 4 *The code in Construction* 2 *is a casual* (n, c, ν, M, q) *multi-version code.*

Proof: To show a version is recoverable, it suffices to show that the total storage allocation for that version in the connected servers is at least 1. Let j be the latest common version among c servers. Note that there are at most $\nu - j + 1$ groups, since Group 1, Group 2, ..., Group $j-1$ are empty.

Case I. When $j \geq 2$, there exists a group, say Group k, with at least $\lceil \frac{c}{\nu-j+1} \rceil \geq \lceil \frac{c}{\nu-1} \rceil$ servers. In our construction, each server in Group k stores $\frac{1}{t}$ of Version k. To prove the theorem, it suffices to show that $\lceil \frac{c}{\nu-1} \rceil \geq t$, since this implies that Version k is recoverable from the servers in Group k. When $c \leq (\nu-1)^2$, then $\lceil \frac{c}{\nu-1} \rceil = t$. Therefore, we need to show this for $c > (\nu-1)^2$. When $c > (\nu-1)^2$, we have $t = \lceil \frac{c-1}{\nu} \rceil + 1$. Therefore, we have $c \in [\nu(t-2)+2, \nu(t-1)+1]$. Notice also that $t \geq \nu$. These imply the following.

$$\begin{aligned}
&\lceil \frac{c}{\nu-1} \rceil \\
\geq &\lceil \frac{\nu(t-2)+2}{\nu-1} \rceil \\
= &\lceil t-1+\frac{t-\nu+1}{\nu-1} \rceil \\
\geq &\lceil t-1+\frac{1}{\nu-1} \rceil \\
= &t.
\end{aligned}$$

Therefore, the theorem is proved for the case where $j \geq 2$.

Case II. When $j = 1$ is the latest common version, if Group i has at least t servers for any $2 \leq i \leq \nu$, then Version i is recoverable and therefore the theorem is proved. Otherwise, there are at most $t-1$ servers in Group i, for all $2 \leq i \leq \nu$, each of which stores $\alpha - \frac{1}{t}$ size of Version 1; and thus at least $c - (\nu-1)(t-1)$ servers in Group 1, each storing α size of Version 1. The total storage cost for Version 1 in these servers is at least

$$(\alpha - \frac{1}{t})(\nu-1)(t-1) + \alpha(c-(\nu-1)(t-1)).$$

And by the choice of α, we know the above amount is at least 1. Therefore, Version 1 can be recovered.

If a server is in State S, Version j arrives, the server simply need to encode based on its stored information and information of Version j: (i) if $j \leq \max S$, the server does nothing; (ii) else the server removes $1/t$ amount of information of Version $\max S$, and replace it with $1/t$ amount of information of Version j. Therefore, the construction is causal. ■

From the above results, we can prove Theorem 1.

Proof of Theorem 1*:* Since Construction 2 is a casual (n, c, ν, M, q) multi-version code by Theorem 4, and has storage cost as in (6), the theorem is proved. ■

In fact, the construction in this section is inspired by computer search for $\nu = 3$ and small values of c using integer linear programming on the allocated storage sizes. In particular, denote by $\mathbf{S} = (\mathbf{S}(1), \ldots, \mathbf{S}(c))$ a c-tuple server state. Define the latest common version $m(\mathbf{S}) = \max \cap_{i=1}^{c} \mathbf{S}(i)$, for $\cap_{i=1}^{c} \mathbf{S}(i) \neq \emptyset$. We assume that the versions are coded separately, and use $\alpha_v^{(S)}$ to denote the

storage allocation of Version v at a server in state S, for $v \in S$. Then we have the optimization problem with respect to variables $\alpha, \alpha_v^{(S)}$:

$$\text{minimize } \alpha, \tag{7}$$

$$\text{s.t. } \alpha_v^{(S)} \geq 0, \text{ for all } S \in \mathcal{P}([\nu]), v \in S, \tag{8}$$

$$\sum_{v \in S} \alpha_v^{(S)} \leq \alpha, \text{ for all } S \in \mathcal{P}([\nu]) \tag{9}$$

$$\bigvee_{v=m(\mathbf{S})}^{\nu} \sum_{i=1}^{c} \alpha_v^{(\mathbf{S}(i))} \geq 1, \text{ for all } \mathbf{S} \in \mathcal{P}([\nu])^c, \bigcap_{i=1}^{c} \mathbf{S}(i) \neq \emptyset \tag{10}$$

where $\vee$ is the "or" operator. In words, we want to minimize the storage size α, subject to the constraint thats the allocation sizes are non-negtive (equation (8)), every node stores no more than α (equation (9)), and the latest common version $m(\mathbf{S})$, or a later version should have enough storage size to ensure recovery (equation (10)).

We can use the Big M Method [22] to convert the "or" constraints in (10) to "and" constraints and solve it by integer linear programming. On application of the Big M method, our optimization problem (7) can be equivalently expresses as

$$\begin{aligned}
&\text{minimize } \alpha, \\
&\text{s.t. } \alpha_v^{(S)} \geq 0, \text{ for all } S \in \mathcal{P}([\nu]), v \in S, \\
&\sum_{v \in S} \alpha_v^{(S)} \leq \alpha, \text{ for all } S \in \mathcal{P}([\nu]) \\
&\text{for all } \mathbf{S} \in \mathcal{P}([\nu])^c, \cap_{i=1}^{c} \mathbf{S}(i) \neq \emptyset : \\
&\quad \sum_{i=1}^{c} \alpha_v^{(\mathbf{S}(i))} \geq 1 - y_v, \forall v \geq m(\mathbf{S}), \\
&\quad \sum_{v \geq m(\mathbf{S})} y_v \leq \nu - m(\mathbf{S}) \\
&\quad 0 \leq y_v \leq 1, y_v \in \mathbb{Z}, \forall v \geq m(\mathbf{S}),
\end{aligned}$$

Plugging in small values of c and $\nu = 3$, one can obtain the constructed code as one solution to the above optimization problem.

We would like to point out the low complexity to update information in the servers in our constructions. As the theorem states, our constructions are *causal* codes. Whenever a version arrives that is the latest among all received ones, the server only needs to delete (a part of) the older version/s and store the latest version. In addition, when $\nu|(c-1)$, no matter how many versions are in the server state, the server stores information about only the latest version. In this case, every server only manages a single version and has relatively low complexity compared to simple MDS coding scheme.

V. Proof of Converse for 2 Versions

In this section, we prove Theorem 2 for the case of $\nu = 2$ versions. The proof is the inspiration for the proof of general value of ν in the next section.

Consider any $(n, c, 2, M, q)$ multi-version code, and consider the first $c \leq n$ servers. We note here that an arbitrary set of c servers can be considered for the converse. We consider the first c servers without loss of generality. In particular, we let the server state be the empty set $\emptyset$ if the server index is larger than c, and we always try to decode from the first c servers.

Informally, the main idea of our argument is as follows. We begin with the following claim: given the values of the two version, $W_{[2]} = (W_1, W_2)$, there exist two system states, $\mathbf{S}_1$, $\mathbf{S}_2 \in \mathcal{P}([\nu])^n$ such that

- the states $\mathbf{S}_1, \mathbf{S}_2$ differ only in the state of one server, say, Server A, and
- W_1 is decodable from the symbols stored among the first c servers in state $\mathbf{S}_1$, and W_2 is decodable from the symbols stored in the first c servers in state $\mathbf{S}_2$.

However, notice that the encoded symbols of the servers $[n] - \{A\}$ are the same in both states $\mathbf{S}_1$ and $\mathbf{S}_2$. This implies that both Version 1 and Version 2 are decodable from the following $c+1$ symbols: the c codeword symbols of first c servers in state $\mathbf{S}_1$, and the codeword symbol of the A-th server in state $\mathbf{S}_2$. Note that $\mathbf{S}_1$, $\mathbf{S}_2$, and A are chosen based on the values of $W_{[2]}$, in fact, they may be viewed as functions of $W_{[2]}$.

We now construct $c+1$ variables $Y_{[c]}, Z$ as follows: Y_i is the value stored in the i-th server for $i \in [c]$, when the server is in state $\mathbf{S}_1(i)$, and Z is the value stored in the A-th server when the server is in state $\mathbf{S}_2(A)$. Notice that the variables $Y_i, i \in [c], Z$ all belong to $[q]$.

Since these 2 versions, W_1, W_2, each of alphabet size M, are decodable from the $c+1$ auxilliary variables $Y_{[c]}, Z$ with an alphabet of size q, we need $(c+1)\log q \geq 2\log M + o(1)$. We provide a formal proof next.

Formal Proof

Let $\mathcal{S}$ be the set of system states

$$\begin{aligned}\mathcal{S} = \{&\mathbf{S} \in \mathcal{P}([\nu])^n : \\ &\mathbf{S}(i) = \{1,2\}, \forall i \in [x], \\ &\mathbf{S}(i) = \{1\}, \forall i \in [x+1, c], \\ &\mathbf{S}(i) = \emptyset, \forall i \in [c+1, n], \\ &\forall x \in [0, c]\}.\end{aligned}$$

For given values of $W_{[2]}$, we define two subsets of $\mathcal{S}$ according to the version decoded from Servers $[c]$, denoted by $\mathcal{S}_1$, $\mathcal{S}_2$: for $i = 1, 2$,

$$\mathcal{S}_i = \{\mathbf{S} \in \mathcal{S} : \psi_{\mathbf{S}}^{([c])}(\varphi_{\mathbf{S}(1)}^{(1)}(W_{\mathbf{S}(1)}), \ldots, \varphi_{\mathbf{S}(c)}^{(c)}(W_{\mathbf{S}(c)})) = W_i\}.$$

We can see that any system state in the set $\mathcal{S}$ has the following structure: for some $x \in [0, c]$, the first x servers have both versions, servers $[x+1, c]$ have the first version, and the remaining servers have no version. Notice that for any system state in $\mathcal{S}$, there exists a latest common version among the first c servers. This means that for every state in $\mathcal{S}$, the corresponding decoding function must return Version 1 or Version 2. Thus, $\mathcal{S}_1 \cup \mathcal{S}_2 = \mathcal{S}$. The subset $\mathcal{S}_i$ is one where the decoding function returns W_i, the value of Version i, from the first c servers, for $i = 1, 2$. When $W_1 \neq W_2$, for any state in $\mathcal{S}$ the decoding function returns only one version, therefore $\mathcal{S}_1, \mathcal{S}_2$ forms a partition of $\mathcal{S}$. When $W_1 = W_2$, for any state we can return both versions, so $\mathcal{S}_1 = \mathcal{S}_2 = \mathcal{S}$.

Claim 1 *For any achievable $(n, c, 2, M, q)$ code, and given values $W_{[2]}$, there are two states $\mathbf{S}_1, \mathbf{S}_2 \in \mathcal{P}([\nu])^n$ such that*

- *The n-length tuples $\mathbf{S}_1$ and $\mathbf{S}_2$ differ in one element indexed by $A \in [c]$, that is, they differ with respect to the state of at most one of the first c servers.*
- *$\mathbf{S}_1 \in \mathcal{S}_1$ and $\mathbf{S}_2 \in \mathcal{S}_2$.*

Proof: Assume $W_1 = W_2$, then simply take $\mathbf{S}_1$ such that their first c elements are all $\{1\}$, and the remaining elements are all $\emptyset$. Take $\mathbf{S}_2$ the same as $\mathbf{S}_1$ except that the first element is $\{1, 2\}$. They differ at index $A = 1$. One can easily check the conditions in the claim.

Assume $W_1 \neq W_2$. Consider a state with the smallest number, A, of occurrences of $\{1,2\}$ in partition $\mathcal{S}_2$ and denote this state as $\mathbf{S}_2$. In other words,

$$\mathbf{S}_2 = \arg\min_{\mathbf{S}\in\mathcal{S}_2} |\{i : \mathbf{S}(i) = \{1,2\}\}|.$$

Let $\mathbf{S}_1$ be a state obtained by replacing the A-th element of $\{1,2\}$ of $\mathbf{S}_2$ by $\{1\}$. Notice that, since the number of occurrences of $\{1,2\}$ in the state tuple $\mathbf{S}_1$ is smaller than the number occurrences of $\mathbf{S}_2$, the state $\mathbf{S}_1$ does not lie in partition $\mathcal{S}_2$. Furthermore state $\mathbf{S}_1$ lies in $\mathcal{S}$. Therefore $\mathbf{S}_1$ lies in partition $\mathcal{S}_1$. It is easy to verify that states $\mathbf{S}_1$ and $\mathbf{S}_2$ satisfy the conditions of the claim. ■

Next, we define $c+1$ variables $Y_{[c]}, Z$. Denote by A the number of servers in $\{1,2\}$ for $\mathbf{S}_2$ found by the proof of Claim 1, or the largest server index in state $\{1,2\}$ for $\mathbf{S}_2$. Denoted by $Y_{[c]}$ the values stored in the first c servers when the system state is $\mathbf{S}_1$: for $i \in [c]$,

$$Y_i = \varphi^{(i)}_{\mathbf{S}_1(i)}(W_{\mathbf{S}_1(i)}).$$

Denote by Z the value stored in the A-th server when the server state is $\mathbf{S}_2(A) = \{1,2\}$:

$$Z = \varphi^{(A)}_{[2]}(W_{[2]}).$$

Proof of Theorem 2 for $\nu = 2$: Consider any $(n, c, \nu = 2, M, q)$ code. Given the value of the variable A, we can determine the two states $\mathbf{S}_1, \mathbf{S}_2$ as in Claim 1. Therefore, if we are given the values of A, $Y_{[c]}$ and Z, we can determine the values of $W_{[2]}$:

$$\begin{aligned} W_1 &= \psi^{([c])}_{\mathbf{S}_1}(Y_{[c]}), \\ W_2 &= \psi^{([c])}_{\mathbf{S}_2}(Y_{[A-1]}, Z, Y_{[A+1,c]}). \end{aligned}$$

Therefore, there is a bijective mapping from $(Y_{[c]}, Z, A)$ to $W_{[2]}$. Therefore, the following equation is true for any distribution over $W_{[2]}$

$$H(W_{[2]}|Y_{[c]}, Z, A) = 0.$$

Therefore,

$$I(Y_{[c]}, Z; W_{[2]}|A) = H(W_{[2]}|A) = H(W_{[2]}) - I(W_{[2]}; A) \geq H(W_{[2]}) - \log c.$$

The last inequality holds because the alphabet size of A is at most c. We have the following chain of inequalities:

$$\begin{aligned} &(c+1)\log q \\ \geq &I(Y_{[c]}, Z; W_{[2]}|A) \\ \geq &H(W_{[2]}) - \log c. \end{aligned}$$

The first inequality follows because Y_i, Z belong to $[q]$ for every $i \in [c]$. Since the code should work for any distribution of $W_{[2]}$, we assume that W_1, W_2 are independent and uniformly distributed over $[M]$. Then the theorem statement follows. ■

It is instructive to observe that in the above proof (and similarly the proof for general ν of Theorem 2) that, for different values $W_{[2]}$, the parameters $A, Y_{[c]}, Z$ may take different values. If we constrain the multi-version codes so that the decoding function $\psi^{(T)}_{\mathbf{S}}$ in Definition 1 returns a fixed version index m given the system state $\mathbf{S}$ and the set of connected servers T, $T \subseteq [n], |T| = c$, then the lower bound can be strengthened [23]. Our formulation converse proof here is applicable even for multi-version codes where the decoded version index m depends not only on $\mathbf{S}$ and T, but could also depend on the values $W_{[\nu]}$.

VI. Proof of Converse for an Arbitrary ν

In this section, we provide a proof of Theorem 2 for arbitrary values of ν. Given an (n, c, ν, M, q) multi-version code, we can obtain a (c, c, ν, M, q) multi-version code by simply using the encoding functions corresponding to the first c servers of the given (n, c, ν, M, q) code. Furthermore, the storage cost of the (c, c, ν, M, q) multi-version code is identical to the storage cost of the (n, c, ν, M, q) multi-version code. Therefore, to derive a lower bound on the storage cost, it suffices to restrict n to be equal to c. Consider an arbitrary (c, c, ν, M, q) multi-version code. Consider the set $\mathcal{W}$ of message-tuples whose components are distinct, that is,

$$\mathcal{W} = \{W_{[\nu]} : W_i \neq W_j \text{ if } i \neq j\}. \tag{11}$$

Denote by $\mathbb{1}_{W[\nu]\in\mathcal{W}}$ the indicator variable:

$$\mathbb{1}_{W_{[\nu]}\in\mathcal{W}} = \begin{cases} 1, & \text{if } W_{[\nu]} \in \mathcal{W}, \\ 0, & \text{o.w.}. \end{cases}$$

For a given multi-version code, we construct auxilliary variables $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}$, where $Y_i, Z_j \in [q], i \in [c-1], j \in [\nu]$, $1 \leq A_1 \leq \cdots \leq A_\nu \leq c$, and a permutation $\Pi : [\nu] \to [\nu]$, such that there is a bijection from values of $\mathcal{W}$ to $(Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi)$. In particular, we describe a mapping **AuxVars** from values in $\mathcal{W}$ to $(Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi)$ in Section VI-A, and prove in in Section VI-C that **AuxVars** is bijective.

Consider an arbitrary probability distribution on $W_{[\nu]}$, then the bijection implies that

$$H(Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi \mid W_{[\nu]}, \mathbb{1}_{W[\nu]\in\mathcal{W}} = 1) = H(W_{[\nu]} \mid Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi, \mathbb{1}_{W[\nu]\in\mathcal{W}} = 1) = 0 \tag{12}$$

If we assume a uniform distribution on the elements of $W_{[\nu]}$, then the converse of Theorem 2 follows from the following set of relations.

$$\begin{aligned}
&\log(q^{c+\nu-1}\binom{c+\nu-1}{\nu}\nu!) \\
&\geq H(Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi \mid \mathbb{1}_{W[\nu]\in\mathcal{W}} = 1) \\
&= I(Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi; W_{[\nu]} \mid \mathbb{1}_{W[\nu]\in\mathcal{W}} = 1) \\
&= H(W_{[\nu]} \mid \mathbb{1}_{W[\nu]\in\mathcal{W}} = 1) \\
&= \log|\mathcal{W}| \\
&\geq \log\frac{M^\nu \nu!}{\nu^\nu},
\end{aligned}$$

where the last inequality follows when $M \geq \nu$, and for the first inequality we use the fact that $Y_i, Z_j \in [q]$, there are at most $\nu!$ possibilities for Π, and at most $\binom{c+\nu-1}{\nu}$ possibilities of $A_{[\nu]}$. This implies that

$$\frac{\log q}{\log M} \geq \frac{\nu}{c+\nu-1} - \frac{\nu^\nu \binom{c+\nu-1}{\nu}}{(c+\nu-1)\log M} \tag{13}$$

as required.

To complete the proof, we describe the mapping **AuxVars** in Algorithm 1 in Section VI-A, and show in Section VI-C that **AuxVars** is bijective. Section VI-A describes some useful properties of Algorithm 1.

A. Algorithm Description

The function **AuxVars** which takes as input, an element $W_{[\nu]}$ from $\mathcal{W}$ and returns variables $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}$ is described in Algorithm 1. The algorithm description involves the use of a set valued function χ, which we refer to as the *decodable set function*. We define the function next.

The decodable set function is characterized by the following parameters:

- a positive integer $l \leq c$,
- a subset T of versions, $T \subseteq [\nu]$;

it takes as input,

- l states $S_1, S_2, \ldots, S_l \in \mathcal{P}([\nu])$,
- and messages $W_{[\nu]} \in [M]^{\nu}$,

and outputs a subset of $[M]$. Recall that the decoding function $\psi_{\mathbf{S}}^{[c]}$ returns NULL if there is no common version among the servers $[c]$ in state $\mathbf{S}$. The decodable set function is denoted as $\chi_{l|T}(S_1, \ldots, S_l, W_{[\nu]})$ and defined as

$$\begin{aligned}
&\chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]}) \\
=&\Big\{ \psi_{\mathbf{S}'}^{([c])}(X_1, X_2 \ldots, X_c) : \\
&\mathbf{S}' = (S_1, \ldots, S_l, {S'}_{l+1}, \ldots, {S'}_c), \forall {S'}_j \subseteq T, j \in [l+1, c], \\
&X_m = \varphi_{S_m}(W_{S_m}), 1 \leq m \leq l \\
&X_m = \varphi_{{S'}_m}(W_{S'_m}), l+1 \leq m \leq c \Big\} - \{\text{NULL}\}. \qquad (14)
\end{aligned}$$

To put it in plain words, the decodable set $\chi_{l|T}$ is the set of all non-null values that the decoding function ψ can return, given the states of the first l servers, the message realizations of $W_{[\nu]}$, when the states of the last $c-l$ servers are restricted to be subsets of T. It is instructive to note that $\chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]})$ is a subset of $\{W_i : i \in [\nu]\}$, as stated in the next lemma.

Lemma 1 *For every positive integer $l \in [c]$, set $T \subseteq [\nu]$ and states $S_1, S_2, \ldots, S_l \in \mathcal{P}([\nu])$, we have*

$$\chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]}) \subseteq \{W_i : i \in [\nu]\}.$$

Proof: For every collection of $c-l$ states $S'_{l+1}, S'_{l+2}, \ldots, S'_c \subseteq T$ such that the state

$$\mathbf{S}' = (S_1, S_2, \ldots, S_l, S'_{l+1}, S'_{l+2}, \ldots, S'_c)$$

has a common version, the decoding function $\psi_{\mathbf{S}'}$ returns a message value that was encoded. Since the encoded message is $W_{[\nu]}$, the decoding function returns an element in $\{W_i : i \in [\nu]\}$.

If there is no collection of $c-l$ states $S'_{l+1}, S'_{l+2}, \ldots, S'_c \subseteq T$ such that the state

$$\mathbf{S}' = (S_1, S_2, \ldots, S_l, S'_{l+1}, S'_{l+2}, \ldots, S'_c)$$

has a common version, the decoding function $\psi_{\mathbf{S}'}$ returns NULL. In this case, the decodable set function returns an empty set, which is a subset of $\{W_i : i \in [\nu]\}$. Therefore $\chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]})$ is always a subset of $\{W_i : i \in [\nu]\}$. ■

The following property is useful in our description of Algorithm 1.

Lemma 2 *Consider messages $W_{[\nu]}$ that have unique values, that is, $W_{[\nu]} \in \mathcal{W}$. Then, for any element $W \in \chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]})$, there is a unique positive integer $m \in [\nu]$ such that $W_m = W$.*

Proof: The lemma readily follows from noting that there is a one-to-one correspondence between $[\nu]$ and $W_{[\nu]}$, and that every element W in $\chi_{l|T}$ is also an element in $W_{[\nu]}$ by Lemma 1. ■

The decodable set function has an intuitive interpretation when $W_{[\nu]}$ has unique values, that is, when $W_{[\nu]} \in \mathcal{W}$. If $\chi_{l|T}(S_1, S_2, \ldots, S_l, W_{[\nu]}) - \{W_i : i \in T\}$ is non-empty, then, loosely speaking, this implies that the first l servers contain enough information for at least one message in $[\nu] - T$. This is because the decodable set function restricts the state of the last $c - l$ servers to be from T; as a consequence, if it returns a value corresponding to a version in $[\nu] - T$, then the first l servers must contain sufficient information of this version.

Algorithm 1 Function **AuxVars**: Takes input $W_{[\nu]} \in \mathcal{W}$, and outputs variables $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi$.

1: **AuxVars**($W_{[\nu]}$)
2: Initialize VerCount $\leftarrow 1$
3: Initialize ServCount $\leftarrow 1$
4: Initialize set VersionsEncountered $\leftarrow \{\}$
5: Initialize $Y_j \leftarrow 1, Z_k \leftarrow 1, A_k \leftarrow 1, j \in [c], k \in [\nu]$.
6: **while** VerCount $\leq \nu$ and ServCount $\leq c$ **do**
7: $\quad \mathbf{S}$(ServCount) $\leftarrow [\nu] -$ VersionsEncountered
8: $\quad T \leftarrow [\nu] -$ VersionsEncountered.
9: $\quad U \leftarrow \left\{W_u : W_u \in \chi_{\text{ServCount}|T-\{u\}}\left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(\text{ServCount}), W_{[\nu]}\right), u \in T\right\}$
10: $\quad$ **if** $U \neq \emptyset$ **then**
11: $\quad\quad W \leftarrow \max U$ ▷ Natural ordering on $[M]$ for max
12: $\quad\quad$ Let $v \in [\nu]$ such that $W_v = W$. ▷ From Lemma 2 there exists a unique v
13: $\quad\quad A_{\text{VerCount}} \leftarrow$ ServCount
14: $\quad\quad Z_{\text{VerCount}} \leftarrow \varphi^{(\text{ServCount})}_{\mathbf{S}(\text{ServCount})}(W_{\mathbf{S}(\text{ServCount})})$
15: $\quad\quad \Pi(\text{VerCount}) \leftarrow v$
16: $\quad\quad$ VersionsEncountered $\leftarrow$ VersionsEncountered $\cup \{v\}$
17: $\quad\quad$ VerCount $\leftarrow$ VerCount $+ 1$
18: $\quad$ **else**
19: $\quad\quad Y_{\text{ServCount}} \leftarrow \varphi^{(\text{ServCount})}_{\mathbf{S}(\text{ServCount})}(W_{\mathbf{S}(\text{ServCount})})$
20: $\quad\quad$ ServCount $\leftarrow$ ServCount $+ 1$
21: $\quad$ **end if**
22: **end while**
23: If Π is not a permutation of $[\nu]$, set Π to be an arbitrary permutation. ▷ As a consequence of Lemma 4 and Property (5), this line is never executed.
24: Return $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi$

In Algorithm 1, we describe the function **AuxVars** that takes as input $W_{[\nu]} \in \mathcal{W}$ and returns $(Y_{[c-1]}, A_{[\nu]}, Z_{[\nu]}, \Pi)$. Here, we informally describe the algorithm and examine some properties.

In every iteration of the while loop of Algorithm 1, either VerCount increases by 1, or ServCount increases by 1. In particular, if Line 10 returns true, then VerCount increases by 1, otherwise ServCount increases by 1. Therefore, the while loop terminates, and as a consequence, the algorithm terminates. In our subsequent discussions, we identify an iteration of the while loop by its unique VerCount–ServCount pair at the beginning of the iteration.

Every iteration of the while loop begins by setting the server state $\mathbf{S}$(ServCount) in Line 7. If Line 10 is false, then the iteration sets $Y_{\text{ServCount}}$ in line 19 and then increments ServCount. If Line 10 is true, then the iteration sets $A_{\text{VerCount}}, Z_{\text{VerCount}}$ and Π(VerCount) respectively in Lines 13,14,15, and then increments VerCount. In particular, A_{VerCount} is set to the server index ServCount, and Π(VerCount) is set to the version index VerCount. Note that A_1 is the smallest value of ServCount such that Line 10 returns true, that is, it is the smallest integer such that $\chi_{A_1|[\nu]-\{u\}}([\nu], [\nu], \ldots, [\nu], W_{[\nu]})$

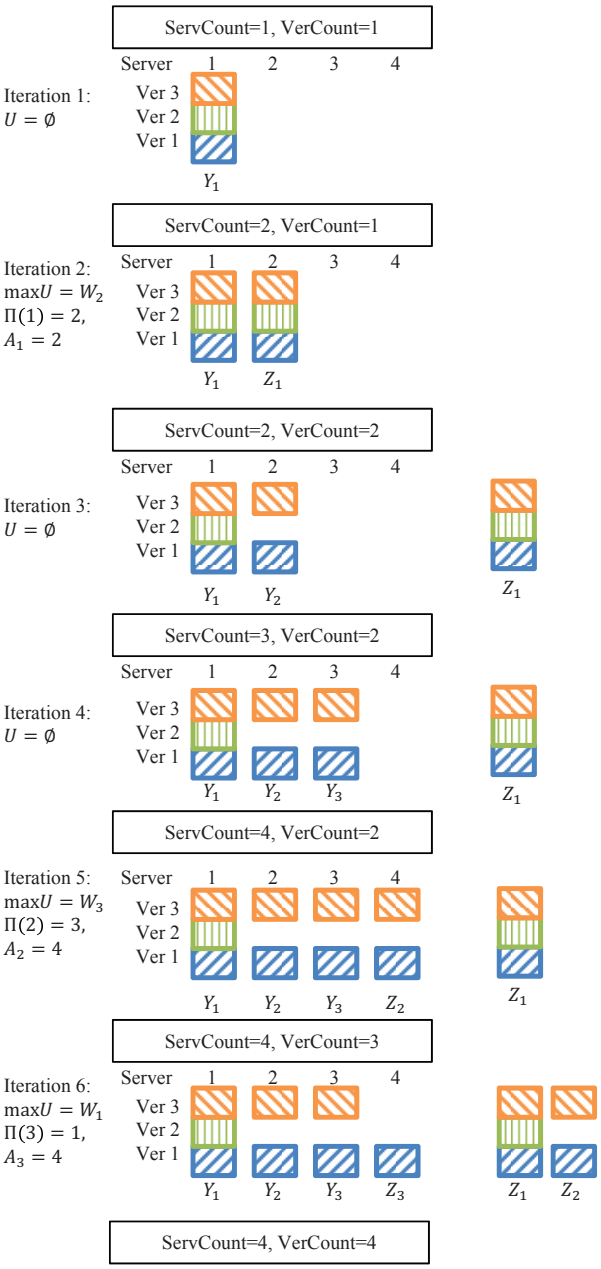

Fig. 3. Example of the algorithm. $c = 4, \nu = 3$. The resulting server indices are $A_{[3]} = (2, 4, 4)$, and the permutation on the versions is $\Pi = (2, 3, 1)$.

contains W_u for some $u \in [\nu]$. Intuitively speaking, A_1 is the smallest integer such that the first A_1 servers have enough information about some version $v \in [\nu]$, when the states of these servers are all set to $[\nu]$. If more than one version in $[\nu]$ returns true for the iteration with ServCount $= A_1$, then v is picked to be the version index corresponding to the maximum value of $\{W_u : \chi_{A_1|[\nu]-\{u\}}([\nu], [\nu], \ldots, [\nu], W_{[\nu]})$ contains $W_u\}$. The iteration sets Π(VerCount) to the version index v.

In Figure 3, we show an example of a possible execution of the algorithm for $\nu = 3, c = 4$ that happens to halt at ServCount $A_\nu = 4$ for a particular multi-version code and message tuple $W_{[3]}$. The states of the servers are set one by one to $\{1, 2, 3\}$ as in Line 7. The algorithm proceeds incrementing

ServCount in every iteration where Line 10 returns false. In an iteration where Line 10 returns true, VerCount is incremented. Suppose at ServCount $= 2$, Line 10 returns true for the first time in the execution, and suppose that $v = 2$ in Line 12; the algorithm sets $\Pi(1) = 2, A_1 = \text{ServCount} = 2$, and, in the next iteration, the state of the second server is reset to $[\nu] - \{\Pi(1)\} = \{1, 3\}$. Then the algorithm proceeds incrementing ServCount every time Line 10 returns false, setting the state of the corresponding server to $\{1, 3\}$. Now, suppose that Line 10 returns true at ServCount $= 4$, and that $v = 3$ in Line 12. Then $\Pi(2) = \text{VerCount} = 3, A_2 = \text{ServCount} = 4$. In the next iteration, the state of server 4 is set to $\{1\}$. Then A_3 is set similarly.

We later show that from the variables $Y_{[3]}, Z_{[3]}, A_{[3]}, \Pi$, one can recover all of the 3 versions $W_{[3]}$. Here we provide an informal overview of the argument. From Iteration 6 in Fig. 3, we can observe that W_1 is equal to $\psi^{[4]}(Y_1, Y_2, Y_3, Z_3)$, where the decoding function $\psi^{[4]}$ is evaluated with states $S_1 = \{1, 2, 3\}, S_2 = S_3 = \{1, 3\}, S_4 = \{1\}$. The states $S_1, \ldots, S_4$ can be inferred from $A_{[3]}$ and Π. Similarly, from Iteration 5 in Fig. 3, we can observe that W_3 is equal to $\psi(Y_1, Y_2, Y_3, Z_2)$ with states $S_1 = \{1, 2, 3\}, S_2 = S_3 = \{1, 3\}, S_4 = \{1, 3\}$. In the converse proof, we will show that, given W_1, W_3, the value W_2 can be recovered by using the conditional decoding function as the maximum value of the set $\chi_{2|\{2,3\}}(\{1, 2, 3\}, \{1, 2, 3\}, W_1) - \{W_1, W_3\}$, which can be evaluated using the values W_1, W_3, Y_1, Y_2, Z_1 as

$$\Bigg\{ \psi_{\mathbf{S}'}^{([4])}(X_1, X_2, X_3, X_4) : \\ \mathbf{S}' = (\{1, 2, 3\}, \{1, 2, 3\}, S'_3, S'_4), \forall S'_j \subseteq \{1, 3\}, j \in [3, 4], \\ X_1 = Y_1, X_2 = Z_1, \\ X_m = \varphi_{S'_m}^{(m)}(W_{S'_m}), m = 3, 4 \Bigg\} - \{W_1, W_3, \text{NULL}\}.$$

In particular, we will show that the above set is a singleton set, with the element being W_2.

B. Properties of Algorithm 1

We next list some useful and instructive properties of Algorithm 1 before proceeding to formally prove that **ConverseAuxilliaryVars** is invertible.

Property (1) If the last iteration of the while loop begins with VerCount $= \nu$, the server indices satisfy $1 \le A_1 \le \cdots \le A_\nu \le c$. Moreover, for any $t \le \nu$, every iteration of the while loop with ServCount $= A_t - 1$ has VerCount $\le t$. Later, in Lemma 4, we show that in every execution of Algorithm 1, the last iteration of the while loop indeed begins with VerCount $= \nu$.

Property (2) For the iteration of the while loop with VerCount and ServCount, the server states are set by the algorithm to be

$$\mathbf{S}(i) = \begin{cases} [\nu], & i \in [1, A_1 - 1], A_1 > 1, \\ [\nu] - \{\Pi(x) : x \in [m]\}, & i \in [A_m, A_{m+1} - 1], m \in [\text{VerCount} - 2], A_{m+1} > A_m, \\ [\nu] - \{\Pi(x) : x \in [\text{VerCount} - 1]\}, & i \in [A_{\text{VerCount}-1}, \text{ServCount}]. \end{cases} \tag{15}$$

Property (3) For the iteration of the while loop with VerCount $= j$, $j \ge 2$, and ServCount $= k$, let $\mathbf{S}(i)$ be the state of Server i, $i \in [k]$. Let $t < j$. Consider the last iteration with ServCount $= A_t - 1$. Suppose that, in this iteration, VerCount $= x$, and $\hat{\mathbf{S}}(i)$ is the state of Server i, $i \in [A_t - 1]$. Then for $i \in [A_t - 1]$, the states $\mathbf{S}(i)$ are the same as the sates $\hat{\mathbf{S}}(i)$.

Property (4) At the beginning of an iteration of the while loop, the set VersionsEncountered is $\{\Pi(1), \ldots, \Pi(\text{VerCount-1})\}$. The set T indicates the set of versions not encountered, which is $[\nu] - \{\Pi(1), \ldots, \Pi(\text{VerCount-1})\}$.

Property (5) In Line 9, $U \subseteq \{W_i, i \in T\}$, where $T = [\nu] -$ VersionsEncountered. Therefore, when Line 10 returns true, $U \cap \{W_i : i \in \text{VersionsEncountered}\} = \emptyset$. As a consequence, $\Pi(\text{VerCount}) \notin \{\Pi(1), \ldots, \Pi(\text{VerCount} - 1)\}$. Therefore, if the last iteration of the while loop begins with $\text{VerCount} = \nu$ and Line 10 returns true in this iteration, then Π is indeed a permutation at the end of the iteration.

Property (6) Consider the last iteration of the while loop of the algorithm where Line 10 returns true. Suppose this iteration begins with $\text{VerCount} = j$, $\text{ServCount} = A_j$. Then we have for all $i \in [A_j - 1]$,

$$Y_i = \varphi^{(i)}_{\mathbf{S}(i)}(W_{\mathbf{S}(i)}),$$

where $\mathbf{S}(i)$ is specified in Property (2) with $\text{VerCount} = j$, $\text{ServCount} = A_j$. Moreover, for all $i \in [j]$,

$$Z_i = \varphi^{(A_i)}_{\mathbf{S}'(A_i)}(W_{\mathbf{S}'(A_i)}),$$

where $\mathbf{S}'(A_i) = \{\Pi(i), \ldots, \Pi(\nu)\}$.

Property (7) Note that when Line 10 returns true, even though $W_{[\nu]}$ is an input to the function $\chi_{\text{ServCount}|T-\{u\}}$ in Line 9, for every $u \in T$, we can generate the output of the function only from $\mathbf{S}(1), \ldots,$ $\mathbf{S}(\text{ServCount})$, $\{W_i : i \in T - \{u\}\}$, $Y_m = \varphi_{\mathbf{S}(m)}(W_{\mathbf{S}(m)}), 1 \leq m \leq \text{ServCount} - 1$, and $Z_{\text{ServCount}} = \varphi_{\mathbf{S}(\text{ServCount})}(W_{\mathbf{S}(\text{ServCount})})$. In particular,

$$\begin{aligned}
&\chi_{\text{ServCount}|T-\{u\}}\left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(\text{ServCount}), W_{[\nu]}\right) \\
&= \Bigg\{ \psi^{([c])}_{\mathbf{S}'}(X_1, X_2 \ldots, X_c) \neq \text{NULL} : \\
&\mathbf{S}' = (\mathbf{S}(1), \ldots, \mathbf{S}(\text{ServCount}), S'_{l+1}, \ldots, S'_c), \forall S'_j \subseteq T - \{u\}, j \in [l+1, c], \\
&X_m = Y_m, 1 \leq m \leq \text{ServCount} - 1 \\
&X_m = Z_m, m = \text{ServCount} \\
&X_m = \varphi^{(m)}_{S'_m}(W_{S'_m}), \text{ServCount} + 1 \leq m \leq c \Bigg\}. \qquad (16)
\end{aligned}$$

We use this property in showing that **AuxVars** is one-to-one.

We next state Lemmas 3 and 4. Statement (i) of Lemma 3 is useful in proving Lemma 4. Statement (ii) of Lemma 3 is useful in the proof of Theorem 2, in particular, in inverting **AuxVars** to obtain $W_{[\nu]}$ from $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi$. Lemma 4 shows that at the beginning of the last iteration in the algorithm, the variable VerCount is equal to ν. This means that all ν versions returns true at Line 10 in some iteration of the while loop.

Lemma 3 *(i) Consider any execution of* ***AuxVars*** *and consider an iteration of the while loop. After Line* 8 *is executed in the while loop, the following statement is true for any* $u \in T$*:*

$$\chi_{\textit{ServCount}|T-\{u\}}\left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(\textit{ServCount}), W_{[\nu]}\right) \subseteq \{W_i, i \in T\}. \qquad (17)$$

where ServCount represents the value at the beginning of the while loop iteration.

(ii) Consider any execution of ***AuxVars*** *where the final iteration of the while loop begins with VerCount* $= k$ *. For any* $t \in [k]$*, we have*

$$\{W_{\Pi(t)}\} = \chi_{A_t|T-\Pi(t)}(\mathbf{S}(1), \ldots \mathbf{S}(A_t), W_{[\nu]}) - \{W_i : i \in T - \{\Pi(t)\}\},$$

where, $T = \{\Pi(t), \Pi(t+1), \ldots, \Pi(\nu)\}$*, and* $\mathbf{S}$ *is defined as* Property (2) *with VerCount* $= t$*, ServCount* $= A_t$*.*

Proof: (i) Note that $T = [\nu] -$ VersionsEncountered, and VersionsEncountered $= \{\Pi(1), \Pi(2), \ldots, \Pi(\text{VerCount} - 1)\}$ by Property (4). Notice that when VerCount $= 1$, the claim is satisfied automatically because $T = [\nu]$.

We prove by contradiction. We suppose at ServCount $= k$, $k \in [c]$, and VerCount $= j$, $j \in [2, \nu]$, equation (17) is violated, and

$$W_{\Pi(t)} \in \chi_{k|T-\{u\}} \left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(j), W_{[\nu]}\right), \tag{18}$$

for some $t \in [j-1]$. Let $\mathbf{S}$ be the state vector of length ServCount specified in Property (2), and $T - \{u\} = [\nu] - \{\Pi(1), \ldots, \Pi(j-1)\} - \{u\}$. By the definition of decodable set function, there exists a state $\mathbf{S}'$ that decodes $\Pi(t)$ from the first c servers,

$$\psi_{\mathbf{S}'}^{([c])}(\varphi_{\mathbf{S}'(1)}^{(1)}(W_{[\nu]}), \varphi_{\mathbf{S}'(2)}^{(2)}(W_{[\nu]}), \ldots, \varphi_{\mathbf{S}'(c)}^{(c)}(W_{[\nu]})) = W_{\Pi(t)}, \tag{19}$$

such that

$$\mathbf{S}'(i) \begin{cases} = \mathbf{S}(i), & i \in [k], \\ \subseteq [\nu] - \{\Pi(1), \ldots, \Pi(j-1)\} - \{u\}, & i \in [k+1, c]. \end{cases} \tag{20}$$

If $A_t = 1$, then by Property (2), $\mathbf{S}'(i) \subseteq [\nu] - \{\Pi(1), \ldots, \Pi(t)\}$ for all $i \in [c]$. By Remark 1 in Section II, we know $\psi_{\mathbf{S}'}^{([c])}$ should return a value corresponding to a version in $\cup_{i\in[c]}\mathbf{S}'(i) \subseteq [\nu] - \{\Pi(1), \ldots, \Pi(t)\}$, which contradicts (19). So we assume $A_t > 1$.

Consider the last iteration with ServCount $= A_t - 1$. Let VerCount $= x$ in this iteration. By Property (1), we know that $x \leq t$. We will show that if (18) holds, then, in this iteration, Line 10 returns true. Therefore, in this iteration, VerCount $\leftarrow x + 1$ and ServCount $= A_t - 1$ remains unchanged. This contradicts our assumption that the last iteration with ServCount $= A_t - 1$ has VerCount $= x$. So, to complete the proof, it suffices to show that Line 10 returns true in this iteration. We show this next.

For the iteration of the while loop with ServCount $= A_t - 1$ and VerCount $= x$,

- let $\hat{\mathbf{S}}$ be the server states as in Property (2),
- let $\hat{T} = [\nu] - \{\Pi(1), \ldots, \Pi(x-1)\} = \{\Pi(x), \ldots, \Pi(\nu)\}$ be the set in Line 8,
- let $\hat{U} = \left\{W_u : W_u \in \chi_{\text{ServCount}|\hat{T}-\{u\}} \left(\hat{\mathbf{S}}(1), \hat{\mathbf{S}}(2), \ldots, \hat{\mathbf{S}}(\text{ServCount}), W_{[\nu]}\right), u \in \hat{T}\right\}$ be the set in Line 9.

Here note that $\Pi(t) \in \hat{T}$ because $x \leq t$ by Property (1). By Property (2) and Property (3), and (20), we note that

$$\mathbf{S}'(i) \begin{cases} = \mathbf{S}(i) = \hat{\mathbf{S}}(i), & i \in [A_t - 1], \\ \subseteq \hat{T} - \Pi(t), & i \in [A_t, c]. \end{cases} \tag{21}$$

Combining (21) and (19), we know that $\Pi(t)$ is in the decodable set function at VerCount $= t$ and ServCount $= A_t - 1$ using the state $\mathbf{S}'$, that is,

$$W_{\Pi(t)} \in \chi_{A_t-1|\hat{T}-\{\Pi(t)\}} \left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(A_t - 1), W_{[\nu]}\right),$$

which combined with the fact that $\Pi(t) \in \hat{T}$ implies $\hat{U} \neq \emptyset$ and Line 10 returns true. Thus VerCount $\leftarrow x + 1$ and ServCount $= A_t - 1$ should stay unchanged. Hence we get a contradiction.

(ii) Consider the iteration when $\Pi($VerCount$)$ is set in Line 15, which has VerCount $= t$, ServCount $= A_t$. After Line 8 is executed, we have $T = \{\Pi(t), \Pi(t+1), \ldots, \Pi(\nu)\}$. We know Line 10 returns true, and by Line 11, $W_{\Pi(t} = \max U$. Thus letting $u = \Pi(t) \in T$, we have

$$W_u \in \chi_{A_t|T-\{u\}} \left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(A_t), W_{[\nu]}\right).$$

Moreover, $W_u \notin \{W_i : i \in T - \{u\}\}$. Combined with statement (i), we obtain the desired statement. ■

Lemma 4 *In any execution of Algorithm* 1*, at the beginning of the final iteration of the while loop, we have VerCount*$= \nu$*, and Line* 10 *returns true in that iteration.*

Proof: We prove the lemma by contradiction. Suppose that, in an execution of Algorithm 1, the last iteration begins with $\mathsf{VerCount} = j$, for some $j < \nu$. This means we have found a set of versions in the set $\mathsf{VersionsEncountered}$ such that Line 10 is not satisfied for $\mathsf{VerCount} = j$. Furthermore, knowing $\mathsf{VersionsEncountered} = \{\Pi(1), \ldots, \Pi(j-1)\}$, we note that Line 10 is satisfied for version $\Pi(i)$, when $\mathsf{ServCount}$ is equal to A_i for $i \in [j-1]$.

Consider the last iteration of the while loop, $\mathsf{VerCount} = j, \mathsf{ServCount} = c$. The states of the first c servers are shown in Property (2). Note that there exists a latest common version among these c server states, which is $\max([\nu] - \{\Pi(1), \ldots, \Pi(j-1)\})$. Therefore, the decoding function $\psi_{\mathbf{S}}^{([c])}$ returns a non-null value. Specifically, for some $u \in [\nu]$, we have

$$\psi_{\mathbf{S}}^{([c])}(\varphi_{\mathbf{S}(1)}^{(1)}(W_{[\nu]}), \varphi_{\mathbf{S}(2)}^{(2)}(W_{[\nu]}), \ldots, \varphi_{\mathbf{S}(c)}^{(c)}(W_{[\nu]})) = W_u.$$

Furthermore, because $\mathsf{ServCount} = c$, we have

$$\begin{aligned}&\chi_{\mathsf{ServCount}|T-\{u\}}\left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(\mathsf{ServCount}), W_{[\nu]}\right)\\ &= \{\psi_{\mathbf{S}}(\varphi_{\mathbf{S}(1)}^{(1)}(W_{[\nu]}), \varphi_{\mathbf{S}(2)}^{(2)}(W_{[\nu]}), \ldots, \varphi_{\mathbf{S}(c)}^{(c)}(W_{[\nu]}))\} = \{W_u\}\end{aligned}$$

The above result combined with statement (i) of Lemma 3 implies that the set $U \neq \emptyset$ and Line 10 is satisfied, which contradicts our assumption. This completes the proof. ■

C. Proof of Theorem 2

Now we are ready to prove the lower bound in Theorem 2.

Proof of Theorem 2 for general ν: Suppose there is a (c, c, ν, M, q) multi-version code. Run Algorithm 1 on every ν-tuple distinct version values $W_{[\nu]}$. By Lemma 4, we know the algorithm terminates with $\mathsf{VerCount} = \nu$ and $1 \leq A_1 \leq \cdots \leq A_\nu \leq c$. We use a dummy variable $A_0 = 1$. First, since the algorithm is deterministic, we know there is a mapping $\mathbf{AuxVars}$ from $W_{[\nu]} \in \mathcal{W}$ to $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi$. Next, we show that $\mathbf{AuxVars}$ is a one-to-one mapping, that is, we create a mapping from $Y_{[c-1]}, Z_{[\nu]}, A_{[\nu]}, \Pi$ to $W_{[\nu]} \in \mathcal{W}$.

We now describe how to obtain $W_{[\nu]} \in \mathcal{W}$ from the output of $\mathbf{AuxVars}$. In particular, for any $t \in \{1, 2, \ldots, \nu\}$, we describe a procedure to obtain $W_{\Pi(t)}$ from $Y_1, Y_2, \ldots, Y_{A_t}$, Z_t and $W_{\Pi(t+1)}, W_{\Pi(t+2)}, \ldots, W_{\Pi(\nu)}$. The procedure automatically implies that we can obtain $W_{[\nu]}$ from $Y_{[c-1]}, A_{[\nu]}, Z_{[\nu]}$ and Π. For any realization of distinct values $W_{[\nu]} \in \mathcal{W}$, if we are given $\Pi, A_{[\nu]}$, we can set the state to be

$$\mathbf{S}(i) = \begin{cases}[\nu] - \{\Pi(1), \ldots, \Pi(j-1)\}, & i \in [A_{j-1}, A_j - 1], \forall j \in [t],\\ [\nu] - \{\Pi(1), \ldots, \Pi(t-1)\}, & i = A_t.\end{cases}$$

By Property (2), the above states are the same as the states in Algorithm 1 in iteration of the while loop with $\mathsf{VerCount} = t$, $\mathsf{ServCount} = A_t$. Note that at that iteration, by Property (6) we know $Y_{[A_t-1]}, Z_t$ are the values of Servers $[A_t]$, which corresponds to the above states $\mathbf{S}(1), \ldots, \mathbf{S}(A_t)$. That is $Y_i = \varphi_{\mathbf{S}(i)}(W_{[\nu]})$, for $i \in [A_t - 1]$ and $Z_t = \varphi_{\mathbf{S}(A_t)}(W_{[\nu]})$. Let $T = [\nu] - \{\Pi(1), \ldots, \Pi(t-1)\}$. Thus, $W_{T-\{\Pi(t)\}} = W_{\{\Pi(t+1), \ldots, \Pi(\nu)\}}$. From Lemma 3 (ii), we know that

$$\{W_{\Pi(t)}\} = \chi_{A_t|T-\{\Pi(t)\}}\left(\mathbf{S}(1), \mathbf{S}(2), \ldots, \mathbf{S}(A_t), W_{[\nu]}\right) - \{W_i, i \in T - \Pi(t)\}.$$

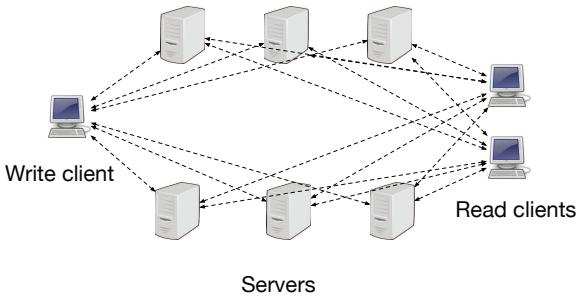

Fig. 4. System architecture of a toy model with one write client and many read clients.

Therefore, to obtain $W_{\Pi(t)}$, it suffices to evaluate the set

$$\chi_{A_t|T-\{\Pi(t)\}}\left(\mathbf{S}(1),\mathbf{S}(2),\ldots,\mathbf{S}(A_t),W_{[\nu]}\right)-\{W_i,i\in T-\Pi(t)\}.$$

Property (7) states that the above set can be computed using $Y_{[A_t-1]},Z_t,W_{T-\{\Pi(t)\}}$ via equation (16). Therefore, we can compute $W_{\Pi(t)}$ as

$$\begin{aligned}
&\{W_{\Pi(t)}\}=\\
&=\Bigg\{\psi^{([c])}_{\mathbf{S}'}(X_1,X_2\ldots,X_c)\neq \text{NULL}:\\
&\mathbf{S}'=(\mathbf{S}(1),\ldots,\mathbf{S}(A_t),{S'}_{A_t+1},\ldots,{S'}_c),\forall {S'}_j\subseteq T-\{\Pi(t)\},j\in[l+1,c],\\
&X_m=Y_m,1\leq m\leq A_t-1\\
&X_m=Z_t,m=A_t,\\
&X_m=\varphi_{{S'}_m}(W_{S'_m}),A_t+1\leq m\leq c\Bigg\}\\
&-\{W_i:i\in T-\{\Pi(t)\}\}.
\end{aligned}$$

Therefore, given $W_{\{\Pi(t+1),\ldots,\Pi(\nu)\}},Y_{[A_t-1]},Z_t,A_{[\nu]},\Pi$, we see that the value of $W_{\Pi(t)}$ is determined. Noting that $A_i\leq c$ for all $i\in[\nu]$ by Lemma 4, we infer that, given $Y_{[c-1]},Z_{[\nu]},A_{[\nu]},\Pi$, we can determine values of $W_{\Pi(\nu)},W_{\Pi(\nu-1)},\ldots,W_{\Pi(1)}$ one by one. Hence we have a mapping from $Y_{[c-1]},Z_{[\nu]},A_{[\nu]},\Pi$ to $W_{[\nu]}\in\mathcal{W}$, implying that **AuxVars** is one-to-one. Then we know (12) and hence (13) are satisfied, thus the theorem is proved. ■

Remark 4 *If $W_{[\nu]}$ is not uniformly distributed, then the proof of Theorem* 2 *can be appropriately modified to obtain a lower bound on the storage size per server using* (13)*:*

$$\log q\geq\frac{H(W_{[\nu]}|\mathbb{1}_{W_{[\nu]}\in\mathcal{W}}=1)-\log(\nu!\binom{c+\nu-1}{\nu})}{c+\nu}.$$

The above bound can be much smaller compared to the one in Theorem 2*, if, for example, the versions are dependent and have some structure. In particular, the storage cost lower bound would be much smaller if, because of the dependency of the versions, $H(W_{[\nu]}|\mathbb{1}_{W_{[\nu]}\in\mathcal{W}=1}))<<\log|\mathcal{W}|$. In this case, it is an open problem to study codes that exploit the dependency amongst versions to obtain a smaller storage cost.*

VII. Toy Model of Distributed Storage

The multi-version coding problem has no temporal aspect in its formulation. Here, we study a toy model of storage system that evlolves over time and demonstrate the potential of multi-version codes for an asynchronous setting. In particular, we explicitly describe an arrival model for new

versions and channel models for the links between the encoders, servers and decoders. Our study of the toy model establishes a physical interpretation for the parameter ν of multi-version coding. In particular, our toy model demonstrates the connection between the parameter ν and the degree of asynchrony in a storage system. We begin with a description of our model.

Consider a distributed storage system with N servers, a write client that generantes different versions of the message, and read clients that aim to to read the message versions (See Fig. 4). The write client aims to store the new version of the message in a distributed storage system consisting of the servers. We aim to design server storage strategies that implement a consistent distributed storage system, and are tolerant to f server failures. We now describe toy models for the channels between the clients and the servers, and the arrival model at the clients.

Message arrival model: In the toy model here, we assume that a new version of the message appears at the write client in every time-slot. The message at time slot t is denoted as W_t, $t \in \mathbb{N}^+$, where $W_t \in [M]$. At time slot t, the write client sends a packet containing time stamp and the message, that is a packet with (t, W_t), to every server.

Channel Model: We describe two models for the channels between the clients and the servers. The first model is a delay based model, and the second model is an erasure based model.

Delay model: We assume that a transmitted packet sent by the write client at time slot t to a server arrives at the server in any one of the time slots $\{t, t+1, \ldots, t+T-1\}$. In other words, the sent message has a delay that can be any one of $0, 1, 2, \ldots, T-1$. The delay is not known a priori, and can be different for different packets. It is useful to note that even if the same packet is sent in the same time slot to different servers, the delay can be different for different packets.

Note that in the message arrival model, for every time slot t, there is a message (t, W_t) sent to every server. Let $\mathcal{S}_m^{(t)} \subset \{1, 2, \ldots, t\}$ denote the set of versions received by server m at time t. Then in the delay model

$$\mathcal{S}_m^{(t)} = \left(\{t-T+1\} \cup R_m^{(t)}\right) - \bigcup_{j=1}^{t-1} \mathcal{S}_m^{(j)}, \tag{22}$$

where $R_m^{(t)}$ is some arbitrary subset of $\{t-T+2, \ldots, t\}$. The arrival time set $\mathcal{S}_m^{(t)}$ at the servers are not known apriori. Our model is adversarial, that is, we want our decoding constraints (specified below) to be satisfied for all possible arrival time sets $\mathcal{S}_m^{(t)}$ that are of the form (22).

Erasure Model: In the erasure model for the channel, a packet sent by the write client to the server may be erased. There is no packet delay in the erasure model. Our erasure model is adversarial, with the following packet delivery guarantee: for every subset of $N-f$ servers, for any consecutive T packets, there is at least one least one packet such that it arrives at all the $N-f$ servers. Mathematically, the received packet versions $\mathcal{S}_m^{(t)}$ at server m at time t is

$$\mathcal{S}_m^{(t)} = \begin{cases} \{t\} & \text{if } \exists n_1, n_2, \ldots, n_{N-f-1} \in [n] \text{ s.t. } \bigcup_{j=t-T+1}^{t-1} \mathcal{S}_m^{(j)} \cap \bigcap_{k=1}^{N-f-1} \mathcal{S}_{n_k}^{(j)} = \phi \\ \{t\} \text{ or } \phi & \text{otherwise} \end{cases} \tag{23}$$

Encoding requirements: At time t, for every $m \in [N]$, server m stores a symbol $X_m^{(t)}$ which is a function of the stored symbol $X_m^{(t-1)}$ and the received packets $W_{\mathcal{S}_m^{(t)}}$. We assume that $X_m^{(t)} \in [q]$ for every value of t, m. As usual, for a given encoding scheme, we measure its storage cost as $\frac{\log q}{\log M}$.

Decoding requirements: We intend to design a failure tolerance of f servers. In our decoding requirement, a read client accesses any subset of $N-f$ servers and requires to decode the latest common message version among the $N-f$ servers, or the message corresponding to a later version. Our model is adversarial, that is, in the delay model, we want the read client decode the latest common verison for every possible packet arrival pattern at the servers that satisfies (22). Similarly,

in the erasure model, we intend the read client to decode the latest common version for every possible packet arrival pattern that satisfies (23). In our toy model we assume that the link between the servers and read clients are perfect, that is there is no erasure or delay for the packets between the servers and the read client.

It is instructive to note that both the erasure model and delay model ensure that there is at time t, there is a latest common version for all the servers among the versions $\{t, t-1, \ldots, t-T+1\}$. Therefore, under our decoding requirements, a read client that aims to read from the storage system at time slot t must decode a message in $\{W_{t-T+1}, W_{t-T+2}, \ldots, W_t\}$.

We will see that the multi-version coding problem can be used to reveal the fundamental storage cost performance of this setting. In particular, our achievability and converse results of an (n, c, ν) multi-version code provides insights on the storage cost in our setting, where $n = N, c = N - f$ and $T = \nu$.

Claim 2 *Consider a setting where very new message version at the write client takes values from the set $[M]$. If $T|(N-f-1)$, then a storage cost of*

$$\log q = \frac{T}{T+N-f-1} \log M + o(\log M)$$

is achievable.

Proof: The proof is the same for both the erasure model and the delay model. Observe that, in both models, at time t, there is a latest common version among all the servers in $[t-T+1, t]$. As per Construction 2 for an $(N, T, N-f)$ multi-version code, each server stores $\frac{\log M}{T+N-f-1}$ bits of the latest version it has received. Along the same lines as the proof of Theorem 4, we infer that any read client that connects to $N-f$ servers at time t gets at least $\lceil \frac{N-f}{T} \rceil = \frac{N-f+T-1}{T}$ codeword symbols corresponding to at least one version ν^*, where $\nu^* \in [t-T+1, t]$ is the latest common version or a later version among the $N-f$ servers. Therefore, version ν^* is decodable by the read client which reads at time t. There is a a storage overhead of $o(\log_2 M)$ bits since the servers need to store the *time stamp*[2] of the version along with the codeword symbol. ■

Claim 3 *Consider a setting where very new message version at the write client takes values from the set $[M]$. The storage cost of any server storage strategy for the delay model satisfies*

$$\frac{\log q}{\log M} \geq \frac{T-1}{N-f+T-2} - \frac{\log((T-1)^{T-1}\binom{N-f+T-2}{T-1})}{(N-f+T-2)\log M}.$$

Claim 4 *Consider a setting where very new message version at the write client takes values from the set $[M]$. The storage cost of any server storage strategy for the erasure model satisfies*

$$\frac{\log q}{\log M} \geq \frac{T}{N-f+T-1} - \frac{\log(T^T\binom{N-f+T-1}{T})}{(N-f+T-1)\log M}.$$

Claims 3 and 4 are essentially corollaries to Theorem 2. In particular, for both the delay model and the erasure model, we note that the server encoding functions necessarily implements a multi-version code. We provide brief sketches of their proofs here.

[2]In fact, a server that stores a codeword sybol corresponding to message W_t can store the time stamp as $t \mod 2T$. We omit mechanical details of the proof here.

Proof of Claim 3: Consider an arbitrary collection of subset $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \subseteq [t-T+2, t]$. Assume that at time $t-T$, all the packets from the write client to the server are delivered. At times $t-T+1, \ldots, t-1$, no packet is delivered by the channel. At time t, server m receives packets $\mathcal{A}_m \cup \{t-T+1\}$. Given the versions $1, 2, \ldots, t-T+1$, the server encoding functions at time t form a multi-version code over the versions in $[t-T+2, t]$. Since we started with an arbitrary collection of subsets $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, the worst storage cost over all collections of subsets is lower bounded by the cost described in Theorem 2 for $\nu = T, c = N - f$. This completes the proof. ■

Remark 5 *The worst case states described the converse of Theorem 2, when applied to the delay model, may implicitly require the packets sent to be delivered out of order. This is because, when the converse of Theorem 2 is applied in our proof of Claim 3, we may require a server m to contain a version t_1 but not contain a version t_2, where $t_2 < t_1, t_2, t_1 \in [t-T+2, t]$. Our converse for the delay model is therefore more relevant to applications where the versions may be sent in one order, and received in another. The assumption that the order of packets may change is the basis of certain transport protocols such as Stream Control Transmission Protocol (SCTP) [24].*

Proof of Claim 4: Consider an arbitrary collection of subset $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \subseteq [t-T+1, t]$ such that, for every collection of c subsets in $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, there is a common version. Assume that at time t_0 in $[t-T+1, t]$, server $m \in [n]$ receives the packet sent by the write client if and only if $t_0 \in \mathcal{A}_m$. Given the messages $W_{[t-T]}$, the server encoding strategy at times $[t-T+1 : t]$ forms a multi-version code. Since we started with an arbitrary collection of subsets $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, the worst storage cost over all collections of subsets is lower bounded by the cost described in Theorem 2 for $\nu = T, c = N - f$. This completes the proof. ■

The parameter ν is analogous to the parameter T in both the erasure and the delay models. The parameter T is, intuitively speaking, a measure of the degree of asynchrony in the system. Our toy models therefore establishes an explicit connection between the parameter ν and the degree of asynchrony in the storage system. A multi-version code with a larger value for the parameter ν can tolerate a greater degree of asynchrony, albeit at a larger storage cost.

VIII. Concluding Remarks

In this paper, we have proposed the multi-version coding problem, where the goal is to encode various versions in a distributed storage system so that the latest version is decodable. We have given a lower bound on the worst-case storage cost and provide a simple coding scheme that is essentially optimal for an infinite family of parameters. Our problem formulation and solution is a step towards the study of consistent key value stores from an information theoretic perspective. The multi-version coding problem affords a number of interesting generalizations which are relevant to practical consistent distributed storage systems. We discuss some of these generalizations next.

- A useful direction of future work is to study the problem beyond a worst-case setting, for instance, through analysing a restricted set of states. For example, one can assume that the servers always get consecutive versions: $\mathbf{S}(i) = [x, y]$, for some $1 \leq x \leq y \leq \nu$. One can similarly assume that due to network constraints, certain versions only are dispersed to a subset of the servers, namely, $x \notin \mathbf{S}(i)$ for all $i \in I$, where $I \subseteq [n]$ is some subset of server indices. More generally, our problem could be formulated in terms of storage cost per server per state, and the overall storage cost can be optimized based on the workload distributions of the servers.
- Our problem formulation assumes that the number of versions ν, is known a priori. An interesting direction of future work is to manipulate our problem formulation and solutions to incorporate a setting where this parameter is not known.
- Our problem formulation essentially views different versions as being independent. However, in several applications, it is conceivable that different versions are correlated. For dependent

versions, Remark 4 suggests that the converse of Theorem 2 would be applicable after appropriate manipulations. Developing code constructions that exploit dependency in the versions is an interesting area of future work. The ideas of [14], [16] can be useful in this endeavor.

- The framework of our toy model can be developed to study more realistic scenarios. The first step would be to incorporate asynchrony/erasures in the read client. The end goal of the framework would be to understand costs in realistic storage systems, or over models studied in distributed algorithms literature. The standard model in distributed algorithms can be viewed as the delay model of Section VII in the limiting case of asymptotically large T. Furthermore, in distributed computing theory, write and read clients, and servers are modeled as automota (more precisely, input-output automata [1], [25]), the goal is to design client and server protocols that ensure consistency. Developments of our toy model, and appropriate refinements to multi-version coding, can potentially provide information theoretic insights into the storage cost of such systems.
- Minimizing communication costs and latency are important requirements of modern consistent storage services. Refinements of multi-version coding, and our toy model for channels to incorporate these requirements is an important direction of future work. In particular, the tools used in references [14], [15], [16], [17], when appropriately adopted to multi-version coding, may help reduce latency by reducing the amount of information transmitted to disperse and update information related to a new version.

Acknowledgment

The authors would like to thank Prof. Nancy Lynch, Prof. Muriel Médard, and Prof. Tsachy Weissman for their valuable advice and helpful comments.

This work is partially supported by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

[1] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
[2] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.
[3] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, 2008.
[4] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995.
[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, vol. 7, 2007, pp. 205–220.
[6] E. Hewitt, *Cassandra: the definitive guide*. " O'Reilly Media, Inc.", 2010.
[7] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.
[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.
[10] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead byzantine fault-tolerant storage," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 73–86, 2007.
[11] P. Dutta, R. Guerraoui, and R. R. Levy, "Optimistic erasure-coded distributed storage," in *Distributed Computing*. Springer, 2008, pp. 182–196.
[12] V. R. Cadambe, N. Lynch, M. Medard, and P. Musial, "A coded shared atomic memory algorithm for message passing architectures," in *2014 IEEE 13th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2014, pp. 253–260, extended version available at http://arxiv.org/abs/1407.4167.
[13] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, "PoWerStore: proofs of writing for efficient and robust storage," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 285–298.
[14] A. Mazumdar, G. W. Wornell, and V. Chandar, "Update efficient codes for error correction," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 1558–1562.

[15] S. E. Rouayheb, S. Goparaju, H. M. Kiah, and O. Milenkovic, "Synchronizing edits in distributed storage networks," *arXiv preprint arXiv:1409.1551*, 2014.
[16] J. Harshan, A. Datta, and F. E. Oggier, "Compressed differential erasure codes for efficient archival of versioned data," *arxiv preprint*, 2015, http://arxiv.org/abs/1503.05434.
[17] A. S. Rawat, S. Vishwanath, A. Bhowmick, and E. Soljanin, "Update efficient codes for distributed storage," in *2011 IEEE International Symposium on Information Theory Proceedings (ISIT)*. IEEE, 2011, pp. 1457–1461.
[18] Z. Wang and V. Cadambe, "Multi-version coding in distributed storage," in *2014 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2014, pp. 871–875.
[19] C. Tian, "Characterizing the rate region of the (4, 3, 3) exact-repair regenerating codes," *Selected Areas in Communications, IEEE Journal on*, vol. 32, no. 5, pp. 967–975, 2014.
[20] S. Brahma and C. Fragouli, "Pliable index coding," in *2012 IEEE International Symposium on Information Theory Proceedings (ISIT)*. Ieee, 2012, pp. 2251–2255.
[21] L. Song and C. Fragouli, "Content-type coding," *arXiv preprint arXiv:1505.03561*, 2015.
[22] I. Griva, S. G. Nash, and A. Sofer, *Linear and nonlinear optimization*. Siam, 2009.
[23] Z. Wang and V. Cadambe, "On multi-version coding for distributed storage," in *52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2014, pp. 569–575.
[24] R. Stewart, "Stream control transmission protocol," 2007, RFC 4960: Available at https://tools.ietf.org/html/rfc4960.
[25] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, 1987, pp. 137–151.