# *Mist*: Efficient Dissemination of Erasure-coded Data in Data Centers

Jun Li, *Member, IEEE,* Baochun Li, *Fellow, IEEE,* and Bo Li, *Fellow, IEEE*

**Abstract**—Data centers store a massive amount of data in a large number of servers built by commodity hardware. To maintain data integrity against server failures, erasure codes have been extensively deployed in modern data centers to provide a higher level of failure tolerance with less storage overhead than replication. Yet, compared to replication, disseminating erasure-coded data from a source server into multiple servers will also take significantly more time.

In this paper, we design and implement *Mist*, a new mechanism for disseminating erasure-coded data efficiently to multiple receiving servers (receivers) in data centers. *Mist* speeds up the dissemination process by building an efficient topology among the receivers with heterogeneous performance, so that coded data can be received from other receivers in a pipelined fashion, rather than directly from the source. *Mist* flexibly supports a wide range of erasure codes, without imposing constraints to the range of system parameters, and can be extended for specific erasure codes with better performance by taking advantage of the corresponding erasure code. We have implemented *Mist* in Python, and our experimental results in Amazon EC2 have demonstrated that the dissemination time can be reduced by up to 96.3% with different kinds of erasure codes.

## 1 Introduction

An enormous amount of data have been stored in distributed storage systems, such as Google File System (GFS) [1] and Hadoop Distributed File System (HDFS) [2], inside data centers. For example, Facebook stores at least 300 PB of Hive data inside its data centers, increasing by 600 TB per week [3]. These data are stored on servers built by commodity hardware, and frequent failures can be expected even on a daily basis [4]. To maintain high data availability against server failures, data are typically replicated across multiple servers or even across different racks. For example, three copies are stored (*i.e.*, 3-way replication) in HDFS [2] by default. This way, failures of servers or even rack switches will not affect data availability.

Naturally, storing multiple copies of the original data incurs heavy storage overhead in data centers. To save storage overhead while maintaining the same level of data availability, erasure codes have been deployed by some large-scale distributed storage systems [5], [6], with Reed-Solomon codes as the most common choice.

An $(n, k)$ Reed-Solomon code encodes $k$ units of original data into $n$ units of coded data, in which any $k$ units can recover the original data. To maximize the ability to tolerate failures, they are disseminated into $n$ different servers. This way, Reed-Solomon codes can significantly save storage overhead while tolerating the same number of server failures. For example, to tolerate any 2 server failures, three copies must be stored (*i.e.*, 3x storage overhead) while a $(6, 4)$ Reed-Solomon code only requires 1.5x storage overhead.

In this paper, we study the overhead to disseminate erasure-coded data from a source server into multiple servers in the distributed storage system. Comparing with replication, writing data into distributed storage systems with erasure coding can incur much higher overhead than writing replicated data. We use HDFS as a use case to explain this overhead. When data are written from a source server into receiving servers (also known as *datanodes* in HDFS), HDFS will build a pipeline among receivers, and the source will only need to send data to one of the receivers [2], which will relay the received data to another receiver, as shown in Fig. 1(a). If we assume that all servers are connected to a switch with 1 Gbps links, it will then take 2 seconds to write 256 MB of data into all three receivers. However, if the data need to be encoded with a $(4, 2)$ Reed-Solomon code as shown in Fig. 1(b), each receiver should download 128 MB of data from the source in dependently, which can be fininshed in 4 seconds as the outgoing link at the source will be shared by the four receivers now. Therefore, we can see that the outgoing link of the source becomes the bottleneck of the dissemination. As existing production storage systems that use erasure codes all ask their receivers to download data from a single source directly [5], [7], the time it takes to disseminate erasure-coded data in sucn systems should increase linearly

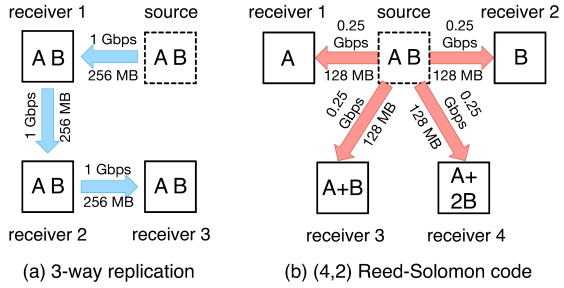



Fig. 1. Data dissemination of replication and Reed-Solomon codes.

- *Jun Li is with the School of Computing and Information Sciences, Florida International University.*
- *Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto.*
- *Bo Li is with the Department of Computer Science and Technology, Hong Kong University of Science and Technology.*

with the number of receivers.

To save the time of disseminating erasure-coded data, we propose and implement a new data dissemination system, referred to as *Mist*. *Mist* eases the burden at the source, by minimizing the number of receivers that directly contact it, and letting other receivers be allowed to compute their desired data from data on these receivers instead of the source. We illustrate one example in Fig. 2 that we disseminate 256 MB of data with a $(4, 2)$ Reed-Solomon code, where we let only 2 receivers contact the source directly. Other receivers, at the same time, will obtain data from the first 2 receivers and compute their desired data on their own. The first two receivers will pipeline their data to their downstream receivers, such that the other two receivers can compute their desired data and finish the dissemination (almost) simultaneously with the first two receivers. Therefore, the time to finish the dissemination can be saved to just 2 seconds, the same as disseminating the replicated data.

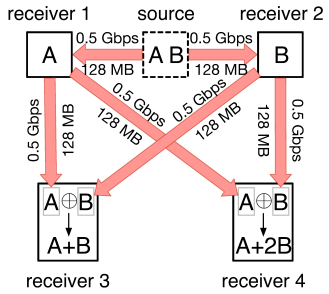



Fig. 2. An example of disseminating data coded by a $(4, 2)$ Reed-Solomon code in *Mist*.

In general, *Mist* is designed to work with any erasure code in any distributed storage systems. *Mist* minimizes the number of receivers that download data directly from the source to alleviate its bottleneck. To disseminate data to all receivers, we build a pipelining topology such that they are allowed to get their desired data from other receivers. We carefully control the traffic going out of any servers in the pipelining topology to avoid any additional bottleneck. Our construction of such a topology can work for a wide range of erasure codes in general, which can also support to disseminate hybrids of both replications and erasure-coded data. In *Mist*, we provide a flexible mechanism to extend its support to even more erasure codes with their specific topology constructions. Besides the general construction, we propose specific topology constructions of two representative kinds of erasure codes in data centers, taking advantage of the corresponding erasure code to save time and even network traffic consumed during dissemination. If the performance of receivers are heteogeneous, *Mist* can even carefully place the receivers into best positions in the topology in order to minimize the time to finish the dissemination.

We have implemented *Mist* in Python and evaluated it with Amazon EC2. We have shown that with *Mist*, disseminating erasure-coded data becomes much less sensitive to the number of receivers. Because of the pipelining topology with carefully controlled out-going traffic, the dissemination time can be very close to the theoretical minimal amount of time to disseminate just the original data. In this way, we can significantly save the time used to disseminate erasure-coded data by up to $96.3\%$ with different kinds of erasure codes.

## 2 Overview of *Mist*

The goal of *Mist* is to provide a general framework to reduce the time of disseminating erasure-coded data in the data center. As shown in Fig. 2, *Mist* minimizes the number of receivers that contact the source directly and lets other two receivers reconstrcut their desired data from such receivers. When we have more than 4 receivers, however, the 2 receivers that contact the source directly will become new bottlenecks as all other receivers obtained data from them. In *Mist*, we build multi-level topologies for any number of receivers incurring no more bottleneck anywhere in the topology, that not only works with Reed-Solomon codes, but with almost all erasure codes in general. In other words, the receivers in the topology will be placed into different levels such that all receivers in any level will obtain data from receivers in another level closer to the source. Besides a topology construction for erasure codes in general, we also propose specific topology constructions of local reconstruction codes [5] and minimum-storage regenerating codes [8], two representative kinds of erasure codes designed for storage systems in data centers, by taking advantage of their own properties.

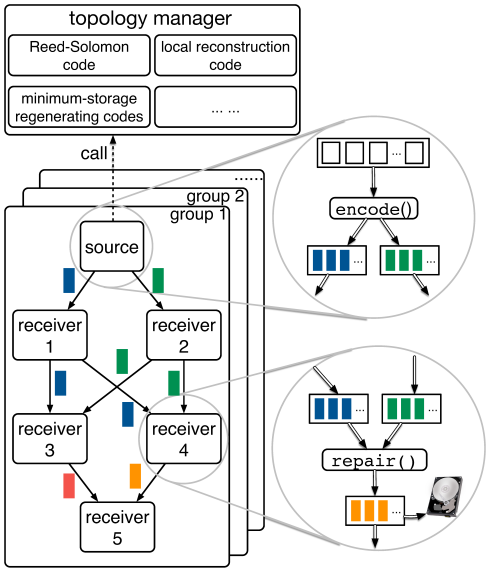



Fig. 3. The architecture of *Mist*.

Fig. 3 shows the architecture of *Mist*. In *Mist*, a group contains one source and multiple receivers. The topology manager in *Mist* is responsible for generating the topology of the corresponding erasure code. All receivers will connect to other receivers or to the source according to the topology. The topology will also instruct the behavior of the source and each receiver by implementing two functions: `encode()` to compute erasure-coded data at the source and `repair()` to compute coded data from other coded data at the receiver. This way, each receiver will compute its desired unit from the source or from other receivers, store the corresponding units to local disks, and meanwhile, send them to other receivers in the next level of the topology.

When there are multiple *Mist* groups running in parallel, the topology manager does not run in a centralized manner. Instead, the source in each group will run its own instance of the topology manager to construct the pipelining topology of its own group, such that we can to a large number of groups *Mist*.

To streamline the transmission, each server will not compute the entire unit of coded data directly, but build a pipeline by dividing them into stripes, each with a fixed size. Once a stripe has been computed, the server will send it immediately to downstream receivers. There may be a trade-off in choosing the stripe size, because a larger stripe will increase the computational delay between two consecutive levels in the topology, while a small stripe will lead to a low throughput of encoding operations [9]. In our forthcoming experiments, we will evaluate how various choices of the stripe size affect the dissemination process.

In the rest of the paper, we will focus on the topology construction of different erasure codes. We start with a construction of general erasure codes in Sec. 3. Moreover, *Mist* provides a flexible mechanism to specifically construct topologies of any particular erasure codes. To implement a new topology for a certain erasure code in *Mist*, we only need to implement the construction of such a topology and the corresponding `encode()` and `compute()` functions. With this flexible mechanism, we consider designs of topologies for specific erasure codes in Sec. 4, by taking advantages of their own properties. In Sec. 3.3, we show that *Mist* can also build a pipelined topology to disseminate replicated data along with erasure coding.

## 3 Topologies for General Erasure Codes

### 3.1 General requirements of erasure codes in *Mist*

Typically, an erasure code in a distributed storage system contains two system parameters $n$ and $k$, $k < n$. An $(n, k)$ erasure code encodes the original data into $n$ units of a fixed size, where any $k$ units can be decoded to recover the original data. If $k$ units of coded data contain the same size of the original data, such erasure code achieves the optimal storage overhead with the same failure tolerance. As an instance, Reed-Solomon (RS) codes achieve such optimality. Moreover, if there exist $k$ blocks of coded data that exactly embed the original data, such erasure codes are called *systematic* codes. The units that contain the original data is also known as the *systematic units*. In other words, the original data can be directly accessed without any decoding operations, promising a better read throughput in the storage system. The RS code illustrated in Fig. 2 is an example of $(n = 4, k = 2)$ systematic erasure codes, with four units of coded data, *i.e.,* $A$, $B$, $A + B$, and $A + 2B$. It is easy to verify that $A$ and $B$ can be decoded from any two units of coded data.

In *Mist*, we do not require the optimal storage overhead or systematic codes as we construct topologies for erasure codes in general. We only require that 1) data are encoded into $n$ units of a fixed size; 2) any $k$ units of coded data can recover the original data. We believe that such assumptions are valid for a very wide range of erasure codes used in distributed storage systems. However, the topology can achieve better performance if the erasure code is optimal in terms of the storage overhead or is systematic.

### 3.2 Topology design

Given an $(n, k)$ erasure code, we can compute any one unit of coded data from any other $k$ units. For example, with the RS code shown in Fig. 2, if a server can get $A$ and $B$ (without decoding operations in this case since this RS code is systematic, otherwise after decoding any two units if this RS code is not systematic or there is at least one non-original unit in these two units), then it can encode these 2 units again to compute $A+B$ or $A+2B$. We refer to this operation as *repair*, as in distributed storage systems the repaired data replace the old data when they are not available.

As described above, the repair of any unit of coded data requires decoding and re-encoding, *i.e.,* decoding existing units of coded data to get the original data, and then encoding the original data again to get the desired unit.[1] Therefore, if a receiver is going to repair one unit of coded data, it can also behave like a source to serve other receivers because it needs to recover the original data anyway. Following this intuition, we can build a topology to disseminate erasure-coded data to multiple servers without incurring a bottleneck at the source.

We show the topology construction through an example of the topology built for an $(8, 2)$ erasure code in Fig. 4. Without loss of generality, we assume that receiver $i$ needs block $i$, $i = 1, \ldots, 8$, and we show the corresponding block beside each edge in Fig. 4. We build a multi-level topology to disseminate the coded data without incurring a bottleneck at the source server or any receiver. In order to compute one unit from other units of coded data, we need to have at least 2 units of them. Hence we place two receivers (receiver 1 and 2) at the first level in this topology. With these two receivers, all the other receivers can repair their desired units without visiting the source directly. On the other hand, adding any more receivers into the first level will decrease the incoming throughput of each receiver. Hence, we can place at most two receivers at the first level in this example. Since the repair operation always incurs encoding overhead, we can put the 2 receivers with the systematic units at the first level if the $(8, 2)$ erasure code is systematic, such that the encoding overhead at the source can be saved.

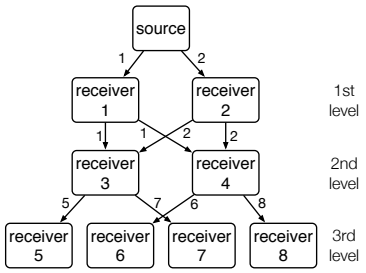



Fig. 4. The topology of an (8,2) erasure code in *Mist*.

Now that we have put two receivers into the first level, we can ask them to forward received data to receivers put in

1. There exist erasure codes that can directly compute other units of coded data without decoding. We do not consider this property for now, in pursuit of generality.

the second level, which repair their own data from the first two units. However, if we put too many receivers into the second level, the first two receivers will need to send out too many data and incur a new bottleneck of the network. How many receivers can we put into the second level at most? We know that a receiver needs to contact two servers to repair its desired data, and thus each receiver at the second level needs to contact both two receivers at the first level. Notice that both the source and the first two receivers send out one unit of coded data to each of their downstream receivers. Since the source is serving two servers, the two receivers should serve at most two other receivers as well. In other words, we can have two receivers in the second level.

However, the two receivers at the second level (receiver $3$ and $4$) can serve more receivers at the third level, because different from receiver $1$ and $2$ that receive their desired units directly from the source, receiver $3$ and $4$ repair their desired data from data downloaded from their upstream receivers. In other words, they can decode the original data and thus behave like a source for any receivers at the third level. In this way, we can have four receivers at the third level so that both receiver $3$ and $4$ serve two receivers.

If there are even more receivers in the dissemination group, we can follow the way described above to add more levels into such a topology. For example, in the fourth level, there can be still at most four receivers, since they need to repair their data by contacting receivers at the third level. Then they will send units of coded data to at most $8$ receivers at the fifth level, and so on. Notice that we strive to balance the workload of receivers by assigning the upstream receivers in a round-robin manner, as shown in Fig. 4.

Similarly, we can have a general construction to build the topology of $(n, k)$ erasure codes. The general idea is to add only $k$ receivers — the theoretical minimum number of receivers from which other units can possibly be repaired — at the first level and let $k$ more receivers be placed at the second level to compute their desired data. Then we will have $k^2$ receivers at the third and fourth level, and $k^3$ receivers at the fifth and sixth level, *etc*. This way, we can build a topology of an $(n, k)$ erasure code with any valid choices of parameters. The complexity of this method is linear to the number of receivers.

In fact, considering that the practical value of $n$ will be no more than a few times of $k$, the depth of such topologies will also be limited. For example, if $n \leq 2k$, the constructed topology will have at most 2 levels of receivers. Only one more levels will be added if $n$ is more than $2k$ but no more than $2k + k^2$. Since $k \geq 2$, three levels of receivers will be enough for no more than $4k$ receivers. When $n = 4k$, the minimum storage overhead (*e.g.*, with RS codes) is $4$x, which is probably enough for most scenarios in distributed storage systems in current data centers.

The topology constructed with this method can reduce the computational overhead at the source to the theoretical minimum, as it only needs to encode $k$ units of coded data. Even better, there is no such overhead when the erasure code is systematic. On the other hand, receivers at the even levels in the topology will need to decode and re-encode data. In distributed storage systems, most erasure codes are linear codes[2] such that all operations can be applied by linear operations. Linear erasure codes can directly repair one unit of coded data from any other $k$ units without decoding. Typically, these receivers need to repair $k$ units of coded data to their downstream receivers, incurring the same computational overhead as the source. Receivers at the odd levels will only need to send their received data to their downstream receivers, with very little additional computational overhead.

### 3.3 Disseminating erasure-coded data with replications

Though erasure codes provide a higher data availability against server failures with much less storage overhead than replications, it is hard to offer a high read throughput with erasure codes under a high volume of workload. Even though systematic erasure codes can exempt the storage system from decoding, all requests will then be directed to the servers that store the original units. Therefore, it is hard to do load balancing with erasure codes. Currently in the real world, distributed storage systems that deploy erasure codes may also store replications with erasure coded data, and migrate replications into more erasure-coded data as the data becomes cold [10].

In this section, we present a design of topology construction in *Mist*, extended from the topology constructed for erasure-coded data only, to disseminate erasure-coded data along with replications. This design can not only disseminate a mix of both erasure-coded data and replications quickly, but save the system from disseminating unnecessary copies as well.

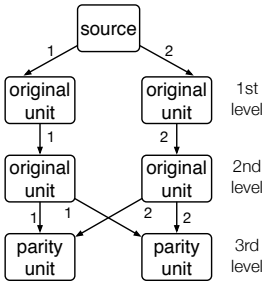



Fig. 5. The topology to disseminate a systematic (8,2) erasure code with 2-way replication of the original units. While replicating the original units at the first two levels, all other (*i.e.*, parity) units are disseminated in the same way as the topology for general erasure codes.

Suppose that we have a systematic $(n, k)$ erasure code, and we need to have an $r$-way replication of each original unit. Notice that we can disseminate replications by simply asking receives to relay its copy to one another. We apply this principle into our topology design, as shown in Fig. 5 where we need to have a 2-way replication of all original units. Comparing with Fig. 4, we add one more level between the source and the first level of receivers, where receivers can the relay original units to the receivers, which now are at the second level. In this way, we can add as many levels as necessary to replicate any number of units for any number of times.

2. For example, RS codes, along with the other erasure codes mentioned in this paper, are all linear codes.

If the erasure code is not systematic, on the other hand, we can still insert the original units into the first $r$ levels similarly, as they can always repair any parity units.

This method works for the topology of any erasure codes supported in *Mist*, including those presented in Sec. 4 as well. Once we have a topology constructed for an erasure code and some units need to be replicated, we can add additional receivers above the corresponding receivers and let them store and relay received data to their following receivers. This way, any topology can be extended to replicate any units of coded data in it.

## 4 Topologies for Specific Erasure Codes

In this section, we present topology constructions that are designed for specific erasure codes in distributed storage systems. Compared with the topology constructed for general erasure codes, we take advantage of code structures of specific erasure codes in these constructions, to save the time of dissemination, and network traffic, or CPU overhead as well.

### 4.1 Local reconstruction codes

Local reconstruction codes, which are deployed in Windows Azure Storage [5], save disk I/O during repair by adding the "locality" into erasure codes, such that most units of coded data can be repaired from a small number of some other units. Thus, only a small number of servers that store these units will be visited and thus incur low disk I/O during repair, without bothering any other servers.

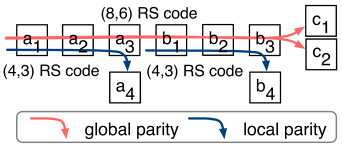



Fig. 6. An example of (6,2,2) local reconstruction codes, with 6 original units, 2 local parity units and 2 global parity units.

Fig. 6 shows an example of local reconstruction codes (LRC). Supposing that we already have an $(8, 6)$ systematic RS code, which computes 6 coded units containing 6 original units and two parity units, we add two more parity units computed from each 3 original units with a $(4, 3)$ systematic RS code. In local reconstruction codes, the parity units of the $(8, 6)$ RS code is called *global parity units*, while the two additional units are called *local parity units*. This way, we build a $(k = 6, l = 2, g = 2)$ local reconstruction code, where $k, l$, and $g$ represent the number of original/systematic units, local parity units, and global parity units, respectively. If any original unit or local parity unit is lost, it can be repaired by accessing only three units rather than six units. For example, to repair $a_1$, we just need to obtain $a_2$, $a_3$ and $a_4$, and then compute $a_1$ with the $(4, 3)$ RS code, rather than decoding 6 units (such as $a_2 - a_3$, $b_1 - b_3$ and $c_1$) with the original $(8, 6)$ RS code. This way, the disk I/O overhead can be saved by visiting fewer servers when repairing most units of coded data (except global parity units). Notice that though we can decode many combinations of 6 units, not every 6 units are sufficient to recover the original data.

As described above, a $(k, l, g)$ local reconstruction code makes it possible to repair a local parity unit from $\frac{k}{l}$ units. However, they must be computed from a specific combination of units. Fortunately, local parity units are all computed from original units instead of global parity units. Therefore, to construct the pipelining topology for local reconstruction codes, we can put original units in the first level and then repair local parity units at the second level. Since the $k$ original units are the smallest combination of units to recover the original data, the source achieves the theoretical maximum throughput to each receiver at the first level. Receivers of local parity units, at the second level, are going to download the corresponding original units. Since each original unit is used to compute only one local parity unit, each receiver of the original unit serves just one receiver so far. Compared with the topology constructed for general erasure codes, these receivers only need to receive $\frac{k}{l}$ units. Thus, network traffic can be saved, and the computational overhead can also be reduced as the RS code used is smaller.

Since the source serves $k$ receivers at the first level, we can consider that all receivers have a capacity to serve at most $k$ downstream receivers without incurring any new bottlenecks. To disseminate global parity units, we first exploit the remaining capacity of the receivers at the first level, since they have served only one receiver. Since erasure codes require that $k$ is at least 2 (otherwise they are equivalent to replications), the source and receivers at the first level can serve at least two receivers. Thus, each receiver at the first level has a remaining capacity of at least one receiver. Therefore, we can repair at least one global parity unit at the third level, except that this receiver needs to contact the $k$ receivers at the first level. Fig. 7 shows an example of the dissemination with a $(4, 2, 1)$ local reconstruction code where the global parity unit is being computed at the third level.

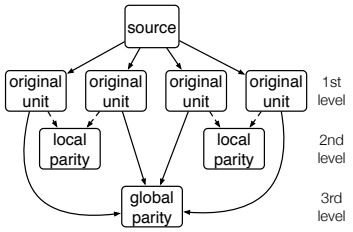



Fig. 7. The topology of a $(4, 2, 1)$ local reconstruction code constructed in *Mist*, where there are 4 original units, 2 local parity units and 1 global parity unit to disseminate. Dashed lines represent the connections to disseminate local parity units, and other lines represent the connections to disseminate original units and the global parity unit.

In summary, in $(k, l, g)$ local reconstruction codes, the $k$ receivers at the first level will serve $l$ receivers of the local parity units at the second level. At the third level, we can have most $k - 1$ receivers to repair their global parity units. If there are more global parity units to disseminate, the receivers at the third level can behave like a source to other receivers at the fourth level, similar to the topology of general erasure codes. In other words, we can have $k(k-1)$ receivers at the fourth level at most, and the same number at the fifth level as each of them needs to compute their units by contacting $k$ servers at the fourth level. Similarly, we will have $k^2(k-1)$ receivers at most at the sixth and

seventh level, and so on until all $g$ global parity units can be computed.

### 4.2 Regenerating codes

Different from local reconstruction codes, the objective of regenerating codes is to optimize the network traffic during repair. Dimakis *et al.* [11] have revealed a lower bound of network traffic to compute a unit of coded data from $d$ existing units ($d \geq k$), with a general $(n, k)$ erasure code. In particular, if the erasure code achieves the optimal storage overhead like RS codes, *i.e.,* the size of a unit is just $\frac{1}{k}$ of the original data, each of the $d$ servers only needs to offer $\frac{1}{d-k+1}$ to the replacing server. The erasure code that achieves such a *repair property* is termed as *minimum-storage regenerating* (MSR) codes.[3] There has been various literature that discusses the construction of MSR codes ( [12] and the references therein). We can see that MSR codes can achieve a significant reduction of network traffic during repair, compared to both RS codes and local reconstruction codes.

To disseminate an $(n, k, d)$ regenerating code efficiently, one challenge comes from the property of regenerating codes that a unit of coded data can be repaired with low network traffic consumption from $d$ units, where $d \geq k$. If we let the source send the $d$ units directly to receivers, the throughput between the source and the receiver will be compromised, which will further affect the throughput in the downstream levels. We do not compromise the throughput in our topology design by reserving the capacity of the receivers involved.

When $d = k$, an $(n, k, d)$ MSR code is equivalent to a general $(n, k)$ RS code. Therefore, we assume that $d > k$ in the following topology construction. In other words, if the source serves only $k$ receivers at the first level, there must be at least one receiver missing before we can compute other receivers by the repair property of MSR codes.

Notice that though an MSR code provides a bandwidth-efficient method to repair a unit of coded data from $d$ existing units (by the repair property), a unit can also be repaired in the way like a general erasure code, by decoding $k$ units and re-encoding the decoded data into the corresponding unit (which, for convenience, is referred to as the *decodability* of the erasure code). In general, to take advantage of MSR codes in the topology construction, we need to apply the repair property on as many receivers as possible to compute their corresponding units of data. In other words, we will compute the first $d$ units by decodability and compute all the rest units by the repair property.

Similar to LRC codes, assuming that the source can send $k$ units to its downstream receivers, we define the *capacity* of a server (a source or a receiver) to be the maximum number of units it can send to its downstream receivers without incurring any additional bottlenecks. We first build a multi-level topology similar to the topology for RS codes, to disseminate data to $d$ receivers by the decodability of the MSR code. However, we reserve $x$ units in the capacity of these $d$ receivers, which will be used to compute other receivers with the repair property. Since the capacity of all receivers should be no more than $k$ to avoid incurring any additional bottlenecks, the value of $x$ must be less than $k$.

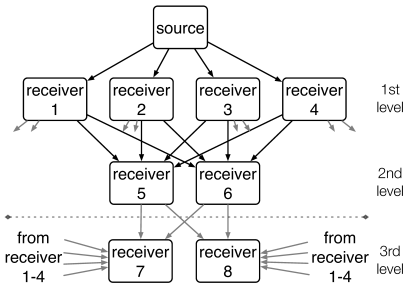



Fig. 8. An example topology for an (8,4,6) minimum-storage regenerating (MSR) codes. The source sends just 4 units of coded data to the receivers at the first level and then repair two more units in the receivers at the second level by decoding and re-encoding. From the six receivers at the first two levels, two other receivers can compute their desired data by the repair property of regenerating codes.

Fig. 8 shows an example of disseminating data with an $(8, 4, 6)$ MSR code. With four receivers at the first level, there are still two more receivers before we can repair data at more receivers by the repair property. Thus, we let the two receivers at the second level repair their units by the decodability of this regenerating code, and meanwhile we reserve two units of the capacity at each receiver in the first level, *i.e.,* $x = 2$. With these six receivers, we can start to repair the desired data of the two remaining receivers by the repair operation.

With the capacity of $x$ units reserved at $d$ receivers for other receivers down in the topology, how many receivers can be placed at each level? As there are $k$ receivers at the first level, there can be at most $k - x$ receivers that contact the $k$ receivers at the first level to compute their units by decodability, and they can also repair and send data to the receivers at the third level. Thus, there can be at most $(k - x)^2$ receivers at the third level. Similarly, there can be at most $\lfloor \frac{(k-x)^3}{k} \rfloor$ and $(k - x)\lfloor \frac{(k-x)^3}{k} \rfloor$ receivers at the third and fourth level, and so on, until the first $d$ receivers have been accommodated.

To compute data at the remaining $n - d$ units by the repair property, it is important to remember that a receiver just receives $\frac{1}{1-d+1}$ instead of the whole unit from each of the $d$ receivers, due to the repair property of the MSR code. In other words, with the capacity of $x$ units, a receiver can actually serve $x(d-k+1)$ receivers by the repair operation.

Suppose that the first $d$ receivers are placed in the first two levels, just like the example in Fig. 8. Hence, the third level can accommodate at most $x(d - k + 1)$ receivers, and the fourth level can have at most $\lfloor \frac{xk(d-k+1)^2}{d} \rfloor$ receivers. Similarly, the fifth level can accommodate $\lfloor \lfloor \frac{xk(d-k+1)^2}{d} \rfloor \frac{k(d-k+1)}{d} \rfloor$ receivers at most, and so on.

To construct the topology for an $(n, k, d)$ regenerating code, the value of $x$ must be given in advance. However, the value of $x$ determines if such a topology exists or not. We show, in the Appendix, that this topology is valid if and

3. There is another family of regenerating codes called minimum-bandwidth regenerating (MBR) codes. The topology construction of MBR codes can also be obtained in a way similar to MSR codes. Due to the space constraints, we do not discuss the topology construction of MBR codes in this paper.

only if $x \in [\frac{d}{d-k+1}, k-\sqrt{k}]$ for MSR codes. Rashmi *et al.* [8] have shown that there exists no deterministic construction of exact-repair MSR codes when $d < 2k-3$. Therefore, in most cases $x$ (when $k \geq 4$) we can simply take 2 as the value of $x$. In the extreme cases that a valid value of $x$ does not exist (*e.g.*, $k = 3$), we can just use the general construction, to compute data at all receivers by the decodability.

## 5 *Mist* WITH HETEROGENEOUS SERVERS

So far, the design of the topologies are considered under the assumption that the performance of all receivers are homogeneous and the throughput of encoding at receivers does not become a bottleneck. However, in a practical cluster the performance of servers can often be heterogeneous. For example, it has been reported that servers in a data center can have different hardware configurations [13], leading to heteogenous performance when encoding data in a *Mist* topology. Given the heterogeneous performance of receivers, we cannot simply place receivers at any position in the *Mist* topology. The reason is that receivers at different levels in a *Mist* topology will need to encode data received from different numbers of other receivers and serve different numbers of other receivers, as we can see from all previous examples. Therefore, a receiver with a low CPU performance may encode data slowly and make itself become the bottleneck in the whole topology. In this section we consider the performance heterogeneity of receivers and assign them to the best positions in the corresponding topology.

In this paper, we consider the topology for general erasure codes only due to the space limit, and similar mechanisms can be developed for local reconstruction codes and regenerating codes as well. Given a *Mist* topology, we first consider the complexity of the encoding at each receiver by taking into account the number of incoming flows and outgoing flows. Assume that a receiver need to download data from $N_i$ servers and $N_o$ receivers download data from this receiver, and then the data this receiver need to compute can be represented as a matrix multiplication by multiplying an $(N_o+1) \times N_i$ matrix on the left of the received data because the receiver will also need to calculate its own desired data. Therefore, the encoding complexity is $O\left((N_o+1)\,N_i\right)$.

Assume that we have already the known the sequence of the CPU performance of all receivers, we can then match them to the corresponding position in the topology where the receiver with the highest CPU performance should have the highest encoding complexity and the receiver with the lowest CPU performance should have the lowest encoding complexity. It is easy to prove that this way can lead to the maximum overall encoding throughput, as exchanging the positions of any two receivers can lead to even lower throughput.

## 6 IMPLEMENTATION

We have implemented *Mist* following the architecture shown in Fig. 3, with 3000 lines of code in Python and 1000 lines of C++ code implementing a Python module of erasure codes. We use the Intel storage acceleration library [14] to implement the arithmetic operations of erasure codes in this module. The source and the receiver are implemented as standalone applications. When a group is started, the source is going to utilize the topology manager to compute the corresponding topology. The topology manager is a class that defines the interface to generate the topology for various erasure codes. A topology manager for a specific kind of erasure codes will inherit this base class to implement its corresponding topology construction.

In order to avoid any potential bottleneck of the write throughput in the hard disk, once a receiver has received or repaired a stripe of its desired unit, we first save the data into a buffer in the memory, and use another thread to keep writing the data in the buffer into the hard disks as long as the buffer is not empty.

We have currently supported RS codes, local reconstruction codes, and MSR[4] codes in *Mist*, where the topology construction of RS codes is built by the general topology construction as described in Sec. 3, and other codes have their own specific topology construction as described in Sec. 4. Any erasure codes supported in *Mist* can also work with replications, as described in Sec. 3.3.

To make sure that all receivers can receive all their desired data, each receiver will notify the source when they've received all the data successfully. The source will exit successfully when it has received the notifications from all receivers. On the other hand, the pipelining topology will make all receivers receive all data almost at the same time. Hence, we can set a timeout at the source such that a receiver will be considered as failed if it does not finish after a certain amount of time since the source finishes sending all data to receivers. The source will also exit with an error code if any TCP connection fails during transmission or any other exception occurs. Therefore, the user only needs to check the status of the source to detect failures in *Mist*. Once any failure has been detected, the user can kill the corresponding *Mist* group and restart by launching a new *Mist* group.

## 7 EVALUATION

### 7.1 Evaluation methodology

We have evaluated *Mist* in Amazon EC2. In each experiment (unless mentioned otherwise), we create a given number of EC2 nodes of type c4.xlarge (with 4 CPU cores on an Intel Xeon E5-2666 processor and 7.5 GB of memory) in the same availability zone. Each node will run one instance of the source or the receiver in one group of the dissemination. From various experiments with different configurations, we have found that the performance, especially the dissemination time in *Mist* is rather stable. Therfore, the experimental results are obtained by calculating the average result of 10 iterations, running on the same EC2 nodes. The standard devation of the results in most cases is within 5% of the average.

We measure the time spent and network traffic incurred during the dissemination of erasure-coded data. We believe that these two metrics are essential to the distributed storage system as they determine the write performance of the system. For the purpose of performance comparison, the legacy

4. The MSR codes implemented in *Mist* is constructed using the product-matrix method proposed by Rashmi *et al.* [8].

topology that all receivers download their data directly from the source is also supported in *Mist*.

In a typical distributed storage system, such as HDFS, the data are stored into multiple blocks with a fixed size. Therefore, the original data contains 256MB (unless mentioned otherwise) in the experiment. In each experiment, we launch *Mist* with one source and a given number of receivers, randomly selected from existing EC2 nodes. The experiment runs on Python 2.7.6 with the stripe size set to be 1MB (unless mentioned otherwise).

### 7.2 Dissemination time

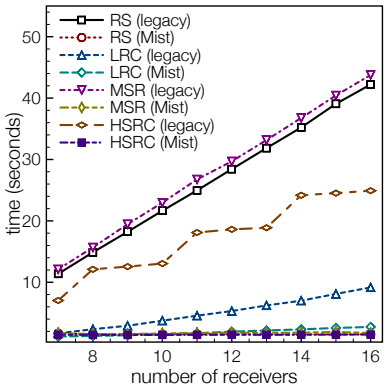



Fig. 9. A comparison of the dissemination time of (n, 4) Reed-Solomon codes, (n, 4, 6) minimum-storage regenerating (MSR) codes and (4, 2, n - 6) local reconstruction codes (LRC), with the topology in *Mist* and the legacy topology. The value of n indicates the number of receivers in the dissemination.

Fig. 9 compares the total dissemination time of the topologies in *Mist* with the legacy topologies. We can see that the dissemination time of the legacy topology increases with the number of receivers. In addition, the dissemination time is bottlenecked by the encoding operation instead of the outgoing bandwidth at the source, since with the same amount of data going out of the source the dissemination time of different erasure codes can vary. The topologies built in *Mist*, on the other hand, can significantly reduce the dissemination time (by up to 96.3%). Compared to the legacy topologies, the dissemination time of all erasure codes with *Mist* is very close to the theoretical minimum dissemination time (0.89 seconds). Hence, with *Mist* the dissemination time no longer increases significantly with the number of receivers. The reason is that in our experiment, the major contribution of dissemination time comes from the network (as we elaborate in Sec. 7.3), which is now related to only the size of original data, instead of all data including original data and parity data. Additional delay in the dissemination time comes from the computational delay of processing stripes at receivers. Thus the dissemination time will only increase slightly with the number of levels in the topology.

In this experiment, we also incorporate another family of erasure codes, called *self-repairing homomorphic codes* (HSRC) and its construction of the pipelining topology [15] into *Mist*. Though the purpose of this is to demonstrate the extensibility of *Mist* instead of performance comparison, HSRC codes achieve even lower dissemination time in *Mist*

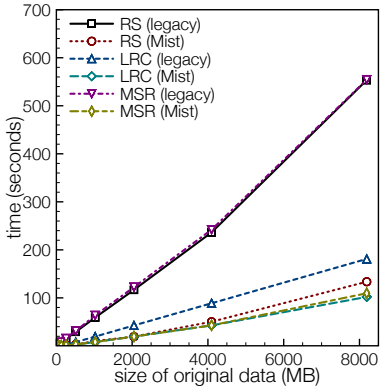



Fig. 10. A comparison of the dissemination time with original data of different sizes, with (8, 4) Reed-Solomon codes, (8, 4, 6) minimum-storage regenerating (MSR) codes and (4, 2, 2) local reconstruction codes (LRC).

(except for a small number of receivers) because every unit of coded data with HSRC codes can be computed from two other units by XOR operations, leading to the lowest computational delay in this experiment. In other words, the performance of dissemination time in *Mist* can be affected by the complexity of the erasure code itself.

As Fig. 10 shows, the dissemination time also increases linearly with the size of the original data. This is easy to understand as *Mist* decomposes the original data into multiple stripes of a fixed size. Therefore, *Mist* can scale well to disseminate very large volumes of data.

### 7.3 Analysis of the bottleneck in the pipeline

As data are transmitted through a pipeline in the topology, the dissemination time in *Mist* will be bottlenecked by the slowest component in the pipeline. To understand the bottleneck, we analyze the throughput time of each component in the pipeline. We decompose the pipeline into components including the source, each receiver, as well as the network that connect servers between two adjacent levels in the topology. To compare the throughput time, we measure the time of all data going through each component, where the results are shown in Fig. 11. The topology we study in Fig. 11 is built for an $(11, 4)$ RS code, where we can find similar results in topologies built for other erasure codes in *Mist*.

The throughput time of each receiver is measured by performing the `repair` function without sending/receiving data through the network, *i.e.*, all data that should have been sent or received through the network in the pipeline are now written into or read from memory. Similarly, we encode data into memory on the source to measure its throughput time. We also measure the network throughput time by launching $4$ parallels TCP connections to send a total of $256$ MB of data from one server to another one, which is the maximum amount of data going out of any server in this topology. In Fig. 11, we also show the overall throughput time of the whole topology, *i.e.*, the dissemination time. We can see that the network has caused the most significant throughput time, much more than all other components in the topology. In other words, the throughput of network dominates the throughput of the whole topology, while the rest of the

overall throughput time is contributed by the delay of other components, mainly the time to process the first stripe in each component. We can also see that receivers at the same level in the topology have very similar throughput time, but receivers at different levels can have different throughput time. This reflects their different behaviors defined in the topology. For example, receivers at the first level only need to receive data from the source and relay data to the next level, while receivers at the second level will also have to decode received data to get their desired data, incurring higher throughput time.

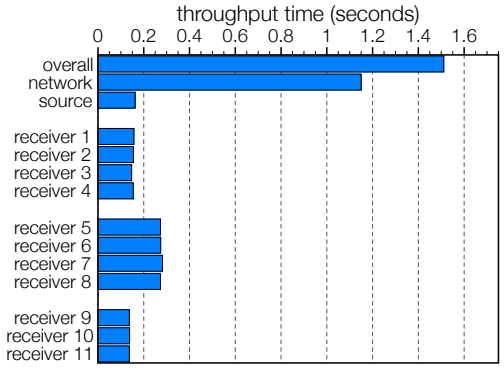



Fig. 11. A comparison of the throughput time of each component in the pipeline, with (11, 4) Reed-Solomon codes.

### 7.4 Impact of the stripe size

Fig. 12 demonstrates the impact of the stripe size in the pipeline. As a surprise, we find that the stripe size does not have a significant impact on the overall dissemination time in both the legacy topologies and those constructed by *Mist*. Even though when the stripe size is too small (less than $64$ KB) or too large (more than $1$ MB), the performance of dissemination time will be slightly affected, any stripe size in-between will provide similar performance with other sizes. We also find that the topologies in *Mist* are more easily affected by small stripe sizes.

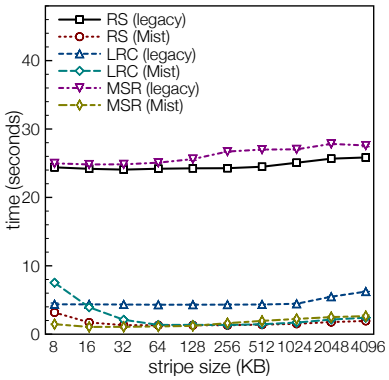



Fig. 12. The impact of the stripe size in the dissemination, with an (11, 4) Reed-Solomon (RS) code, an (11, 4, 9) minimum-storage regenerating (MSR) code and a (4, 2, 5) local reconstruction code (LRC).

### 7.5 Network traffic

In this experiment, we measure the network traffic incurred by *Mist* in the dissemination. Fig. 13 illustrates the results of RS codes, MSR codes, and local reconstruction codes. As for the same parameter, the same amount of traffic will be consumed each time, data given in Fig. 13 are the results of one run only.

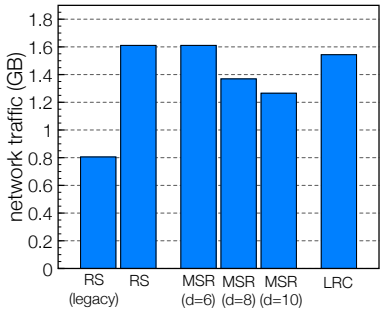



Fig. 13. The network traffic consumed in the dissemination of (12, 4) Reed-Solomon (RS) codes, (12, 4, d) minimum-storage regenerating (MSR) codes (d=6, 8, 10) and (4, 2, 6) local reconstruction codes (LRC).

In a legacy topology, each receiver just needs to download its desired data from the source directly, and thus it incurs the theoretically minimum amount of network traffic. In the topology built by *Mist*, however, more traffic will be expected since a receiver needs to repair its desired unit with multiple units obtained from other receivers.

Therefore, to disseminate a $(12, 4)$ RS codes, *Mist* will require almost twice the amount of network traffic of that in the legacy topology. However, this is much less than the theoretical maximum value, as if all receivers except the first $k$ receivers repair their data from other receivers, theoretically *Mist* is going to incur $(8\times4+4)/12 = 3$ times of the traffic incurred in the legacy topology. It is because in the topology built in *Mist* for RS codes, we exploit the property that any receiver that repair its data from other receivers can also repair any other unit of coded data, and thus we let such receivers behave like a source to its following receivers. This way, we save a significant amount of network traffic in the dissemination process.

As for the other two kinds of erasure codes, the network traffic consumed in the dissemination can be further saved since we take advantage of their repair properties from their specific topology constructions. Local reconstruction codes save a bit of traffic than RS codes ($4.2\%$) as local parity units needs a small number of original units to repair. MSR codes, on the other hand, save much more traffic. With a higher value of $d$, we can save more traffic in the dissemination. The reason can be directly inferred from the repair property of MSR codes, that the traffic consumed to repair one unit of coded data with MSR codes decreases with $d$. Fig. 13 shows that when $d = 8$, MSR codes can consume $15\%$ less network traffic than RS codes in *Mist*, and the saving increases to $21.4\%$ when $d = 10$.

So far, we can see that though *Mist* incurs additional network transfer, it can still save a significant amount of dissemination time. From the construction of topologies in *Mist*, we can know that the reason is that the traffic is generated from multiple sources instead of one in the legacy topologies, and hence the bottleneck at the outgoing link of any server can be avoided.

### 7.6 Disseminating erasure-coded data with replications

As *Mist* can support disseminating hybrids of erasure-coded data with replications, we measure the dissemination time of an $(n, 3)$ RS code with an $r$-way replication of its original units in *Mist*. Still, the legacy topology asks receivers to download all units, including the replicated units, directly from the source.

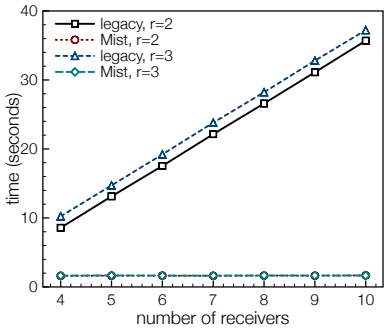



Fig. 14. The dissemination time of a systematic $(n, 3)$ RS coded data with an $r$-way replication of original units, where the value of $n$ indicates the number of receivers of coded data in the dissemination, excluding the additional receivers of replications.

Fig. 14 shows the comparison of the dissemination time between *Mist* and the legacy topology. By adding one more copy of each original unit (*i.e.,* let $r$ go from 2 to 3), *Mist* will spend almost the same time, thanks to the pipelining topology. On the other hand, the legacy topology will spend roughly 2 more seconds for the additional replications.

### 7.7 Concurrent *Mist* groups

In this experiment, we measure the performance of disseminating multiple files with *Mist*. As shown in Fig. 3, the concurrent dissemination can be well handled by launching multiple *Mist* groups. In the legacy topologies, only the source needs to encode data while all receivers just need to receive coded data from the source. However, in the topologies constructed by *Mist*, receivers need to repair data for other receivers. Moreover, with multiple groups running at the same time, the network can be congested when flows from different groups are conflicting with the same physical link. Therefore, one may doubt if the performance of *Mist* will decrease when we run multiple *Mist* groups on the same server. In this experiment, we launch multiple *Mist* groups where the source and receivers are randomly selected from a given number of servers.

Fig. 15 shows the results of running various numbers of *Mist* groups on 8 servers, where each group selects one server as the source and all the rest servers as receivers. Compared to the legacy topologies, topologies in *Mist* will have a higher chance to be affected by the other groups. This is because in the legacy topologies, the only bottleneck is the source. If the source does not coincide, the performance won't be affected. Naturally, with the increase of the number of groups, the chances of having multiple groups with the same source also increases. Hence we can observe a slight increase of dissemination time over an increasing number of groups.

Topologies in *Mist*, however, have more chances to have conflicts of CPU or network among different groups. In our experiments, the source and receivers are always selected from the same set of servers, to maximize the chance of conflicts. With one more group, each server (except those with no child in the topology) will have similar additional amounts of outgoing traffic, as traffic are managed to be evenly distributed to avoid additional bottlenecks. Hence, we can see that the dissemination time of *Mist* topologies increases almost linearly with the number of concurrent groups. However, since the great saving of dissemination time of the *Mist* topology, even though there are 8 concurrent groups, topologies in *Mist* still outperform legacy topologies while there will actually be less concurrent writing on the same server in practice [16]. Notice that in Fig. 15 the dissemination time of different topologies can increase with different paces when the number of concurrent groups is increased. We believe that there exist opportunities to design topology constructions in *Mist* that can achieve lower dissemination time with multiple groups, which will be left as our future work.

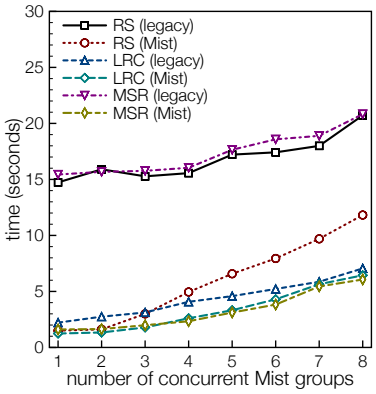



Fig. 15. The dissemination time of multiple concurrent *Mist* groups running with the same source and receivers, with $(8, 4)$ RS codes, $(8, 4, 6)$ MSR codes and $(4, 2, 2)$ LRC codes.

### 7.8 Heterogeneous servers

We now evaluate the performance of *Mist* running on servers with heterogeneous CPU performance. In this experiment, we have 10 receivers, where 8 of those receivers are Amazon EC2 instances of type C4.xlage (with 4 CPU cores on a Intel Xeon E5-2666 processor and 7.5 GB of memory) and the other 2 receivers are of type m2.micro (with 1 CPU core on an Intel Xeon Processor and 1 GB of memory). Our measurement has also shown that the CPU performance of m2.micro instances in EC2 is much slower than those of type c4.xlarge. With this setting, it is easy to imagine that the last 2 receivers are going to be the bottleneck if they are assigned to the position in the topology with a heavy workload. In this experiment, we launch a server as the sender of type c4.xlarge and disseminate data with a $(10, 8)$ RS code. Following the topology design in Sec. 3, it is easy to know that the topology will have eight receivers at the first level that just relay data to the next level and the rest two receivers at the second level that will repair their own data from data forwarded from receivers in the first level. In

other words, if the two receivers with low CPU performance are placed at the second level, they will naturally become the bottleneck because of their low throughput to reconstruct their desired data.

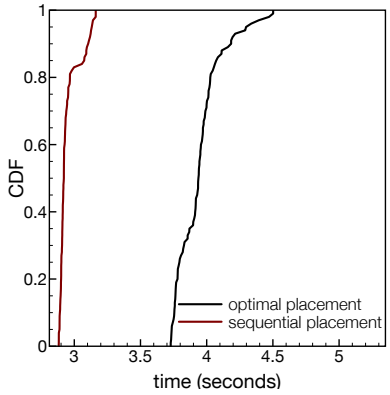



Fig. 16. The CDFs of dissemination time of the topology to disseminate data encoded with a $(10, 8)$ RS code, with and without the algorithm proposed in Sec. 5.

In this experiment, we assume that the $8$ c4.xlarge receivers are by default placed before the $2$ m2.micro receivers. We run two different rounds with and without using the algorithm proposed in Sec. 5 that place the receivers into the topology by their CPU performance. We show the distribution of the dissemination time by repeating each round by $100$ times. As shown in Fig. 16, we can see that with the algorithm, the dissemination time can be improved by $0.88 - 1.34$ seconds. In addition, we can also observe the dissemination becomes more stable than without using the algorithm. This is because that with lower overload, the two receivers with low CPU performance will also work more stable as they only need to forward data to the receivers in the next level.

## 8 Related Work

Erasure codes have demonstrated both advantages and disadvantages to store data inside data centers. One of its most prominent advantages is to protect data against failures with much lower storage overhead than replications [17]. As $N$-way replication requires to store $N$ copies of the original data, many distributed storage systems [5], [6], [7] have deployed or been in a transition to deploy erasure codes inside data centers.

Though different kinds of erasure codes with different system parameters are deployed in distributed storage systems, there exists a significant bottleneck when erasure-coded data are to be written into the system. For example, in Windows Azure storage [5], when data are to be written as erasure codes, a server will be appointed as the source which computes all coded data for all servers that will actually store the coded data. Then all coded data will be disseminated from the source into corresponding servers. Even though deploying different erasure codes, all current distributed storage systems store erasure-coded data, follow such a manner [4], [5], [7], [18]. Naturally, this manner will lead to an increasing of the time to write data into the system with the number of receiving servers.

Distributed storage systems that store replications in the data center do not suffer from this problem as a pipelining method can be used to alleviate the bottleneck at the source. For example, Google File System (GFS) [1] and Hadoop File System (HDFS) [2] applies this method to build a pipeline in which receiving servers can relay data to the next one while receiving data. This method, however, cannot be applied to write erasure-coded data into the data center.

To solve this problem, some specific erasure codes have been proposed to build similar pipelines. RapidRAID [19], for example, is a family of erasure codes which can be built into a pipeline, so as to progressively generate coded data in a decentralized manner. This leads to a constraint on the constructed codes. For example, RapidRAID codes are not systematic. Thus, to read the actual content from coded data, the storage system that deploys such codes will have to decode the coded data first, leading to a limited throughput of read operations. Juarez *et al.* [20] focus on the reduction of network traffic instead of the time with the pipeline. These two kinds of erasure codes also require that some servers have already stored replicated data and they are both designed to disseminate coded data from existing replications.

Another solution [15] creates a pipelined topology to write data into servers. However, it still supports only one kind of erasure codes with limited choices of parameters, and also compromise the failure tolerance. Different from these works, this paper proposes and implements a new pipelined mechanism to disseminate coded data with general erasure codes, with no requirement of existing replications, that saves the time to disseminate erasure-coded data significantly.

To work universally with different distributed storage systems, *Mist* needs to support different kinds of erasure codes for distributed storage systems in data centers. Apart from traditional erasure codes such as Reed-Solomon codes, existing erasure codes designed for distributed storage systems can be roughly classified into two categories: locally repairable codes [4], [5], [18] and regenerating codes [11], [12]. They are proposed to reduce the consumption of disk I/O and network traffic in the process to repair any lost coded data inside data centers, respectively. *Mist* does not rely on any specific kind of erasure codes, and offers a mechanism to support different kinds of erasure codes in general, with no constraint on their parameters. Moreover, we support one family of erasure codes in each of these two categories, *i.e.*, local reconstruction code [5] and minimum-storage regenerating codes [8].

## 9 Conclusion

In this paper, we have designed, implemented, and evaluated *Mist*, a new mechanism for disseminating erasure-coded data to multiple servers in a data center. *Mist* alleviates the conventional bottleneck at the source server in such dissemination processes, by building a pipelining topology where receiving servers can be allowed to compute their desired data from other receivers, rather than from the source server directly. It provides a flexible general topology to support different kinds of erasure codes, yet with no constraints on the system parameters. As examples,

we have designed specific topologies of two representative types of erasure codes for data storage inside data centers, taking advantage of their corresponding properties. *Mist* can even disseminate erasure-coded data along with replications efficiently. We have implemented *Mist* in Python, and our experimental results in Amazon EC2 have shown that, with the topologies it constructed, the time to disseminate erasure-coded data to multiple servers can be significantly saved, and becomes much less sensitive to the number of receivers.

## ACKNOWLEDGMENT

We would like to thank our editor and anonymous reviewers for their efforts on improving the quality and the presentation of this paper. The research was support in part by RGC GRF grants under the contracts 16211715 and 16206417, a RGC CRF grant under the contract C7036-15G.

## REFERENCES

[1] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," in *Proc ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "Hadoop Distributed File System," in *IEEE Mass Storage Systems and Technologies (MSST)*, 2010.

[3] P. Vagata and K. Wilfong, "Scaling the Facebook data warehouse to 300 PB," 2014, https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/.

[4] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," in *Proc. VLDB Endowment*, 2013.

[5] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *Proc. USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[6] W. Wang and H. Kuang, "Saving capacity with HDFS RAID," 2014, https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/.

[7] Z. Zhang, A. Wang, K. Zheng, U. M. G., and V. B, "Introduction to HDFS Erasure Coding in Apache Hadoop," *Cloudera Engineering Blog*, Sept. 2015, https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.

[8] K. Rashmi, N. Shah, and P. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction," *IEEE Trans. on Inform. Theory*, vol. 57, no. 8, 2011.

[9] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage," in *Proc. 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[10] W. Wang, "Facebook's Approach to Big Data Storage Challenge," *Hadoop Summit*, 2013, https://www.youtube.com/watch?v=OGZpqkKntkQ.

[11] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. Inform. Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[12] A. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A Survey on Network Codes for Distributed Storage," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 476–489, Mar. 2011.

[13] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *ACM SoCC*, 2012.

[14] "Intel Storage Acceleration Library," https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.

[15] L. Pamies-Juarez, A. Datta, and F. E. Oggier, "In-network Redundancy Generation for Opportunistic Speedup of Data Backup," *Future Generation Comp. Syst.*, vol. 29, no. 6, pp. 1353–1362, 2013.

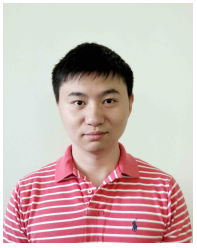
[16] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *; login:: the magazine of USENIX & SAGE*, vol. 35, no. 2, pp. 6–16, 2010.

[17] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *Proc. International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[18] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "HitchHiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers," in *Proc ACM SIGCOMM*, 2014.

[19] L. Pamies-Juarez, A. Datta, and F. E. Oggier, "RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems," in *Proc. IEEE INFOCOM*, 2013, pp. 1294–1302.

[20] L. Pamies-Juarez, F. E. Oggier, and A. Datta, "Decentralized Erasure Coding for Efficient Data Archival in Distributed Storage Systems," in *Proc. International Conference on Distributed Computing and Networking (ICDCN)*, 2013, pp. 42–56.



**Jun Li** received his Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2017, and his B.S. and M.S. degrees from the School of Computer Science, Fudan University, China, in 2009 and 2012. He is currently an assistant professor in the School of Computing and Information Sciences, Florida International University. His research interests include erasure codes and distributed storage systems.

**Baochun Li** received his Ph.D. degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 2000. Since then, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. He is a member of the ACM and a fellow of the IEEE.

**Bo Li** is a professor in the Department of Computer Science and Engineering. Hong Kong University of Science and Technology. He has been the Chief Technical Advisor for Chinacache Corp. (a NASDAQ listed company), the leading CDN operator in China since 2008. He held a Cheung Kong Visiting Chair Professor in Shanghai Jiao Tong University (2010-2013) and an adjunct researcher in Microsoft Research Asia (1999-2007) and in Microsoft Advance Technology Center (2007-2009). His current research interests include: datacenter networking, cloud computing, content distribution in the Internet, and mobile wireless networking.

He made pioneering contributions in the Internet video broadcast with a system called Coolstreaming, which was credited as first large-scale Peer-to-Peer live video streaming system in the world. This work appeared in IEEE INFOCOM (2005) received the inaugural The Test-of-Time Paper Award from IEEE INFOCOM (2015). He has been an editor or a guest editor for over a dozen of IEEE journals and magazines. He was the Co-TPC Chair for IEEE INFOCOM 2004.

He received six Best Paper Awards from IEEE. He received the Young Investigator Award from Natural Science Foundation of China (NFSC) in 2005, the State Natural Science Award (2nd Class) from China in 2011. He is a Fellow of IEEE. He received his B. Eng. (summa cum laude) in the Computer Science from Tsinghua University, Beijing, and his PhD in the Electrical and Computer Engineering from University of Massachusetts at Amherst.