R-Drive: Resilient Data Storage and Sharing for Mobile Edge Computing Systems

***Abstract*—Mobile edge computing (MEC) systems (in which intensive computation and data storage tasks are performed locally, due to absence of communication infrastructure for connectivity to cloud) are currently being developed for disaster response applications and for tactical environments. MEC applications for these scenarios generate and process significant mission critical and personal data that require resilient and secure storage and sharing. In this paper we present the design, implementation and evaluation of R-Drive, a *resilient* data storage and sharing framework for disaster response and tactical MEC applications. R-Drive employs erasure coding and data encryption, ensuring resilient and secure data storage against device failure. R-Drive adaptively chooses erasure coding parameters to ensure highest data availability with minimal storage cost. R-Drive's distributed directory service provides a resilient and secure namespace for files with rigorous access control management. R-Drive leverages opportunistic networking, allowing data storage and sharing in mobile and loosely connected edge computing environments. We implemented R-Drive on Android, integrated it with existing MEC applications. Performance evaluation results show that R-Drive enables resilient and secure data storage and sharing.**

I. Introduction

Mobile Edge Computing (MEC) has gained significant popularity over traditional cloud computing due to low latency guarantee for data storage and processing. In this architecture, devices form a local cloud using available computing and storage resources, allowing applications to process and store data locally [1], [32], [44], [27], [34]. Edge computing platforms that are designed for mobility need to handle disconnected environments where infrastructure networks such as cellular and Wi-Fi are unavailable and cloud services are unreachable [28], [15], [24]. In such cases, MEC applications are entirely dependent on available edge resources for operations. Disconnection-tolerant MEC platforms for disaster response and wide area search and rescue operations are gaining significant popularity [9], [13], [36]. In such scenarios, first responders are equipped with necessary hardware including a manpack, mobile devices, wearable gadgets, sensors etc., to perform mission critical operations (Figure 1).

Devices employed in MEC, e.g., on-body cameras, smart watches, gesture recognition devices, body sensors (heat, gas, water etc.) as well as MEC applications on mobile devices generate large amounts of mission critical data and perform storage intensive tasks. Storage intensive tasks such as urban sensing, survey collection, geo-spatial data collection, text and media files storage and any other quantitative and qualitative data storage by users etc., require *resilient data storage* with low overhead [10], [33]. Device failure can occur due to hardware malfunction and battery depletion due to heavy use

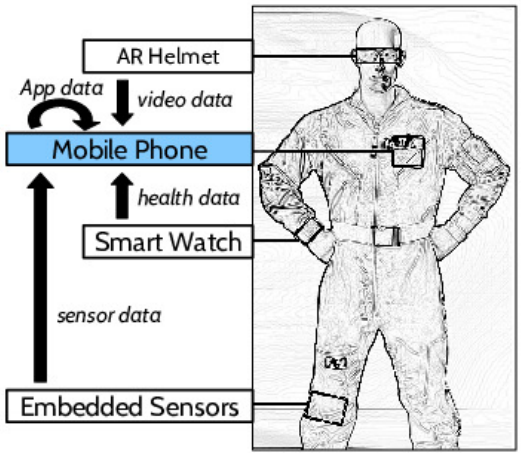


Fig. 1: Next generation first responders equipped with wearable technologies including AR helmets, mobile phones, smart watches and embedded sensors, which generate large amounts of data that require resilient storage for processing [29]

of edge devices. Hence, data storage in mobile edge must employ replication based distributed storage so that data is not lost due to device failure. Additionally, *disconnection resilient data sharing* among entities in MEC is difficult due to absence of infrastructure networks and frequent device mobility. In a search and rescue scenario, since first responders perform their respective tasks being agnostic of network connectivity, data sharing among entities needs to be network disconnection tolerant.

Existing file/data storage services, e.g., Dropbox [16], Google Drive [20], OneDrive [26] etc. are not designed for MEC and cannot operate in the absence of cloud. Although these services allow users to store and modify files offline, the files are simply stored locally, making them prone to data unavailability/loss due to device failure by energy depletion or hardware malfunction. Moreover, mobile devices at the edge are prone to frequent disconnection/separation from the network, due to mobility or network congestion. Thus, local updates may not propagate to the cloud despite intermittent cloud access. Thus, there is a need for a cloud-like data storage service at the edge that uses available edge resources for storing data.

Existing storage services also enable *data sharing* among devices. This can only be accomplished through the cloud via infrastructure networks. Users may employ data sharing applications that make use of ad-hoc network connectivity

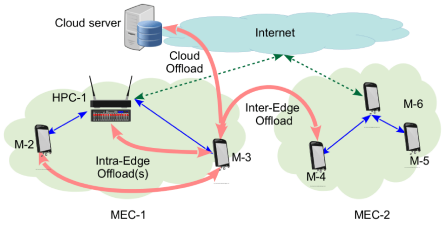


Fig. 2: Mobile devices form two mobile edge networks (MEC-1 and MEC-2) and share resources among themselves, or with the cloud, to perform data storage/sharing and processing tasks.

(e.g., Bluetooth, Wi-Fi Direct). But, disconnection may occur during a data sharing session; thus, users are required to minimize movement and stay connected until the data sharing session completes. Hence, traditional data sharing schemes are impractical for disconnection oriented mobile edge.

To address the aforementioned limitations, we designed and implemented *R-Drive*, a data storage and sharing framework for MEC environments. *R-Drive* handles both device and network failures in MEC environments, eliminating the above mentioned data storage and sharing problems by bringing cloud services to the edge. *R-Drive* utilizes available storage resources of devices to the edge to resiliently store data and allows users/applications to securely share them with proper access control. The key features of *R-Drive* and the contributions of the paper are as follows:

- *R-Drive* employs distributed data storage with encryption and erasure coding, enabling resilient data storage against device failure.
- *R-Drive* employs opportunistic networking, maximizing the use of available bandwidth, at the same time abstracting network failure from client applications.
- *R-Drive* incorporates resilient distributed directory service with secure access control.
- *R-Drive* transparently enables existing data storage applications to share data without assistance from the cloud.

II. Background and State of the Art

A. Mobile Edge Computing for Disaster Response and Tactical Environments

Figure 2 depicts a MEC architecture for disaster response or tactical environments. Multiple mobile devices form an edge network that can be disconnected from the Internet and cloud servers. Mobile devices may be connected to a *HPC node* that manages communications (e.g., LTE, Wi-Fi, Wi-Fi Direct), IP addresses allocations to devices, and DNS services, mapping device names to their corresponding IPs. The central node also performs high-performance computing (HPC) functions on data produced by edge devices. In addition to the HPC node, data can also be offloaded to connected mobile devices for processing and storage. As shown in the figure, two edge networks (where nodes HPC-1 and M-6 serve as central nodes for edges 1 and 2, respectively) can be connected over Internet, or they can discover each other locally.

Fig. 3: Anonymous - a MEC platform for disaster response consisting of: a) LTE antenna, b) Wi-Fi AP, c) LTE eNB, d) Intel NUC that runs LTE EPC and HPC; e) Battery; f-g) Helmet with body camera; h) Android phones

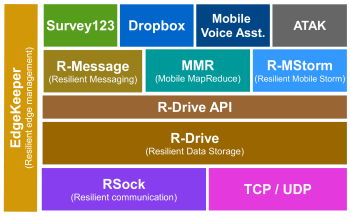


Fig. 4: Software ecosystem for the Anonymous mobile edge computing suit

An example of a MEC system for disaster response is shown in Figure 3. The system Anonymous , consists of a manpack equipped with wireless communication (LTE and Wi-Fi) and processing capabilities (HPC). Multiple mobile devices communicate among themselves and with the manpack using wireless communication, for sharing data storage and computational resources. The software architecture for Anonymous is shown in Figure 4. As shown, several applications have been specifically designed for MEC. For example, R-MStorm (Resilient Mobile Storm) [11] is for real-time stream processing, MMR (Mobile Map Reduce) for batch processing, a mobile virtual voice assistant for emergency medical services, edge resource orchestration, etc. Survey123 [5] is used by several disaster response teams (e.g., Texas Task Force) to gather field data (e.g., number of survivors, hazardous locations etc.) and send it to a database located either in the cloud or a local server. Survey123 can operate in completely disconnected environments, however, in such environments, the data is cached locally on mobile devices and only uploaded once a cellular or Wi-Fi connection to the Internet is established. ATAK [14] is an Android application used by the US military to share mission critical data during combat missions or disaster response operations. Similar to Survey123, ATAK stores data on local storage if communication with a master ATAK server is unavailable.

As shown in Figure 4, two important components of the Anonymous architecture are EdgeKeeper and RSock (Resilient Socket). **EdgeKeeper** [8] is a distributed coordination and service discovery, and meta-data storage application which runs on all devices of the edge. EdgeKeeper is based on

a primary/master-replica/slave architecture in which at one time one device acts as a master, whereas other devices act as slaves. EdgeKeeper master is responsible for maintaining distributed consensus among devices and providing services such as device authentication, service discovery, edge health monitoring, network topology management, metadata store etc. EdgeKeeper slaves are standby devices to take over dead master and maintain services. EdgeKeeper employs *Globally Unique Identifier* (GUID) [41] to uniquely identify each edge device. Each GUID is a unique 40 characters long string, generated with a unique public and private key pair, assigned to one user. EdgeKeeper is responsible for performing DNS-like GUID to/from IP mapping. GUID based device identification allows applications on different devices to communicate with each other being agnostic of IP assignments. Such identification scheme enables mobile devices pertaining to different edges to communicate, and perform inter-edge tasks. **RSock** [2] is a resilient transport protocol designed for sparsely connected network environments aiming to make best use of available network bandwidth and to ensure timely data delivery. RSock provides routing by GUID and replication of packets, to be sent over multiple paths for device-to-device communication (i.e., using any available wireless interface - LTE, Wi-Fi, Wi-Fi Direct). RSock employs the Hybrid Routing Protocol (HRP) [47], which performs packet replication to reduce the packet delivery delay. A RSock header contains a sequence number for the corresponding packet so that receiver can assemble the packet in its respective position. For a file to be received successfully, all RSock packets must have to be received and assembled by their packet numbers at the receiver. RSock API allows user to set a time to live (TTL) value for a packet in the header. The TTL value entails for how long (in seconds) a packet can be alive in the network. If the TTL for a packet expires, the packet will be immediately discarded from the network.

B. Motivation and State of the Art

Applications in MEC platforms for disaster response generate significant amounts (e.g., gigabytes) of mission critical and personal data that require resilient and secure storage. As examples, R-MStorm generates media data about disaster victims, Mobile Voice Assistant collects patients' personal medical information, Survey123 collects various survey data during search and rescue operations. Currently, this data is stored only on a device's local storage; if a device's storage runs out, these applications cannot store new data. Also, if the device fails, the data stored on it is lost or inaccessible. Existing data storage services such as Dropbox, Google Drive, OneDrive etc. *require cloud connectivity to store data.* During large scale disasters, infrastructure networks such as LTE, Wi-Fi, etc. may not be available, hence above services can not be used for storing data. Moreover, device failure may occur due to energy depletion or hardware malfunction. Hence, *data is vulnerable to loss/unavailability* if stored on a single device without added resilience. Pure data replication across devices to ensure resilience against device failure is not a feasible solution for mobile edge due to limited storage availability. Furthermore, some storage services store offline files in local storage without any protection (i.e., encryption), allowing *data tampering by injecting corrupt data* by malicious applications.

We conducted experiments with above mentioned storage services and observed that these services do not provide both data resilience and security when storing locally. We stored large amount of offline data and observed that, when device storage runs out, these services become inoperational, despite other devices in same network having large amounts of available storage. We conducted another set of experiments in which we tampered offline data of Dropbox and Survey123 by injecting malicious data and observed that corrupted updates were later propagated to cloud when applications were started. Google Drive and OneDrive do not store offline files in external storage, hence they are not directly accessible via Android filesystem. But, the offline files are still vulnerable of device failure as they are stored in single device. Table I summarizes the limitations of existing storage solutions, as well as storage intensive MEC applications.

Services	**Cloudless Storage**	**Resilient Storage**	**Offline Encrypt**
Dropbox	×	×	×
OneDrive	×	×	✓
Google Drive	×	×	✓
Survey123	×	×	×
R-MStorm	✓	×	×
R-Drive	✓	✓	✓

TABLE I: Existing storage services and MEC applications rely on cloud connectivity for data storage.

To further address the limitations, we investigated state of the art distributed storage and file system solutions for mobile edge. CODA [23], maintains a local cache during disconnected operations to store edited data and requires cloud connectivity to synchronize local cache with replica servers. OFS [30], carries heavy storage overhead since it keeps a copy of the same data to both local device and cloud to ensure data availability. MEFS [38] tried to retro-fit a cloud based file system to use for mobile edge, but the solution greatly relied on cloud communication. PFS [17], FogFS [31] rely in specific mobility models that makes them impractical for disconnected mobile edge. Although HDFS [43] and GFS [18] use erasure coding instead of replication to store data in replica devices, these solutions are too heavy-weight for mobile devices in terms of memory footprint and computation. Hyrax [25] tried to port HDFS for Android devices and experimented in mesh networks. Despite decent engineering efforts, Hyrax showed poor performance for CPU bound tasks. MDFS [12] implementation was based on a purely connected network, which is a major fallacy in disconnected mobile edges. MDFS did not provide a file system-like functionality, such as directory service, access control management etc.

As mentioned earlier, infrastructure networks may be unavailable during large scale disasters. Existing services such as Dropbox, Google Drive, OneDrive etc. can only *share data across devices if there is connectivity to the cloud.*

Services	Cloudless Share	Opportunistic Share	Cloudless Namespace
Dropbox	×	×	×
OneDrive	×	×	×
Google Drive	×	×	×
Share Apps	×	×	×
R-Drive	✓	✓	✓

TABLE II: Existing data sharing services cannot share data without cloud connectivity.

Users can use file sharing applications (Google Files [21], SHAREit [40] etc.) that do not require cloud connectivity and can operate over ad-hoc networks such as Wi-Fi Direct, Bluetooth, NFC etc. But, ad-hoc networks rely on short range communication and constant connectivity. During large scale search and rescue operations, team members are mobile and scattered across large areas; it may not always be possible for two team members to stay within each other's communication range to exchange data. For instance, two team members may not be directly connected yet reachable via one or many intermediate mediums/people that frequently travel back and forth between them. Consequently, MEC platforms for disaster response should support *network opportunistic data sharing over multiple hops.* Moreover, during disaster response operations, first responders are divided into groups to perform their respective tasks. Team members often need to share mission critical data among themselves to co-ordinate their tasks. Also, sharing data with other teams require rigorous access control so that critical data is only shared with authorized personnel (e.g., team leaders). Existing data sharing services cannot sync directories across devices in absence of cloud connectivity. Consequently, first responder teams need to have a *common namespace to manage data and permissions* that does not rely on cloud connectivity. Table II summarizes the limitations of the existing data sharing schemes in disconnection oriented MEC platforms.

To apply erasure coding for data storage, two important input parameters are n and k. A high n and low k increases data availability at a cost of storage, and vice-versa. (n, k) should be decided dynamically depending on the edge resource availability and user provided quality of service (QoS) parameters. Although HDFS [49], GFS [18] use erasure coding for distributed storage in a cluster, the choice of parameters for erasure coding (n, k) is fixed, since both HDFS and GFS were designed primarily for large storage clusters consisting of hundreds of storage nodes residing in stationary racks, in a data-center. MDFS [12] incorporated erasure coding for disconnection prone mobile edge, and took network link quality, energy cost etc. in consideration. But they did not provide any online algorithm to select n and k values for variable storage availability and file sizes. Zhu et al., presented an online adaptive code rate selection algorithm for cloud storage [50]. The algorithm takes real-time user demands as one of the input metrics to a regret minimization problems to decide the most optimum n and k values. The solution states as one of their assumptions is that all the candidate

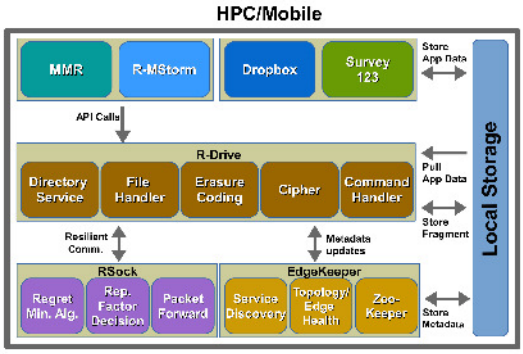


Fig. 5: *R-Drive* components and integration with the Anonymous software ecosystem

storage devices have enough storage capabilities, which can be a big assumption for mobile edge computing environment where devices are heterogeneous. HACFS [46] is another novel solution, where authors implemented an extension to HDFS to adaptively choose between fast code and compact code depending on data read hotness. The solution also up-codes and down-codes previously encoded data to ensure data resiliency against loss due to various reasons. Despite having the capability of switching between two coding schemes, their solution involves using fixed coding parameters for each of the coding schemes. Zhang et al., proposed an erasure coded storage system consisting Android devices, also provided no study for choosing the most efficient nodes and n, k values [48]. Shu et al., proposed an erasure coded distributed storage system on fog nodes, but provided no analysis as to how to choose the n, k values [42].

III. R-Drive System

A. R-Drive System Overview

1) System Components: Figure 5 shows the *R-Drive* system components and its integration with the Anonymous software ecosystem. As shown, *R-Drive* consists of five major components. **Directory Service** provides a namespace for all files and directories. **File Handler** performs file and directory operations (e.g., creation, retrieval and removal). **Erasure Coding** component encodes and decodes data into fragments using Reed-Solomon erasure coding [35]. **Cipher** encrypts and decrypts data using 256 bit AES encryption. **Command Handler** handles commands for basic storage operations such as *-put, -get, -mkdir, -ls, -rm* etc.

Client applications such as MMR, R-MStorm use the *R-Drive* API to perform data storage and sharing operations. Applications such as Dropbox, Survey123 etc. generate and store *app data* on device's local storage. *R-Drive* periodically fetches the *app data* and stores it in *R-Drive* for resilience against device failure. *R-Drive* communicates with RSock and EdgeKeeper applications via JSON based RPC calls over local TCP sockets.

2) Data storage in R-Drive: Storage in *R-Drive* takes place via *R-Drive* user interface (UI) or Java client API.

Client applications can make appropriate *R-Drive* API calls to perform storage and sharing tasks. *R-Drive* UI allows a device operator to directly interact with the application. *R-Drive* also provides command line interface (CLI) that allows a desktop user (e.g., a search and rescue operation coordinator) to connect to a device in the field and perform *R-Drive* operations with permissions. The *R-Drive* API is as follows:

```
int mkdir(String rdriveDirectory,
            List<String> permissionList);
List<String> ls(String rdriveDirectory);
int put(String localFilePath,
            String rdriveFilePath,
            List<String> permissionList);
int get(String rdriveFilePath,
            String localFilePath);
int rm(String rdriveFilePath);
```

R-Drive also allows data storage by monitoring files in user-selected directories on local storage, similar to Google Backup and Sync [19]. A user selects a directory in the local storage that *R-Drive* will monitor for any changes in data and periodically pulls the data to store in *R-Drive* system. User can select application directories which are prone to data loss due to device failure. Currently, *R-Drive* supports backing up *app data* for Survey123, ATAK, Dropbox.

3) Data sharing in R-Drive: R-Drive enables inter device data sharing using RSock communication channel. A device can share data to one or multiple devices as follows: a) unicast the data to any device; b) upload the data to *R-Drive* system and set permission appropriately for authorized users. In scenario (a), no data erasure coding takes place and the entire file is sent. The TTL value in RSock is set appropriately, to decrease the likelihood of data being discarded during routing, but to also not congest the network (by keeping too many copies of the data).

B. Directory Service

The Directory Service provides all metadata-related operations such as creation of new metadata, retrieval of existing metadata, checking metadata permissions, presenting a namespace to clients etc. *R-Drive* maintains a hierarchical directory structure; the top level directory is root(*/*) and below are subsequent subdirectories. A metadata in *R-Drive* system is called an **rnode**, with its structure shown in Table III). A rnode represents either a file or a directory entity in *R-Drive*. After creating a rnode, the Directory Service stores it in EdgeKeeper, which internally stores it in ZooKeeper data nodes, commonly known as znode. ZooKeeper replicates znodes to replica devices for fault tolerance, to handle master failure or cluster disconnection. If one or more replica devices leave the edge, other edge devices become new replicas and the lost znodes are replicated to the new replica devices. Consequently, if EdgeKeeper has r replicas, then *R-Drive* metadata remains intact and it can provide Directory Services despite device failures as long as there are $\lceil r/2 \rceil$ devices available at the edge.

TABLE III: *R-Drive* rnode structure

Field	Size	Description
rnodeType	1 Byte	File or Directory rnode
rnodeID	16 Bytes	Unique rnode ID
fileName	Variable	Original File Name
fileSize	8 Bytes	Original File Size
fileID	16 Bytes	Unique File ID
filePath	Variable	R-Drive File Path
N	2 Bytes	N value for EC
K	2 Bytes	K value for EC
blockCount	2 Bytes	Number of Blocks
fragLocation	Variable	locations of fragments
fileList	Variable	List of Files
folderList	Variable	List of Subdirectories
permission	Variable	Access Control List
timeStamp	8 Bytes	Time of Creation

A directory creation takes place when a client invokes the *mkdir()* API function or when the command *-mkdir* is executed in the CLI. The Directory Service creates a new rnode for the target directory. Directory Service then fetches a copy of immediate parent rnode of the target directory from EdgeKeeper and inserts the target rnode information in the parent rnode. Finally, Directory Service pushes both parent and target rnodes to EdgeKeeper. EdgeKeeper stores the rnodes in ZooKeeper as data nodes. Directory retrieval is initiated when a client invokes the *get()* API function or when the command *-ls* is executed in the CLI.

1) Access Control Management: R-Drive leverages ZooKeeper's access control for managing permissions for rnodes. ZooKeeper [4] supports pluggable authentication schemes. *R-Drive* implements its own custom authentication scheme as part of the Directory Service. *R-Drive* follows the standard UNIX permission scheme which can be set during file or directory creation. Permissions can also be set via the *-setfacl* and *-getfacl* commands entered through the CLI. A file or directory creator can set permission for any rnode for OWNER, WORLD, or a list of GUIDs. Each rnode permission only pertains to itself and does not apply to children.

C. R-Drive Data Storage

All data is stored in *R-Drive* as files. File creation involves copying a file from local file system to *R-Drive* using the *put()* API function or the *-put* command. The File Handler loads the target file and divides it into fixed sized blocks. Each block is then encrypted with a unique secret key and later converted into n fragments using erasure coding. Directory Service communicates with EdgeKeeper to push the rnode for newly created file. If the rnode update succeeds, all fragments are sent through RSock by invoking the RSock client API. All fragments contain a timestamp that acts as a version number for fragments. A receiver device only accepts fragments with same or higher timestamps. Figure 6 shows the steps for a file creation process in *R-Drive* .

1) R-Drive Data Encryption: R-Drive uses 256 bit AES encryption using a unique secret key for file encryption. The key is further divided into B key-shards using Shamir's Secret Sharing Scheme (SSSS) [39]. SSSS is a distributed secret sharing scheme in which a secret is divided into shards in

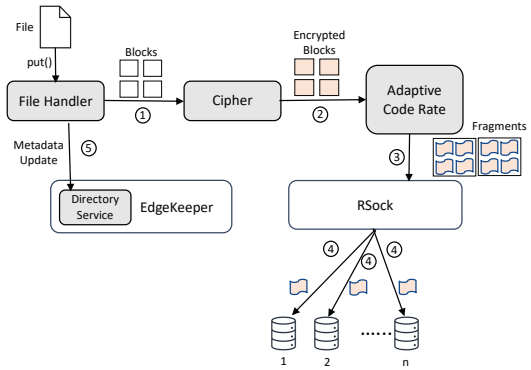


Fig. 6: *R-Drive* file storage steps: partitioning the file into blocks, encrypting them, applying the adaptive erasure coding and distributing the fragments to best suitable n nodes

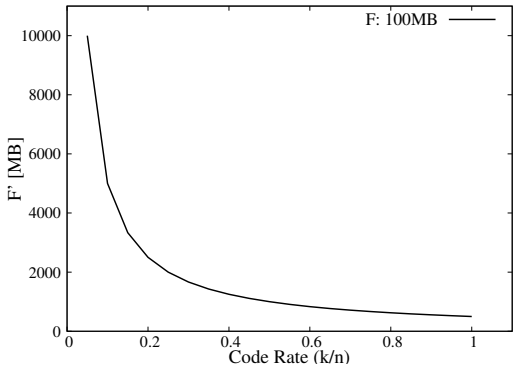


Fig. 7: File size F' after erasure coding (applied to a file F of size 100MB) as a function of the code rate

such a way that individual shards cannot reveal any part of the secret, whereas an allowed number of shards put together can reveal the secret. (T, N) is the conventional way to express the SSSS system, where N is the total number of secret shards, and T is the minimum number of shards required to unveil the secret. In *R-Drive* we used (B, B) as parameters for SSSS, where B is the number of blocks.

2) Resilient Storage through Erasure Coding: R-Drive uses Reed-Solomon erasure coding for data redundancy [35]. Erasure codes are forward error correction (FEC) techniques that take a message of length M and convert it into coded message of length greater than M by adding redundancy so that the original message can be reconstructed by a subset of the coded message. In *R-Drive* storage, a file of size F is divided into k fragments, each of size F/k. Applying (n, k) encoding on k fragments will result in n fragments, each of size F/k, where $n \geq k$. Hence, the total file size will be $n \cdot F/k$. Encoded n fragments are then stored in geographically separated storage devices. To reconstruct the file, any k fragments are sufficient. Thus, the system tolerates up to $n - k$ device failures.

3) Adaptive Code Rate: The most widely used erasure coding library is Reed-Solomon that takes (n, k) as parameters. The choice of n and k values are directly related to data redundancy (hence availability) at cost of additional storage overhead. So, choosing devices that has enough available storage is the basic requirement for storing data in mobile edge. Also, devices in edge are prone to device failure due to energy exhaustion, hardware failure, etc. So, choosing the devices that has more chance of survival against device failure is also an important factor to consider. Hence, in summary, the problem statement is, how to choose the best n and k values, and the fittest n nodes (in terms of available battery life, storage capacity etc) so that the entire edge system can achieve highest data availability for the least storage cost.

The ratio k/n in erasure coding, or the ***code rate***, indicates the proportion of data bits that are non-redundant. As a rule of thumb, when code rate decreases, the file size after erasure coding increases, and vice-versa. But, at the same time, lower code rate usually comes with a higher n and lower k values, providing added fault tolerance to the data. So, we cannot simply choose the lowest possible code rates; in that case, we will exhaust the system storage capacity very rapidly. Figure 7 shows the file size after erasure coding as a function of code rate to illustrate the fact that erasure coded file size increases exponentially with decreasing code rate.

We need an online algorithm that dynamically chooses the (n, k) values, and the fittest n nodes for file storage in the edge. The algorithm's main focus will be to incorporate edge specific parameters (remaining battery, available storage, user/file specific quality of service parameters etc) to decide n and k values to optimize between data availability and storage cost. To reduce complexity, we will avoid all Reed-Solomon library specific parameters and use the default values for them.

To enable erasure coding in *R-Drive*, we need to answer the following: *1) What code rate and what (k, n) pair should the system choose? 2) Given a chosen code rate and (k, n) value pair, which specific n devices should the system store the n file fragments to? 3) What system parameters used in answering 1) and 2) will be collected, and how?*

Q1: What k and n values? Code rate is calculated as k/n. If k/n is small, there is a high probability to recover a file because only a small number of file fragments are required to reconstruct the original file. The file size after erasure coding with code rate k/n is calculated as $F' = F * n/k$, where F represents the original file size. In this case, if k/n is too small, n/k becomes very large, then the encrypted file size F' becomes very large as well. Small (k, n) entails higher file availability at a cost of larger storage overhead for the entire *R-Drive* system. To address this trade-off, we present the cost of availability and storage C as a weighted sum and formulate the problem as a minimization problem as follows:

$$\underset{(k,n)}{\text{minimize}} \quad C(k, n, w_a) = w_a * k/n + (1 - w_a) * n/k \quad (1)$$

$$\text{subject to:} \quad F/k \leq S_n, \quad (2)$$

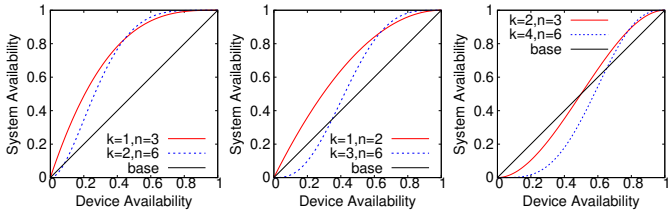
$$T \leq T_k, \quad (3)$$

$$1/N \leq k \leq n \leq N, k, n \in Z^+ \quad (4)$$

$$0 \leq w_a \leq 1 \quad (5)$$

where w_a denotes the weight of availability cost, $1 - w_a$ the weight of storage cost, S_n the n^{th} maximum available storage of all nodes, T_k denotes the k^{th} longest remaining time among the total available N devices, T denotes the minimum time that a file is expected to be available in *R-Drive*. In the minimization problem, constraint (2) ensures that the storage allocation for a node does not exceed available storage, constraint (3) ensures that only devices with enough battery (for the selected lifetime T of a file) are selected, constraint (4) ensures that only positive n and k are selected, in the range $[1/N, N]$.

The weight w_a is adjusted adaptively for different files; for a critical file, the system sets a large w_a so that a small k/n is chosen to improve its availability; for a large but unimportant file, the system sets a small w_a so that a small n/k is chosen to reduce the total storage cost.



(a) k/n=1/3 (b) k/n=1/2 (c) k/n=2/3

Fig. 8: Examples showing how different (k, n) pairs determine different system availability. The example contains three small groups (a, b, c) and each group contains two (k, n) pairs of same ratio. The baseline in each group represents pure local storage

Since both k and n need to be integers, we can easily solve the above minimization problem by iterating over all possible (k,n) pairs and choose those with the minimum costs as solutions. The time complexity of this method is $O(N^2)$. However, there are sometimes several (k, n) pairs with the same minimum costs. To further select among these (k, n) pairs, we need a more precise method to depict the system availability. For simplicity, we assume each device has the same availability p. Then, the system availability can be calculated as follows:

$$A(k, n, p) = C_k^n p^k (1-p)^{(n-k)} + ... + C_n^n p^n \qquad (6)$$

where C_k^n denotes the number of ways for choosing k from n devices. This equation is complex. In order to get an intuitive understanding of it, we show a few simple examples in Figure 8, where we compare the system availability for 6 (k, n) settings calculated based on the above equation. We further divide these six settings into three groups. Within each group, the core rate k/n is identical. As we can see, when the erasure rate increases from $1/3$ to $1/2$ then to $2/3$, the system availability gradually decreases. This indicates the rationality of representing the system availability with k/n for simplicity in Equation (1). Meanwhile, we observe that, in each group, when the device availability p is small, although the k/n values of different settings are the same, the (k, n) setting with a smaller n has higher availability than the other with a bigger n. However, as the device availability p gradually increases over a threshold, the setting with a bigger n starts to achieve higher system availability than the setting with a smaller n. Therefore, what (k, n) to choose for a specific k/n is determined by the device availability p.

In *R-Drive*, for simplicity, we calculate the availability p_i of device i as follows:

$$p_i = \begin{cases} 1, & T_i \geq T \\ T_i/T, & 0 < T_i < T \end{cases} \qquad (7)$$

where T_i represents the remaining time of device i. When *R-Drive* selects between (k_1, n_1) and (k_2, n_2) with the same k/n values, it first calculates the value of $A(k_1, n_1, \overline{p})$ and $A(k_2, n_2, \overline{p})$, where $\overline{p}$ represents the average availability of devices, and then chooses the one with a larger value.

Q2: Which specific n devices? After deciding (k, n), the next question to answer is which n devices to store the n file fragments. *R-Drive* adopts a simple strategy for this issue. First, it chooses all devices with the remaining storage space larger than F/k. Next, it sorts the picked devices based on the expected remaining time in descending order. Finally, it chooses the top n devices with the longest remaining time to store the n file fragments. The complete algorithm for choosing (k, n) and specific n devices is given in Algorithm 1.

Q3: How are algorithm input parameters decided? Here we provide a general recommendation for setting w_a and T before data storage tasks are initiated. w_a and T are not meant to be changed for every file; instead, user should set particular values for w_a and T for a particular collection of data. w_a and T values should be set based two factors - how important/mission critical the file is, and how soon user is expected to access/read the data. As an example, for mission critical data such as victim personal image/video files, w_a can be set high such as 0.8, 0.9, 1.0 etc. Also, if user is expected to access the stored data in a near future, user can set an approximate T, and the algorithm will choose at least k candidate devices with at least T battery remaining time. Since the algorithm outputs n and k, which are integers, fine tuning w_a may not always have impact on the output. Hence, we recommend choosing w_a as a multiple of 0.1.

4) Cost Function Lower Bound : Figure 9 shows how the choice of code rate impacts the cost function for different w_a. We identified that for each w_a, there is a code rate for which the cost is the lowest, which is the optimal cost. The algorithm tries to reach towards the optimal cost, regardless of the selection of n and k values. For a particular (n, k), if the code rate is similar to the optimal cost code rate, the algorithm will try to hold onto this particular (n, k), unless the devices do not check out storage and battery remaining time requirements (as mentioned in equation 1). Table IV shows the optimal cost for variable w_a and the code rates for which the optimal cost is achieved.

A natural question may arise, if the cost for variable w_a is constant, why not use a look-up table to find the code rate with the lowest cost? The answer is, choosing the code rate with the lowest cost does not tell us the exact values of n and

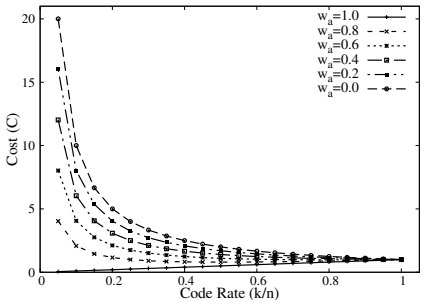


Fig. 9: Cost as a function of code rate for different w_a

w_a	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.0
Cost (C)	1/N	0.6	0.8	0.91	0.98	1.0	1.0	1.0
Code Rate	1/N	0.35	0.5	0.65	0.8	1.0	1.0	1.0

TABLE IV: Cost (C) lower bound, as a function of w_a and the corresponding code rate k/n for the lower bound

k and the particular devices. As an example, for $w_a = 0.8$, the code rate 0.5 can be achieved by 15 different combinations of (n, k). So, our algorithm not only chooses code rate with lowest cost, but also chooses devices with minimum required storage and battery remaining time.

D. R-Drive Data Retrieval

Data retrieval in *R-Drive* involves gathering all blocks of a file and reconstructing it to its original form, as illustrated in Figure 10. File retrieval is initiated by calling *get()* API function or executing *-get* command. Directory Service first communicates with EdgeKeeper and fetches the target metadata rnode, given that a rnode for the target file exists and user has permission to access the file. The *fragLocation* field in rnode contains location information of all fragments of all blocks. To reconstruct each block, File Handler must retrieve any k fragments out of n, where $k \leq n$. To retrieve any k fragments, File Handler sends fragment requests to k unique devices. Upon receiving a fragment request, a device resolves it by replying with target fragment to the requestor. When k fragment replies are received, File Handler signals Erasure Coding and Cipher components to initiate block decoding and decryption processes respectively. When all the blocks are reconstructed, the original file can also be reconstructed by merging all blocks. All fragment requests and replies are sent/received through RSock.

1) Choice of Replica Devices for Data Retrieval: To choose the replica device to request fragments from, File Handler requests a list of devices from EdgeKeeper with most remaining energy levels and sends k fragment requests to first k devices on the list. In an intermittently connected network environment, a fragment request or reply may be delayed or never be received. One way to deal with this issue is to resend the request for which no reply has been received. The question that still needs to be answered is, how long the sender should wait before initializing resend. *R-Drive* leverages the TTL in RSock to make sure that a request is resent only when the previous request failed. A fragment requestor can set a TTL within which it wants the reply to be received. If no reply is received within the set time limit, it is guaranteed that the request packet has failed and it is safe to resend the request to a different available replica device.

Algorithm 1: Choose (k, n) and n devices

Input : F, T, S_i, T_i, w_a
Output: (k,n) and n devices

$(k, n) \leftarrow (1, 1)$
$C_{min} \leftarrow 1$
for $n' \in 1...N$ **do**
  for $k' \in 1...n'$ **do**
    if *Satisfying Eq.(3.2)(3.3)* **then**
      if $C(k', n') < C_{min}$ **then**
        $(k, n) \leftarrow (k', n')$
        $C_{min} \leftarrow C(k', n')$
      if $k'/n' = k/n$ **then**
        if $A(k, n, \overline{p}) < A(k', n', \overline{p})$ **then**
          $(k, n) \leftarrow (k', n')$
$V \leftarrow$ pick up devices with $S_i > F/k$
sort V based on T_i in descending order
$V_n \leftarrow$ choose top n devices with the largest T_i
return (k, n) and V_n

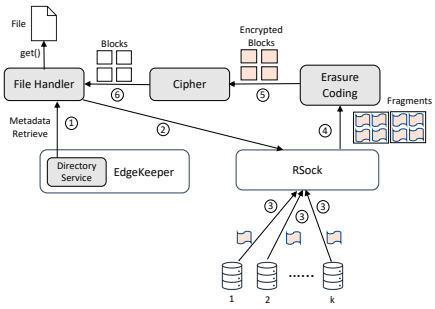


Fig. 10: *R-Drive* file retrieval steps: obtaining from the directory service the location of fragments, deciding which k fragments to retrieve and asking RSock for their delivery, applying erasure coding and re-creating the file from the decrypted blocks

E. Inter-Edge Data Exchange

This experiment description aims to illustrate inter-edge data storage. Two Anonymous edge environments MEC-1 and MEC-2 were set up indoor where both edges had separate WiFi networks and each edge had four phones connected (P1, P2, P3, P4 and P5, P6, P7, P8 respectively). At time T1, P1 stored a 100MB file in *R-Drive* system with (N, K) values of (8,4) and [P1, P2, P3, P4, P5, P6, P7, P8] as chosen nodes. Hence, P1 aimed to store 8 file fragments [f1, f2, f3, f4, f5, f6, f7, f8], each of 25MB size, to all 8 phones respectively. However, since P1 was not part of MEC-2, some fragments [f5, f6, f7, f8] waited for a link to establish between the two edges. At time T2, a new phone P9 was introduced which toggled WiFi connectivity between two edges and each time

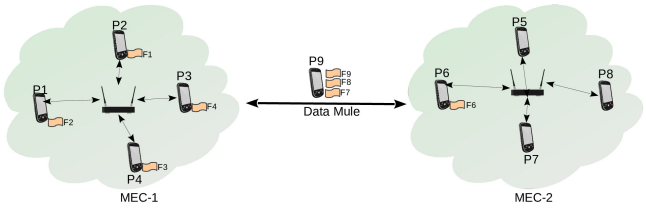


Fig. 11: Inter-edge data storage in R-Drive

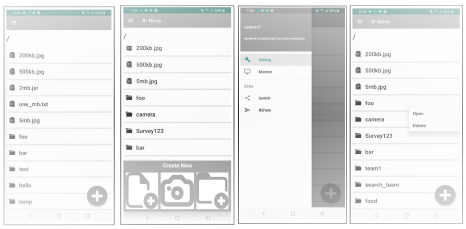


Fig. 12: *R-Drive* Android Application

stayed connected for approximately 1 minute. The purpose of P9 was to act as a *data mule* between two edges and transfer file fragments from MEC-1 to MEC-2. For first few WiFi toggles, no fragments were transferred to MEC-2. This is due to the fact that, RSock requires a minimum amount of time to learn about the networks and the connectivity pattern. Starting at third WiFi toggle, RSock could successfully transfer the fragments from MEC-1 to MEC-2 and any device in MEC-2 could retrieve the entire file. We repeated the experiment 5 times and the average fragment transfer time was approximately 6 minutes for f5, 8 minutes for f6, and 10 minutes for f7, f8. Figure 11 shows the inter-edge data storage in R-Drive.

IV. R-Drive Implementation

We implemented *R-Drive* as an app for Android mobile devices and as a daemon process for Linux desktops (HPC nodes). The implementation of *R-Drive* has approximately 10,000 lines of Java code. The Android app (shown in Figure 12) is compatible with Android version 7.1, 8.0, 10.0. The app runs as an Android background service aiming for hands-free use. *R-Drive* home page shows the current directory name and the files and folders in the current directory. Long pressing on each item will open a floating menu that allows a user to either open or delete the file/folder. On the top left, the hamburger icon opens a navigation drawer where users can find options to change application setting. The File Handler module (shown in Figure 5) exposes the *R-Drive* Java API and also handles buttons from the Android app. The module uses the *javax.crypto* package for the 256 bit AES encryption, as the Cipher. For Shamir's Secret Sharing algorithm, *R-Drive* employs *secretsharejava* [45], an open source library implementing the LaGrange Interpolating Polynomial Scheme [37]. We used BackBlaze [7] Reed-Solomon erasure coding library, an open source implementation available for both academic and commercial use.

R-Drive provides command line interface (CLI) for Linux desktop users, to perform storage operations on remote devices if the device operators allow permissions. *R-Drive* commands are interpreted by the Command Handler. Command Handler consists of a hand-written lexer and parser. Lexer takes an input command as text stream, converts into a series of tokens and parser converts the tokens into a parse-tree. The parse-tree enables Command Handler to identify the type of command. Below is the grammar *R-Drive* uses for file system commands.

```
COMMAND::= 'dfs' OPTION ARGUMENT
OPTION::= -put | -get | -mkdir | -ls | -rm
    | -setfacl | -getfacl
ARGUMENT::= PATH | PERMISSION | PATH PERMISSION
PATH::= <local_path> | <rdrive_path>
    | <local_path>  <rdrive_path>
PERMISSION::= 'OWNER' | 'WORLD' | USERS
USERS::= GUID | USERS GUID
GUID::= <40 bytes ASCII printable characters>
```

Here *local_path* means the local absolute path of a file in local file system. *rdrive_path* means either a file or a directory path in *R-Drive* file system. GUID is a unique 40 bytes long string comprising both characters and numbers.

V. R-Drive Performance Evaluation

We employed two systems for *R-Drive* benchmarking: 1) NIST Public Safety Communications Research (PSCR) deployable system, equipped with LTE (Star Solutions COMPAC-N) and Wi-Fi (Ubiquiti EdgerouterX) networks. The system has 10MHz downlink and uplink channels, the observed LTE data rates are about 95 and 20Mbps respectively. 2) Anonymous manpack system, which can be carried in a backpack of a first responder. It also consists of both LTE (BaiCells Nova 227 eNB) and Wi-Fi (Unify 802.11AC Mesh) networks, and Intel Next Unit of Computing Kit (NUC) as application server. For 20MHz downlink and uplink channels, the LTE can provide a maximum data rate of 110 and 20Mbps respectively [6]. Wi-Fi are capable of providing around 100Mbps data transfer rate. We used 15 Android devices of Essential PH-1, Samsung S8 and Sonim XP8 devices with Android versions 7.1, 8.0 and 10.0.

A. Adaptive Erasure Coding Evaluation

In this section, we provide an in-depth analysis of how w_a parameter impacts the choice (k, n) values, hence also the code-rate and F'(file size after erasure coding). We also analyze the choice of code-rate and it's impact on the cost function. We performed experiments on variable network sizes (10, 20, 30), for a file size F of 500MB, and expected file availability time T of 300 minutes. The storage S_i and expected battery remaining times T_i for nodes were generated using pseudo-random value generator with mean-variance of $(100, 20)$ and $(300, 80)$ respectively. The experiments were conducted for 30 runs before results were averaged.

1) Achieved cost for variable w_a: Table V shows the average achieved cost for variable w_a and network size. With larger network size the cost function is computed over more combinations of (n, k) values, hence the algorithm achieves cost value closer to optimal value.

w_a	Lower Bound	Achieved Cost		
		NS=30	NS=20	NS=10
1.0	0.00	0.2402	0.3613	0.66
0.9	0.6	0.6	0.6048	0.6782
0.8	0.8	0.8	0.8121	0.8360
0.7	0.9165	0.9165	0.9166	0.9183
0.6	0.9797	0.9797	0.9799	0.9807
0.5	1.0	1.0	1.0	1.0

TABLE V: Achieved cost for variable w_a and Network Size (NS)

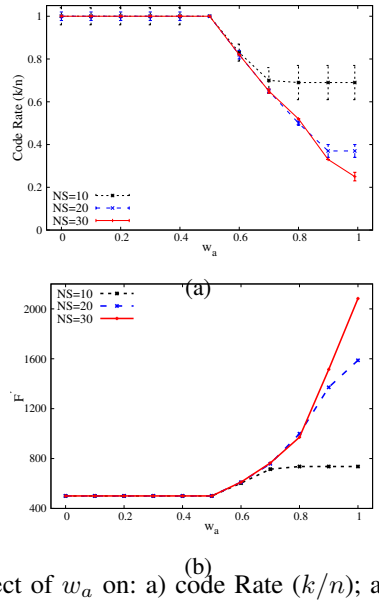


Fig. 13: Effect of w_a on: a) code Rate (k/n); and b) file size F', for network sizes, NS=10, 20 and 30

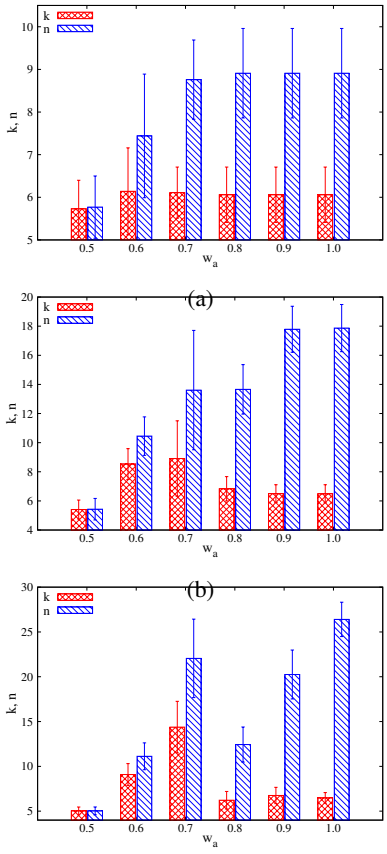


Fig. 14: Average (k, n) for different network sizes NS: a) 10; b) 20; and c) 30

2) Impact of w_a and Network Size on Code Rate and F': Figure 13a illustrates that with increased w_a, the code rate decreases. This is expected, since the algorithm takes w_a as an input for the weight of availability. With larger w_a, the algorithm chooses a larger n in an attempt to provide more data redundancy, hence the code rate decreases. Figure 13b shows the F' as a function of w_a. F' increases exponentially with higher w_a. Since chosen code rate is higher for network size 10, F' is higher compared to network size 20 and 30.

Figure 14 shows the averages of chosen n and k values for variable network size over 30 iterations. For higher w_a, the algorithm chooses larger n value to provide data redundancy. In Figure 14c, for w_a 0.8, the chosen (n, k) values are lower than the values selected for 0.7. This is because for w_a of 0.8, the optimal cost code rate is 0.5, and the algorithm produced resultant (n, k) values of (10,5), (12,6), (14,7) over 30 runs that averaged to (13.07, 6.53).

B. Directory Service Resilience

We measured resilience of *R-Drive* Directory Service based on how fast Directory Service becomes operational after failure event takes place due to several reasons such as configuration changes, device or network failures etc. EdgeKeeper reforms a new ensemble with available devices as soon as it detects that previous one is broken. The experiment was conducted at NIST testbed with Samsung S8 phones and LTE networking backbone. Each bar in Figure 15 shows average delay of reforming an edge after an event takes place. For each x-axis ticks, the equation describes the event. The term inside parentheses on the left side of the equation represents an initial condition and the remaining terms represents the changes that have been introduced. The right side of the equation represents the final stable condition. As example, **(3R)+2C-1R=3R+1C** denotes that, in an stable ensemble of 3 devices, we simultaneously added 2 new devices and took out 1 replica device, and measured how long it took to reform the ensemble with 3 devices. When an ensemble is stable, only adding new devices takes very small amount of time, as the new devices only join as clients and no reformation takes place.

C. Directory Service Latency

Figure 16a and 16b show the average metadata read and write latencies on 9 Android phones, for variable metadata sizes and number of EdgeKeeper servers, using both Wi-Fi and LTE. Figure 16a shows that for each metadata size group, as the number of EdgeKeeper servers increases more than 1, metadata retrieval latency drops. This is because more servers can perform better load balancing, resulting in overall lower retrieval latency. Variable metadata sizes have very little effect on retrieval latency. As the range of metadata size is very

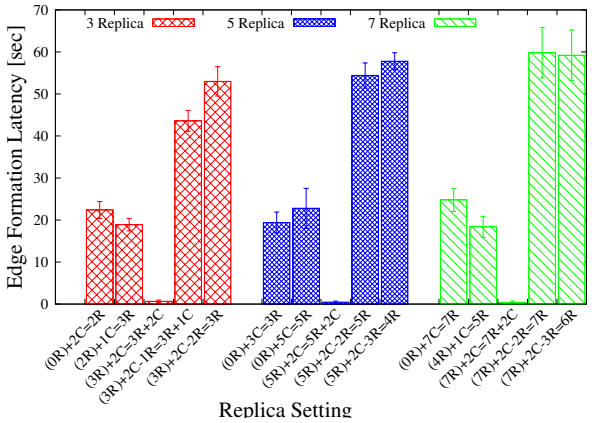


Fig. 15: Edge formation latency for variable EdgeKeeper replica settings. Here **R** and **C** denote for replica and client respectively

small, usually within 1 to 15KB, the average cost to fetch most metadata is almost the same. Figure 16b also shows that, as number of servers increases more than 1, write latency increases significantly, due to the fact that, having more than 1 server brings additional cost to check for quorum among servers before the data is committed. For both read and write, adding more servers does provide additional fault tolerance, but does not minimize latency significantly.

D. Data Throughput

Figures 17 show the average data read and write throughput for variable code rates, block sizes and link availability. The experiments were conducted with 9 Android devices with pure Wi-Fi and LTE connectivity. Each phone stored and retrieved 3GB of data simultaneously, comprising of file sizes ranging between 10 to 200MB. We controlled the link availability using another android application that can turn on/off networking based on a presetting probability. As figures suggest, read/write throughput is higher in a purely connected network compared to loosely connected network. Also, increasing block size increases throughput for both read and write due to higher block size ensures lower block count, resulting in lower number of total fragments to transfer over network. Moreover, for most block size groups, throughput slightly drops with lower code rates due to lower code rate comes with higher n and k values, resulting in more fragments to be distributed or retrieved respectively.

We compared data read/write throughput of *R-Drive* with MDFS [12], which resembles the closest with *R-Drive* in terms of design paradigm. For 2MB files and (n, k) parameters as $(7, 3)$, MDFS provided 2.3MB/sec and 2.0MB/sec of read and write throughput respectively, whereas *R-Drive* provides 11.5 and 6.5 MB/sec for read and write respectively.

E. R-Drive Overhead

1) Memory Footprint: We traced real-time memory footprint for *R-Drive* using Android Profiler [22] during file storage and retrieval process. Table VI shows the average heap object allocation and deallocation during file creation and file retrieval for variable iterations. The number of dangling objects starts to increase over time as number of file creation/retrieval increases.

Count	File Creation		File Retrieval	
	Alloc	**Dealloc**	**Alloc**	**Dealloc**
10	4,381	4,381	2,526	2,526
100	43,885	43,876	25,298	25,288
1,000	438,911	438,884	253,012	252,996

TABLE VI: Number of allocated and de-allocated objects for different numbers of file creation and retrieval

Device	Runtime	Consumed			Dist-NG
	h:min	%	mAh	Wh	Wh
[1]	3:30	12.5	377.4	1.5	3.5
[2]	3:05	11.9	323.5	1.2	3.2
[3]	3:15	12.6	381.8	1.5	3.8

TABLE VII: *R-Drive* energy consumption for different Android devices: Samsung S8 [1]; Google Pixel 2 [2]; and Essential PH1 [3]

2) Energy Consumption: Table VII shows *R-Drive* application average energy consumption for continuous workload in different Android devices. We used Battery Historian [3] to pull Android battery usage data for 100% to 0% exhaustion. We deduce that, if similar devices are used in field, first responders may need to switch the device battery after approximately 3.5 hours.

3) Processing Overhead: We measured processing time for components responsible for encryption key generation, data encryption and data erasure coding, as shown in Table VIII. We observed that, data encryption takes the majority amount of processing time.

	Shamir	AES	Reed-Solomon
Read	5%	87%	8%
Write	3%	84%	13%

TABLE VIII: Processing overhead as percentage of the total delay

4) Algorithm Execution Time: Table IX shows the average algorithm execution time in milliseconds on a Samsung S8 Android device of average over 1000 iterations.

Device	NS=30	NS=20	NS=10
Samsung S8	101.6msec	15.3msec	0.5msec

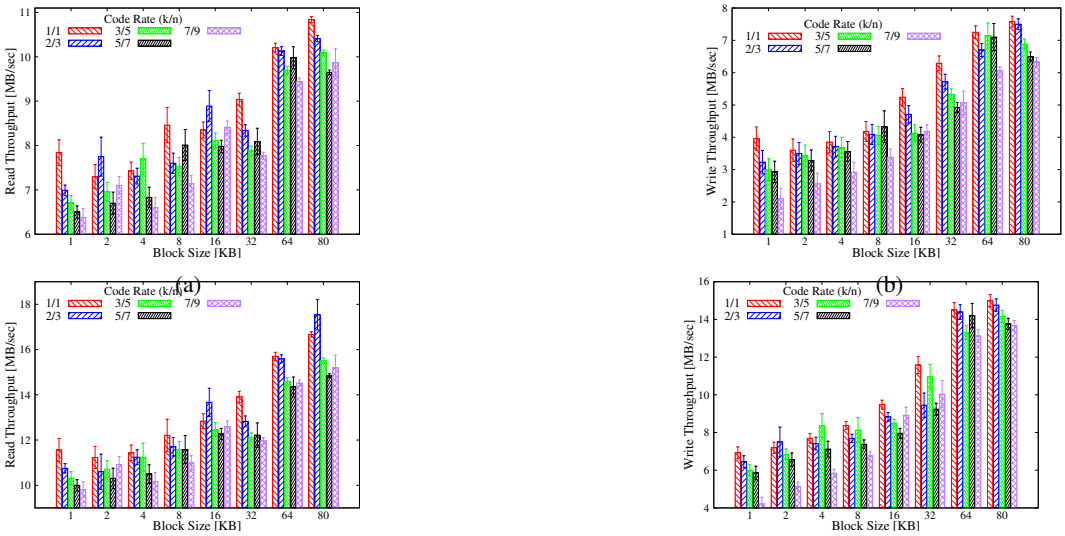
TABLE IX: Execution time of the Adaptive Coding algorithm in Samsung S8

VI. Conclusions and Future Work

We presented R-Drive, a device and network failure resilient data storage and sharing framework, with detailed explanation of how R-Drive resiliently store and share data leveraging edge devices in an environment where network connectivity

(a) (b)

Fig. 16: Metadata read (a) and write (b) latencies as a function of metadata size, for link availability 1.0



(a) (b)

(c) (d)

Fig. 17: Data read and write throughput, for 0.5 link availability (a, b) and 1.0 link availability (c, d)

constantly fluctuates. We presented and evaluated the implementation for various parameters such as device availability, network availability, block sizes, code rates etc. For future work, we want to investigate how to reduce data encryption delay using more robust library. We also plan to investigate fragments transfer from one vulnerable device to a safe one over opportunistic network before device failure.

REFERENCES

[1] AHMED, E., AND REHMANI, M. H. Mobile edge computing: opportunities, solutions, and challenges. *Future Generation Computer Systems 70* (2017).

[2] ALTAWEEL, A., AND STOLERU, R. Rsock: A resilient routing protocol for mobile fog/edge networks. https://github.com/msagor/RSock, 2021. 2021-10-25.

[3] ANDROID. Battery historian. https://developer.android.com/topic/performance/power/setup-battery-historian. 2021-10-25.

[4] APACHE. ZooKeeper programmer's guide. https://zookeeper.apache.org/doc/r3.4.6/zookeeperProgrammers.html, 2013. 2021-10-25.

[5] ARCGIS. ArcGIS Survey123. https://survey123.arcgis.com/. 2021-10-25.

[6] BAICELLS. Baicells nova 227 enb. https://www.doubleradius.com/baicells-nova-227-outdoor-tdd-enb-basestation. 2021-10-25.

[7] BEACH, B. Backblaze open sources reed-solomon erasure coding source code. https://www.backblaze.com/blog/reed-solomon/, 2015.

[8] BHUNIA, S., AND STOLERU, R. EdgeKeeper: Resilient and lightweight coordination for mobile edge computing systems. https://github.com/msagor/EdgeKeeper, 2021. 2021-10-25.

[9] BOUKERCHE, A., AND COUTINHO, R. W. Smart disaster detection and response system for smart cities. In *2018 IEEE Symposium on Computers and Communications (ISCC)* (2018), IEEE, pp. 1102–1107.

[10] CALABRESE, F., FERRARI, L., AND BLONDEL, V. D. Urban sensing using mobile phone network data: a survey of research. *ACM Computing Surveys (csur) 47*, 2 (2014), 1–20.

[11] CHAO, M., AND STOLERU, R. R-MStorm: A resilient mobile stream processing system for dynamic edge networks. In *2020 IEEE International Conference on Fog Computing (ICFC)* (2020), IEEE, pp. 64–72.

[12] CHEN, C., WON, M., STOLERU, R., AND XIE, G. G. Energy-efficient fault-tolerant data storage and processing in mobile cloud. *IEEE Trans. Cloud Computing 3*, 1 (2015), 28–41.

[13] CHENJI JAYANTH, H. *A Fog Computing Architecture for Disaster Response Networks*. PhD thesis, Texas A&M University, 2014. https://oaktrust.library.tamu.edu/handle/1969.1/152563.

[14] CIVTAK. Civtak/atak. https://www.civtak.org/. 2021-10-25.

[15] CUI, Y., SONG, J., REN, K., LI, M., LI, Z., REN, Q., AND ZHANG, Y. Software defined cooperative offloading for mobile cloudlets. *IEEE/ACM Transactions on Networking 25*, 3 (2017), 1746–1760.

[16] DROPBOX. Dropbox. https://www.dropbox.com/?landing=dbv2. 2021-10-25.

[17] DWYER, D., AND BHARGHAVAN, V. A mobility-aware file system for partially connected operation. *ACM SIGOPS Operating Systems Review 31*, 1 (1997), 24–30.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), pp. 29–43.

[19] GOOGLE. Google backup and sync. https://support.google.com/drive/answer/2374987. 2021-10-25.

[20] GOOGLE. Google drive. https://www.google.com/drive/. 2021-10-25.

[21] GOOGLE. Google files. https://files.google.com/. 2021-10-25.

[22] JORDAN, M. Android profiler. In *https://developer.android.com/studio/profile/android-profiler* (2022). Accessed Oct. 25, 2021 [Online].

[23] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 3–25.

[24] LU, Z., SUN, X., AND LA PORTA, T. Cooperative data offload in opportunistic networks: From mobile devices to infrastructure. *IEEE/ACM Transactions on Networking 25*, 6 (2017), 3382–3395.

[25] MARINELLI, E. E. Hyrax: Cloud computing on mobile devices using mapreduce. Tech. rep., Carnegie-Mellon University Pittsburgh PA School of Computer Science, 2009.

[26] MICROSOFT. Microsoft OneDrive. https://www.microsoft.com/en-us/microsoft-365/onedrive/misc-cloud-storage. 2021-10-25.

[27] NAMBURU, R. Advances in computing at the edge for military applications (conference presentation). In *Disruptive Technologies in Information Sciences IV* (2020), vol. 11419, International Society for Optics and Photonics, p. 114190A.

[28] OLANIYAN, R., FADAHUNSI, O., MAHESWARAN, M., AND ZHANI, M. F. Opportunistic edge computing: Concepts, opportunities and research challenges. *Future Generation Computer Systems 89* (2018), 633–645.

[29] OTTO, G. DHS sees wearables as the future for first responders. https://www.fedscoop.com/dhs-wearables-first-responders/, 2014. 2021-10-25.

[30] PAIKER, N. R., SHAN, J., BORCEA, C., GEHANI, N., CURTMOLA, R., AND DING, X. Design and implementation of an overlay file system for cloud-assisted mobile apps. *IEEE Transactions on Cloud Computing 8*, 1 (2017), 97–111.

[31] PAMBORIS, A., ANDREOU, P., POLYCARPOU, I., AND SAMARAS, G. Fogfs: A fog file system for hyper-responsive mobile applications. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)* (2019), IEEE, pp. 1–6.

[32] PREMSANKAR, G., DI FRANCESCO, M., AND TALEB, T. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal 5*, 2 (2018), 1275–1284.

[33] RAHMAN, A., HASSANAIN, E., AND HOSSAIN, M. S. Towards a secure mobile edge computing framework for hajj. *IEEE Access 5* (2017), 11768–11781.

[34] RAY, N. K., AND TURUK, A. K. A framework for post-disaster communication using wireless ad hoc networks. *Integration 58* (2017), 274–285.

[35] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[36] REINA, D., ASKALANI, M., TORAL, S., BARRERO, F., ASIMAKOPOULOU, E., AND BESSIS, N. A survey on multihop ad hoc networks for disaster response scenarios. *International Journal of Distributed Sensor Networks 11*, 10 (2015), 647037.

[37] SCHNEIER, B. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[38] SCOTECE, D., PAIKER, N. R., FOSCHINI, L., BELLAVISTA, P., DING, X., AND BORCEA, C. Mefs: Mobile edge file system for edge-assisted mobile apps. In *2019 IEEE 20th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)* (2019), IEEE, pp. 1–9.

[39] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[40] SHAREIT. Shareit. https://play.google.com/store/apps/anyshare. 2021-10-25.

[41] SHARMA, A., TIE, X., UPPAL, H., VENKATARAMANI, A., WESTBROOK, D., AND YADAV, A. A global name service for a highly mobile internetwork. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 247–258.

[42] SHU, Y., DONG, M., OTA, K., WU, J., AND LIAO, S. Binary reed-solomon coding based distributed storage scheme in information-centric fog networks. In *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)* (2018), IEEE, pp. 1–5.

[43] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), IEEE, pp. 1–10.

[44] SUN, X., AND ANSARI, N. EdgeIoT: Mobile edge computing for the internet of things. *IEEE Communications Magazine 54*, 12 (2016), 22–29.

[45] TIEMENS, T. secretsharejava. https://sourceforge.net/projects/secretsharejava/. 2021-10-25.

[46] XIA, M., SAXENA, M., BLAUM, M., AND PEASE, D. A. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 213–226.

[47] YANG, C., AND STOLERU, R. Hybrid routing in wireless networks with diverse connectivity. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing* (2016), pp. 71–80.

[48] ZHANG, M., BAI, Y., YUAN, S., TIAN, N., AND WANG, J. Design and implementation of file multi-cloud storage system based on android. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)* (2020), IEEE, pp. 212–215.

[49] ZHANG, Z., WANG, A., ZHENG, K., MAHESWARA, G. U., AND VINAYAKUMAR, B. Introduction to HDFS erasure coding in apache hadoop. *Cloudera Engineering Blog* (2015).

[50] ZHU, R., NIU, D., AND LI, Z. Online code rate adaptation in cloud storage systems with multiple erasure codes. Tech. rep., University of Alberta Department of Electrical and Computer Engineering, 2016.