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

M. Sagor, R. Stoleru, A. Haroon, S. Bhunia, M. Chao, A. Altaweel, M. Maurice†, R. Blalock†
Computer Science and Engineering, Texas A&M University, College Station, Texas, USA
†*National Institute of Standards and Technology (NIST)*, Boulder, Colorado, USA
{msagor, stoleru, amran.haroon, sbhunia, chaomengyuan, altaweelala1983}@tamu.edu
†{maxwell.maurice, roger.blalock}@nist.gov

***Abstract*—Mobile Edge Cloud (MEC) systems are currently being developed and gaining popularity for disaster response applications and for tactical environments as they enable intensive computation and data storage tasks at the edge. 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, an adaptive erasure-coded and encrypted resilient data storage and sharing framework for disaster response and tactical MEC applications. R-Drive adaptively chooses erasure coding parameters to ensure the highest data availability with a 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 MEC environments. We implemented R-Drive on Android and integrated it with existing MEC applications. Performance evaluation results show that R-Drive enables resilient data storage and sharing.**

I. Introduction

In Edge Clouds (EC), devices form a local cluster and use available computing and storage resources, thus allowing applications to store and process data locally [1], [2]. EC platforms that are designed for mobility (i.e., Mobile Edge Cloud, or MEC) need to handle disconnected environments where infrastructure networks such as cellular or WiFi are unavailable and cloud services are disconnected [3], [4]. MEC platforms for disaster response (e.g., wide area search and rescue, wildfire fighting) are gaining significant popularity [5] among first responders and tactical teams. In these scenarios, a first responder may be equipped, as shown in Figure 1, with mobile devices, wearable sensors, and a manpack (i.e., a backpack containing data storage, processing and communication capabilities) when performing mission-critical operations.

On-body cameras and other sensors, gesture recognition devices, as well as MEC applications on mobile devices generate large amounts of mission-critical data that needs to be stored in a resilient manner and shared seamlessly among responders [7]. Existing commercial data storage services, e.g., Dropbox [8], Google Drive [9], OneDrive [10], etc. are not designed for MEC and cannot operate in the absence of connectivity to the Internet/cloud. Although these services allow users to store and modify data offline, the data is simply stored locally making it prone to data unavailability/loss due to device failure by energy depletion or disconnection. Also, existing storage applications can only share data through the cloud via infrastructure networks. Users may employ data-sharing applications that make use of Ad-Hoc network connectivity (e.g., Bluetooth, Wi-Fi Direct), but disconnections may occur during data sharing sessions. Thus, users may be required to minimize movement and stay connected until the data-sharing session completes, which is impractical for search and rescue scenarios.

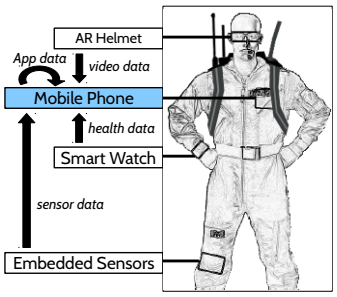


Fig. 1: Next generation first responders equipped with wearable technologies (AR helmet, on-body camera, embedded sensors) and mobile devices [6]

To address the mentioned limitations, we present R-Drive, a resilient data storage and sharing solution for MEC environments. R-Drive handles both device and network failures in MEC environments and provides secure storage of the data in a disconnected environment. The contributions of this work are as follows:

- We present the design of R-Drive, a resilient data storage and sharing system for MEC, its implementation in a real system, and its evaluation, demonstrating its suitability for MEC.
- R-Drive employs a novel Adaptive Erasure Coding scheme designed for MEC that enables resilient data storage against device failure.
- R-Drive employs a novel opportunistic networking solution for seamless data sharing while hiding network failure from client applications.
- R-Drive transparently enables existing data storage (Dropbox) and disaster response applications to share data without assistance from the cloud.

The rest of the paper is structured as follows. In Section II we review MEC architecture and systems and present state of the art solutions for resilient data storage and sharing. In

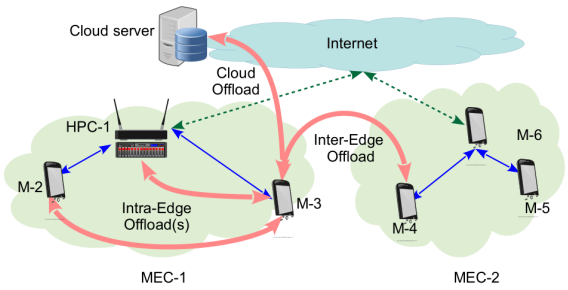


Fig. 2: MEC architecture, where mobile devices form two edge networks MEC-1 (with HPC infrastructure) or MEC-2 (ad-hoc), and share resources among themselves, or with the cloud

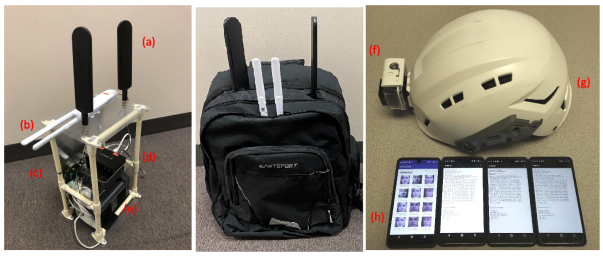


Fig. 3: DistressNet-NG hardware components: a) LTE antenna, b) Wi-Fi AP, c) LTE eNB, d) Intel NUC with LTE EPC and HPC; e) Battery; f-g) Helmet with body camera; h) Mobile devices

Section III we present the design and implementation of R-Drive as resilient and secure data storage and sharing in MEC. In Section IV we evaluate the performance of our proposed solution and conclude in Section V with ideas for future work.

II. Background and State of the Art

A. Mobile Edge Clouds for Disaster Response and Tactical Environments

In MEC, several spatially close mobile edge devices form an edge cluster under effective coordination [11]. Figure 2 depicts such a MEC architecture for disaster response or tactical environments. Within a cluster, each mobile device is a service node that can appropriately share its underutilized resources (e.g., mobile CPU/GPU, communication, memory) while providing its application services. Those devices are typically connected to an HPC node that manages communications (e.g., LTE, WiFi, WiFi-Direct), allocates IPs, provides DNS services, device naming, mapping device names to their corresponding IPs, etc. Data can be offloaded to HPC and the connected mobile devices for processing and storage. As shown in Figure 2, two MECs (where nodes "HPC-1" and "M-6" serve as central nodes for edges 1 and 2, respectively) can provide cloud-like services intra-edge as well as inter-edge.

DistressNet-NG [12], [11], [13], [14], [15] is a next-generation MEC system for disaster response, which consists (as shown in Figure 3) of a manpack equipped with wireless communication (LTE and Wi-Fi) and HPC (Intel NUC). The software architecture for DistressNet-NG is shown in Figure 4. Two important components of the DistressNet-NG architecture are EdgeKeeper and RSock (Resilient Socket). *EdgeKeeper* [11] is a distributed coordination, service discovery,

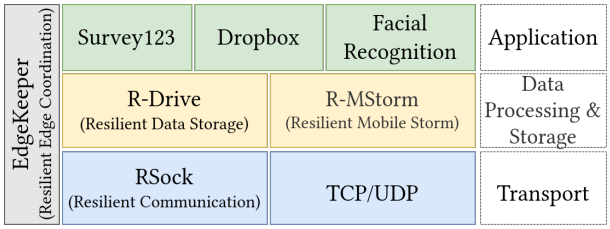


Fig. 4: DistressNet-NG software ecosystem

and meta-data storage application that runs on all devices in MEC. Based on a primary/master-replica/slave architecture, EdgeKeeper is responsible for maintaining consensus among replica storage devices. EdgeKeeper provides services such as device authentication, service discovery, edge health monitoring, network topology management, and metadata store. *RSock* [15] is a resilient transport protocol designed for sparsely connected network environments aiming to make efficient use of available network bandwidth and ensure timely data delivery. RSock performs multipath packet routing over available network interfaces such as LTE, WiFi-Direct, WiFi.

DistressNet-NG also provides R-MStorm[13], a real-time stream processing framework at the edge similar to what Apache Storm [16] provides in the cloud. Applications that were developed usign R-MStorm are Face Detection, Face Recognition, and a Voice Assistant.

B. Motivation and State of the Art

Applications in MEC platforms for disaster response generate gigabytes of mission-critical and personal data that require resilient and secure storage. Often, for further processing, critical data gets distributed among devices in the cluster that is prone to frequent disconnections and failures. This raw and processed data needs to be readily available to the cluster devices for a seamless rescue/tactical operation. Commercial storage solutions (e.g, Dropbox, Google Drive, OneDrive) store the data only on a device's local storage when the device is disconnected from the cloud, thus they are vulnerable to data unavailability due to device failure. Additionally, some applications (e.g., Survey123 shown in Figure 4 and employed by disaster responders) files are not encrypted (even in Android 12), allowing data tampering by injecting corrupt data by malicious applications.

Other solutions that target resilient data storage primarily in the cloud do not apply to MEC. OFS [17], HDFS [18] and GFS [19] are too heavy-weight either for storage overhead, memory footprint or computation overhead. MEFS [20] does not work in absence of cloud. PFS [21], FogFS [22] rely on specific mobility models that may be impractical for MEC that are disconnected from the cloud for long periods of time. Hyrax [23] ports HDFS to Android but shows poor performance for CPU bound tasks. While MDFS [24] is designed for long periods of disconnection in MEC, its implementation is based on a purely connected network, thus, not easily applicable to real world MEC.

Data sharing is equally important for MEC. Applications like Google Files [25] and SHAREit [26], let users share files

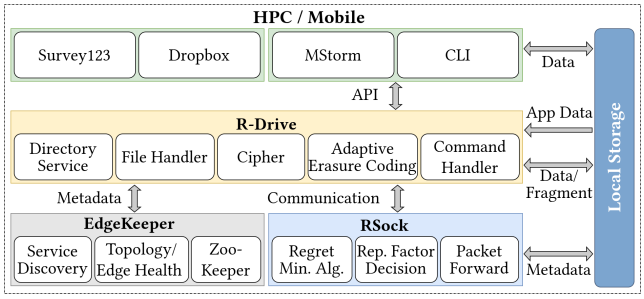


Fig. 5: R-Drive components and their integration with the DistressNet-NG software ecosystem, including EdgeKeeper, RSock and MEC applications for disaster response: MStorm [32], Survey123 [33] among others.

over Ad-Hoc networks (Wi-Fi Direct, Bluetooth, NFC). Ad-Hoc networks, however, rely on short range communication and constant connectivity, making them impractical for first response or tactical environments, characterized by highly dynamic mobility. Applications designed for MEC and deployed in disaster response scenarios need the support for opportunistic data sharing, access control, synchronization in absence of cloud connectivity, and a common namespace to manage metadata and permissions.

C. Erasure Coding for Resilient Data Storage

Reed–Solomon erasure coding [27] is a widely used coding scheme to correct burst errors associated with media failures in mass storage systems. When employing erasure coding for data storage, two parameters (n and k) need to be specified. A high n and low k increase data availability at the cost of higher storage, and vice-versa. We remark that (n, k) should be decided dynamically depending on resource availability in MEC and on user's preference for quality of service (QoS). HDFS and GFS use erasure coding for distributed storage, but the choice of parameters for erasure coding (n, k) is fixed. MDFS does not provide an online algorithm to select n and k values for variable storage availability and file sizes. Zhu et al. [28] presented an online adaptive code rate selection algorithm for cloud storage that considers real-time user demands for optimum (n, k). However, this solution assumes that all candidate storage devices have enough storage capabilities. HACFS [29] implements an extension to HDFS to adaptively choose between two (fast and compact code) coding schemes but their solution involves using fixed coding parameters for each of the coding schemes. Other researchers [30], [31] also proposed solutions for erasure coding-based data storage, yet they do not address how to choose n and k dynamically.

To the best of our knowledge, no solution exists that can provide resilient data storage and sharing in MEC for highly dynamic first responders and tactical environments, where device failures/disconnections are frequent.

III. R-Drive System Design

The R-Drive system architecture (with its five major components and their integration with the DistressNet-NG software ecosystem) is shown in Figure 5. The *Directory Service* provides a namespace for files and directories, the *File Handler* performs file and directory operations (e.g., file creation, retrieval and removal); the *Adaptive Erasure Coding* encodes and decodes data into fragments using Reed-Solomon erasure coding, the *Cipher* encrypts and decrypts data, and the *Command Handler* handles commands for basic storage operations.

TABLE I: 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. R-Drive UI and API design

Storage in R-Drive takes place via R-Drive's user interface (UI) or Java client API. The R-Drive UI allows a user to directly interact with the application. Client applications such as R-MStorm use the R-Drive API to perform data storage. For completeness, the R-Drive client API is as follows:

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

R-Drive also allows resilient data storage by monitoring files in user-selected directories on local storage, similar to Google Backup and Sync [34]. A user can select application directories which are prone to data loss due to device failure. R-Drive will periodically pull new changes and store them in R-Drive. Currently, R-Drive supports backing up application data for Survey123 [33].

In the following sections, we present in detail the design of the core components of R-Drive.

B. Directory Service and Access Control

The Directory Service handles the creation and retrieval of metadata, checking metadata permissions, and presenting a namespace to clients. Metadata in R-Drive is organized as *rnodes*, with the structure of an rnode shown in Table I. An rnode represents either a file or a directory. After creating an rnode, the Directory Service stores it in EdgeKeeper. EdgeKeeper replicates rnode to replica devices for fault tolerance, such as master failure or cluster disconnection. Consequently, if EdgeKeeper is configured with r replicas, R-Drive metadata is stored resiliently and Directory Services are provided, as long as there are $\lceil r/2 \rceil$ devices available. A directory creation takes place when a client invokes the *mkdir()* API function or when the command *-mkdir* is executed in the CLI. Directory retrieval

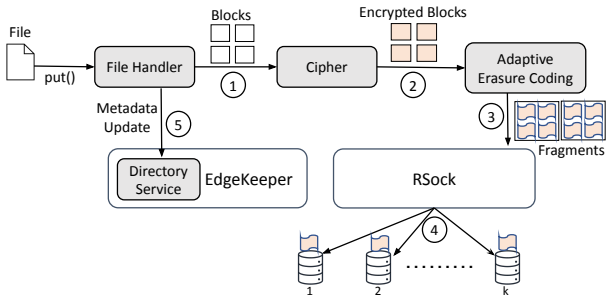
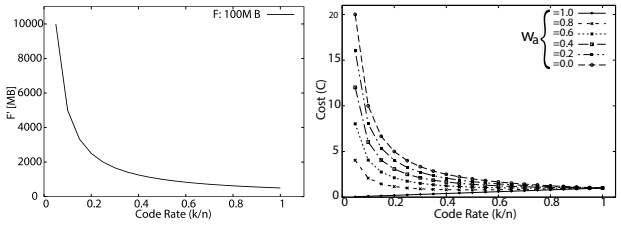


Fig. 6: R-Drive file storage steps: partitioning the file into blocks, encrypting them, applying the adaptive erasure coding and distributing the fragments to best suitable n nodes



(a) (b)

Fig. 7: a) File size F' after erasure coding (applied to a file F of size 100MB) as a function of the code rate (k/n); and b) Cost as a function of code rate for different w_a

is initiated when a client invokes the *get()* API function or when the command *-ls* is executed in the CLI.

R-Drive leverages a pluggable authentication scheme [35] for managing access control. R-Drive also implements its custom authentication as a part of the Directory Service. Permissions can also be set via the *-setfacl* and *-getfacl* commands entered through the CLI for an OWNER, WORLD, or a list of GUIDs. Permissions for an rnode pertain to itself and do not apply to children.

C. R-Drive Data Storage

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 or the *-put* command. Figure 6 shows the steps for a file creation process in R-Drive. As shown, the file is first divided into fixed sized blocks. Each block is then encrypted with a unique secret key and later converted into n fragments using erasure coding. 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. The Directory Service communicates with EdgeKeeper to create an rnode for the new file.

In the sections that follow, we present in detail the cyphering and adaptive erase coding techniques that R-Drive incorporates.

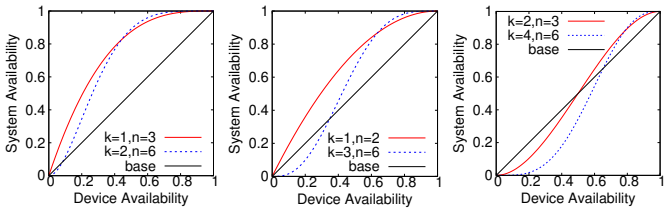
1) R-Drive Data Encryption: R-Drive uses 256 bit AES encryption using a unique secret key for file encryption. The key is further divided into B key-shards using Shamir's Secret Sharing Scheme (SSSS) [36]. SSSS is a distributed secret sharing scheme in which a secret is divided into shards in such a 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) Adaptive Erasure Coding: R-Drive uses Reed-Solomon erasure coding for data redundancy. 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 $F' = n \cdot F/k$. The 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. The choice of n and k values are directly related to data redundancy (hence availability) and storage overhead. Since devices in MEC are prone to failure the question 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 MEC system can achieve the 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 the code rate decreases, the file size after erasure coding increases, and vice-versa. But, a lower code rate usually comes with higher n and lower k values, providing added data redundancy. So, we cannot simply choose the lowest possible code rates; in that case, we will exhaust the system storage capacity very rapidly. Figure 7a shows the file size F' 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 MEC. To employ 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 code rate and a* (k, n) *value pair, which specific* n *devices should the system store the* n *file fragments to?* and 3) *How to obtain the system parameters used in answering 1) and 2)?*

Q1: What k and n values? If code rate k/n is small, there is a high probability to recover a file because only a small number of file fragments needs to be fetched (high availability). The file size after erasure coding with code rate k/n is calculated as $F' = F * n/k$, where F is 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. 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:



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

Fig. 8: Example of different (k, n) pairs determining different system availability. Each group of a, b, c contains two (k, n) pairs of same ratio. The baseline in each group represents pure local storage

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

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

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

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

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

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. Constraint (2) ensures that the storage allocation for a node does not exceed available storage for each device. Constraint (3) ensures that only devices with enough battery will be selected to sustain file lifetime T. 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, i.e., for a critical file, the system sets a large w_a so that a small k/n is chosen to improve its availability and the opposite for non-important files.

We can 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:

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

Figures 8 (a), (b), and (c) each contains two (k, n) pairs with the same ratio. As shown, when the code rate increases from $1/3$ to $1/2$ then to $2/3$, the system availability gradually decreases. Meanwhile, in each group, when the device availability is small, the (k, n) pair with a smaller n has higher availability than the other. However, as the device availability gradually increases over a threshold, the setting with a bigger n starts to achieve higher system availability than the setting with a smaller n. In R-Drive, we calculate the device availability p_i of device i as Equation 7, where T_i is the remaining time of device i.

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

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) ← (1, 1)
C_min ← 1
for n' ∈ 1...N do
    for k' ∈ 1...n' do
        if Satisfying Eq.(3.2)(3.3) then
            if C(k', n') < C_min then
                (k, n) ← (k', n')
                C_min ← C(k', n')
            if k'/n' = k/n then
                if A(k, n, p̄) < A(k', n', p̄) then
                    (k, n) ← (k', n')
V ← pick up devices with S_i > F/k
sort V based on T_i in descending order
V_n ← choose top n devices with the largest T_i
return (k, n) and V_n
```

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

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

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

Q2: Which specific n devices? After deciding (k, n), R-Drive will choose all devices with the remaining storage space larger than F/k. Next, it sorts the selected 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.

Q3: How are algorithm input parameters decided? w_a and T should be set based on two factors - how important/mission critical the file is, and how soon a user is expected to access/read the data. As an example, for mission critical data w_a can be set high, e.g., 0.8-1.0. Additionally, the user can specify an approximate T.

The complete algorithm for choosing (n, k) and the n devices, is given in Algorithm 1. When analyzing the algorithm, it is important to observe that there is a code rate for which the cost is the lowest (optimal cost), as shown in Figure 7b. The algorithm tries to achieve 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 select this (n, k), unless the devices do not meet the storage and battery remaining time requirements (as mentioned in equation 1). Table II 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 k

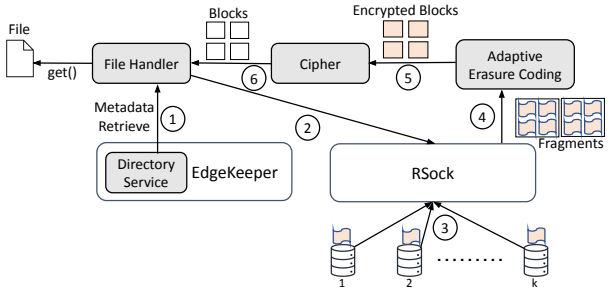


Fig. 9: R-Drive file retrieval steps: obtaining from the directory service the location of fragments, deciding which k fragments to retrieve and asking RSock for their delivery, applying erasure coding and re-creating the file from the decrypted blocks

and which devices can be used. 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 (hence n and k), 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 9. 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 an 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 for all blocks. To reconstruct each block, the File Handler must retrieve any k fragments out of n, where $k \leq n$. To retrieve any k fragments, the File Handler requests from EdgeKeeper a list of devices with their remaining energy and selects k of those with the highest remaining energy. Then, the File Handler sends fragment requests to the k devices. Upon receiving a fragment request, a device resolves it by replying with target fragment to the requestor. When k fragment replies are received, the File Handler employs Erasure Coding and Ciphering for block decoding and decryption, respectively. When all blocks are reconstructed, the original file is obtained by merging the blocks. All fragment requests and replies are sent/received through RSock.

E. R-Drive Command Handler

R-Drive provides a command line interface (CLI) for Linux desktop users, to perform storage operations on remote devices if the device operators allow it. 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
```

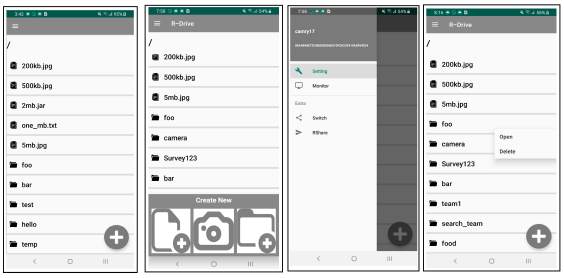


Fig. 10: R-Drive Android application, allowing navigation of the distributed file system and providing capabilities to add/remove files and directories

(a) (b)

Fig. 11: a) NIST deployable used in Gypsum, CO and b) DistressNet-NG deployable Manpack used in Disaster City, TX

```
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 the R-Drive file system. A GUID is a unique 40 bytes long string comprising both characters and numbers.

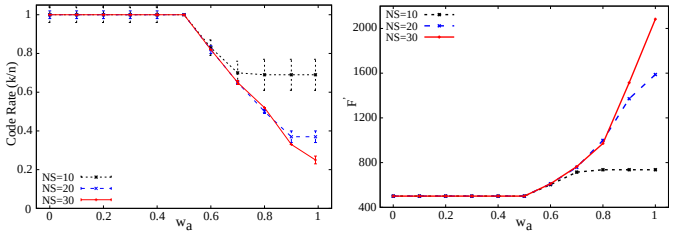
IV. R-Drive Implementation and Evaluation

We implemented R-Drive as an Android app and as a daemon process for Linux desktops. The implementation of R-Drive has approximately 10,000 lines of Java code. The app (shown in Figure 10) runs as an Android background service. The File Handler exposes the R-Drive Java API which is invoked by the user interface of the Android app. We used BackBlaze [37] Reed-Solomon erasure coding library (an open source implementation available for both academic and commercial use) and *javax.crypto* for the 256 bit AES encryption, as the Cipher.

For R-Drive performance evaluation we used both simulations of the algorithms proposed and real system evaluations. For the system evaluation we employed two rapidly deployable systems, as shown in Figure 11: a) NIST - Public Safety Communications Research (PSCR) deployable, equipped with LTE (Star Solutions COMPAC-N) and Wi-Fi (Ubiquiti EdgerouterX) router; and b) DistressNet-NG manpack consisting of LTE (BaiCells Nova 227 eNB [38]) and Wi-Fi (Unify 802.11AC Mesh) connectivity and an Intel NUC as application

TABLE III: Achieved cost for different w_a and Network Sizes (NS)

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



(a) (b)

Fig. 12: Impact of w_a on: a) code Rate (k/n); and b) file size F', for network sizes, NS=10, 20 and 30

server. We used 15 Android devices of Essential PH-1, Samsung S8 and Sonim XP8 devices with Android OS versions 7.1, 8.0 and 10.0.

In our experiments, we controlled device availability through another Android application which turns wireless connectivity on and off, based on a given probability. Experiments below were performed with link availabilities of 0.5 and 1.

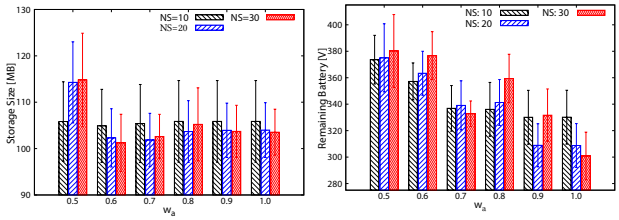
A. Adaptive Erasure Coding Evaluation

In this section, we present simulations results on how the parameter w_a impacts the (k, n) obtained by our algorithm, hence also for the code-rate and F'. We also analyze the choice of code-rate and it's impact on the cost function. We simulated network sizes of 10, 20, and 30 devices, for storing a 500MB file, with an expected file availability time T of 300 minutes. The storage S_i and expected battery remaining times T_i for nodes were generated pseudo-randomly with mean-variance of $(100, 20)$ and $(300, 80)$, respectively. The simulations were done for 30 different MEC scenarios and the results averaged.

1) Achieved cost as a function of w_a: Table III shows the average achieved cost for different w_a and network sizes. With a larger network size the cost function is computed over more (n, k) choices, hence the algorithm achieves cost closer to optimal value.

2) Impact of w_a and Network Size on Code Rate and F': Figure 12a shows that if w_a increases, the code rate decreases. This is expected, since the algorithm uses w_a as weight for Availability. With larger w_a, the algorithm chooses a larger n in an attempt to provide higher data redundancy, hence the code rate decreases. Figure 12b shows F' as a function of w_a. F' increases exponentially with higher w_a. Since the code rate is higher for a network size 10, F' is higher compared to network sizes 20 and 30.

3) Impact of w_a and Network Size on selected storage and battery remaining time: Figure 13 shows the average storage capacity and battery remaining time for the nodes selected by the algorithm. The results illustrate the fact that, on average the algorithm chooses nodes with at least minimum required storage capacity, and with higher battery remaining time. This is due to the fact that, when two different solutions have the same cost, the algorithm chooses the nodes with higher device availability, as given by Equation 7.



(a) (b)

Fig. 13: Impact on w_a on: a) storage size; and b) battery remaining time, for different network sizes NS=10, 20, 30

B. Directory Service Resilience

We investigated the resilience of the R-Drive Directory Service by measuring its startup time with different failure/-configuration modes and under different replica settings. The experiments were done using the NIST deployable system (shown in Figure 11a) and Samsung S8 phones, connected through LTE.

The experimental results are shown in Figure 14. In the figure, we use the following notation for describing MEC dynamics: **(3R)+2C-1R=3R+1C** denotes that, to a stable MEC of 3 devices (i.e., 3 replicas), we simultaneously added 2 devices (i.e., "2C," two clients) and removed 1 replica (i.e., "1R"). The resulting configuration is 3 replicas (i.e., "3R") and 1 client (i.e., "1C"). As shown, we performed experiments for three replica settings: 3, 5 and 7.

The results show that Directory Service becomes available within 20s the first time the MEC is formed. For example, in the 3 replica scenario, "(0R)+2C=2R" shows that two devices are sufficient to start providing directory services within 20s, even when 1 replica is missing. Also, we observe that adding clients to an already formed MEC that has maximum number of replicas, does not impact the Directory Service. In particular, the scenario "(3R)+2C=(3R)+2C" for the 3-replica setting shows that the Directory Service is available with minimal interruption (1-2s). Remarkably, even with a high replica setting (e.g., 7), the R-Drive Directory Service becomes available after approximately 20s.

The R-Drive Directory Service is impacted more by node failures, as shown in the figure. For a 3-replica scenario, losing 2 node replicas will result in doubling the Directory Service startup time. For example, in the "(3R)+2C-2R=3R" the MEC loses 2 replica nodes, resulting in the loss of a quorum. It takes about 55s for R-Drive Directory Service to re-configure, after 2 devices are added for a final configuration with 3 replicas (i.e., "3R"). Similar performance impact is observed for 5-replica and 7-replica MECs, where the Directory Service startup time is doubled, to 50-60s.

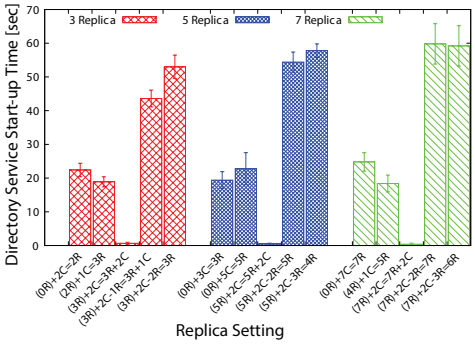
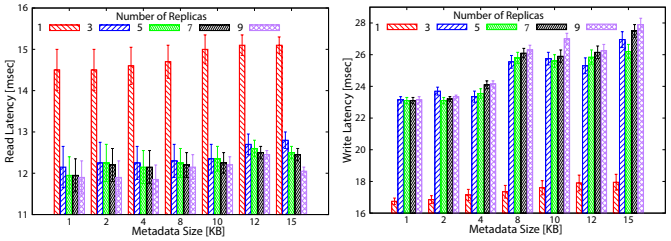


Fig. 14: Directory Service startup time for variable EdgeKeeper replica settings and different failure modes. Here **R** and **C** denote the number of replicas and clients, respectively



(a) (b)

Fig. 15: Metadata read (a) and write (b) latencies as a function of metadata size, for link availability 1.0

C. Directory Service Latency

We used the same experimental setting from the previous section to investigate R-Drive Directory Service latency for metadata read and write requests. For this set of experiments we varied the size of the metadata and the degree of resilience, i.e., number of replicas. We did not simulate node failures for these experiments. The results are shown in Figure 15. We first observe that the amount of metadata has some influence on latency, but rather minimal, for both read and write metadata operations. We also note that the number of replicas has different effects on the write operation than on the read. Increasing the number of replicas results in lower read latencies, primarily because closer (or more optimally placed) replicas can be read from. In contrast, a write operation is successful only after consensus is reached among replicas. Thus, a higher number of replicas will require more time for a successful write operation. We note, however, that the largest increase in the write latency comes from 1-replica to 3-replica. For a 1-replica scenario, the write latency is slightly larger than the read latency. This is, primarily, because the latency comes from nodes communicating with the replica node. Since writing to the local file system does not benefit from caching, the latency of a write operation is higher than for a read operation.

D. R-Drive Data Read and Write Throughput

For this set of experiments we employed the DistressNet-NG deployable system shown in Figure 11b with WiFi and LTE connectivity, and 9 Android devices. Each phone stored and retrieved 3GB of data simultaneously, comprising of file sizes ranging between 10-200MB. As mentioned before, we controlled the resiliency of R-Drive by turning wireless links on/off with a given probability. We measured the read/write throughput as a function of code rate, block size and link availability (0.5 and 1.0).

The experimental results are depicted in Figure 16, The results show, as expected, that the read/write throughput is higher in a purely connected network compared to loosely connected one. Also, increasing block size increases throughput for both read and write. This is due to lower overhead (communication and storage) of fewer blocks. Interestingly, for most block sizes, throughput slightly drops with lower code rates. We attribute this to the fact that lower code rates are associated with higher n and k, resulting in more fragments to be distributed or retrieved, respectively. We also compare the data read/write throughput of R-Drive with that of MDFS [24]. MDFS is the closest to R-Drive in terms of the design. For 2MB files and (n, k) as $(7, 3)$, MDFS provided 2.3MB/sec and 2.0MB/sec for read and write, throughput respectively, whereas R-Drive provides 11.5MB/sec and 6.5 MB/sec for read and write, respectively.

TABLE IV: Processing overhead as percentage of total delay

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

TABLE V: R-Drive energy consumption for different Android devices

Device	Runtime h:min	Consumed			Dist-NG Wh
		%	mAh	Wh	
Samsung S8	3:30	12.5	377.4	1.5	3.5
Google Pixel	3:05	11.9	323.5	1.2	3.2
Essential PH1	3:15	12.6	381.8	1.5	3.8

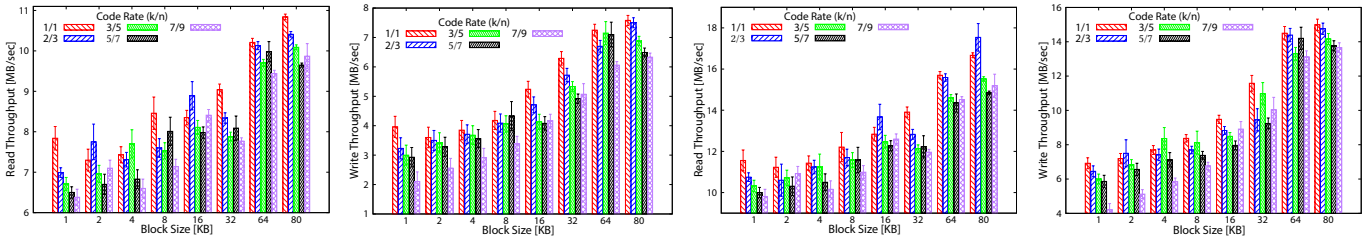
E. R-Drive Overhead

To evaluate the overhead of R-Drive, we investigated the execution time for the Adaptive Erasure Coding algorithm, the processing time for encryption and erasure coding and the energy consumption in R-Drive.

Adaptive Erasure Coding Algorithm Execution Time: We ran the adaptive coding algorithm on a Samsung S8 Android device. The average algorithm execution time (1,000 iterations) for NS of 10, 20, and 30 were 0.5 ms, 15.3 ms, and 101.6 ms, respectively. These results show that the algorithm execution has a minimal effect on R-Drive file creation operations.

Processing Latency: We measured the processing latencies of components responsible for encryption key generation, data encryption and erasure coding. The results are shown in Table IV. The results show that data encryption takes the majority of processing time.

Energy Consumption: We used the Battery Historian [39] to obtain the Android battery usage for 100% to 0% battery capacity. Table V shows R-Drive average energy consumption for a continuous workload on different Android devices. The results show that, if similar devices are used in field and used continuously for storing and retrieving data from R-Drive, a mobile device may last approximately 3.5h.



(a) (b) (c) (d)

Fig. 16: Data read and write throughput, for 0.5 link availability (a, b) and 1.0 link availability (c, d)

V. Conclusions and Future Work

In this paper we presented the design, implementation and evaluation of R-Drive, a resilient data storage and sharing solution for MEC targeting disaster response and tactical environments. R-Drive employs a novel Adaptive Erasure Coding scheme, suitable for highly dynamic environments. Experimental results show that R-Drive is resilient to node failure and its execution incurs low overhead, thus making it suitable to real-world MEC environments. R-Drive has been implemented and integrated with DistressNet-NG applications.

References

[1] E. Ahmed and M. H. Rehmani, "Mobile edge computing: opportunities, solutions, and challenges," *Future Generation Computer Systems*, vol. 70, 2017.

[2] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, 2018.

[3] R. Olaniyan, O. Fadahunsi, M. Maheswaran, and M. F. Zhani, "Opportunistic edge computing: Concepts, opportunities and research challenges," *Future Generation Computer Systems*, vol. 89, 2018.

[4] Y. Cui, J. Song, K. Ren, M. Li, Z. Li, Q. Ren, and Y. Zhang, "Software defined cooperative offloading for mobile cloudlets," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1746–1760, 2017.

[5] A. Boukerche and R. W. Coutinho, "Smart disaster detection and response system for smart cities," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1102–1107, IEEE, 2018.

[6] G. Otto, "DHS sees wearables as the future for first responders," 2014. [last accessed Aug, 2022].

[7] A. Rahman, E. Hassanain, and M. S. Hossain, "Towards a secure mobile edge computing framework for hajj," *IEEE Access*, vol. 5, 2017.

[8] Dropbox, "Dropbox." [last accessed Aug, 2022].

[9] Google. Google Drive. [last accessed Aug, 2022].

[10] Microsoft. OneDrive. [last accessed Aug, 2022].

[11] S. Bhunia, R. Stoleru, A. Haroon, M. Sagor, A. Altaweel, M. Chao, M. Maurice, and R. Blalock, "EdgeKeeper: Resilient and lightweight coordination for mobile edge clouds," in *IEEE International Conference on Mobile Ad-Hoc and Smart Systems (MASS)*, 2022.

[12] H. Chenji, W. Zhang, R. Stoleru, and C. Arnett, "DistressNet: A disaster response system providing constant availability cloud-like services," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2440–2460, 2013.

[13] M. Chao and R. Stoleru, "R-MStorm: A resilient mobile stream processing system for dynamic edge networks," in *2020 IEEE International Conference on Fog Computing (ICFC)*, pp. 64–72, 2020.

[14] A. Haroon, M. Sagor, M. Maurice, L. Jin, R. Stoleru, and R. Blalock, "On edge coordination in highly dynamic cyber-physical systems for emergency response," in *2022 Workshop on Cyber Physical Systems for Emergency Response (CPS-ER)*, pp. 7–12, 2022.

[15] A. Altaweel, C. Yang, R. Stoleru, S. Bhunia, M. Sagor, M. Maurice, and R. Blalock, "Rsock: A resilient routing protocol for mobile fog/edge networks," *Ad Hoc Networks*, vol. 134, p. 102926, 2022.

[16] Apache. Storm. [last accessed Aug, 2022].

[17] N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola, and X. Ding, "Design and implementation of an overlay file system for cloud-assisted mobile apps," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 97–111, 2017.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, 2010.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, 2003.

[20] D. Scotece, N. R. Paiker, L. Foschini, P. Bellavista, X. Ding, and C. Borcea, "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)*, IEEE, 2019.

[21] D. Dwyer and V. Bharghavan, "A mobility-aware file system for partially connected operation," *ACM SIGOPS Operating Systems Review*, 1997.

[22] A. Pamboris, P. Andreou, I. Polycarpou, and G. Samaras, "FogFS: A Fog File System For Hyper-Responsive Mobile Applications," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–6, IEEE, 2019.

[23] E. E. Marinelli, "Hyrax: Cloud computing on mobile devices using mapreduce," tech. rep., Carnegie-Mellon University Pittsburgh PA School of Computer Science, 2009.

[24] C.-A. Chen, M. Won, R. Stoleru, and G. G. Xie, "Energy-efficient fault-tolerant data storage and processing in mobile cloud," *IEEE Trans. Cloud Computing*, vol. 3, no. 1, pp. 28–41, 2015.

[25] Google. Google Files. [last accessed Aug, 2022].

[26] Smart Utils Dev Team. SHAREit. [last accessed Aug, 2022].

[27] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[28] R. Zhu, D. Niu, and Z. Li, "Online code rate adaptation in cloud storage systems with multiple erasure codes," tech. rep., University of Alberta Department of Electrical and Computer Engineering, 2016.

[29] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[30] M. Zhang, Y. Bai, S. Yuan, N. Tian, and J. Wang, "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.

[31] Y. Shu, M. Dong, K. Ota, J. Wu, and S. Liao, "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)*, pp. 1–5, IEEE, 2018.

[32] M. Chao and R. Stoleru, "R-MStorm: A resilient mobile stream processing system for dynamic edge networks," in *2020 IEEE International Conference on Fog Computing (ICFC)*, pp. 64–72, IEEE, 2020.

[33] ArcGIS. Survey123. [last accessed Aug, 2022].

[34] Google, "Google Backup and Sync." last accessed Aug, 2022].

[35] Apache, "ZooKeeper Programmer's Guide." [last accessed Aug, 2022].

[36] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[37] B. Beach, "Backblaze Reed-Solomon erasure coding source code." [last accessed Aug, 2022].

[38] Baicells, "Baicells Nova 227 eNB." [last accessed Aug, 2022].

[39] Android, "Battery Historian." [last accessed Aug, 2022].