R-DRIVE: RESILIENT DATA STORAGE AND SHARING FOR MOBILE EDGE COMPUTING SYSTEMS

A Thesis

by

MOHAMMAD RAISUL IQBAL SAGOR

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2021

Major Subject: Computer Science

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 thesis 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.

DEDICATION

For the Humanity

ACKNOWLEDGMENTS

I want to thank Professor Radu Stoleru for his constant guidance toward me (both academic and personal) throughout past years to become the person I am today. My gratitude also goes towards Professor Chia-Che Tsai and Professor I-Hong Hou for becoming my committee members and being patient with me during the processes of degree plan approval and final exam date selection.

I want to thank my parents for the constant emotional supports to achieve my goals in every major life endeavors. I am utterly grateful to the Texas A&M University system for facilitating me with a place and environment where I love to go back every day to work. Lastly, and most importantly, I want to thank my co-workers who contributed greatly towards this project- Suman Bhunia, Mengyuan Chao, Amran Haroon, Ala Altaweel; I learnt a great deal from them over the years and looking forward in the future to work side by side in collaborative works.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Radu Stoleru and Dr. Chia-Che Tsai of the Department of Computer Science and Engineering, and Professor I-Hong Hou of the Department of Electrical and Computer Engineering.

The R-Drive Directory Service implementation was performed with the help of Suman Bhunia. The Adaptive Code Rate algorithm was developed with the help of Mengyuan Chao. All other work conducted for this thesis was completed by the student independently.

Funding Sources

This research was supported by the National Institute of Science and Technology (NIST), through cooperative agreement 70NANB17H190.

NOMENCLATURE

R-Drive	Resilient Drive
MEC	Mobile Edge Computing
DTN	Delay Tolerant Networking
EC	Erasure Coding
RS	Reed-Solomon
HPC	High Performance Computing
DNS	Domain Name Service
R-MStorm	Resilient Mobile Storm
MMR	Mobile Map Reduce
RSock	Resilient Socket
GUID	Globally Unique Identifier
HRP	Hybrid Routing Protocol
TTL	Time To Live
RPC	Remote Procedure Call
GNS	Global Naming Service
CLI	Command Line Interface
SSSS	Shamir Secret Sharing Scheme
FEC	Forward Error Correction
HDFS	Hadoop Distributed File System
GFS	Google File System
MDFS	Mobile Distributed File System
CR	Code Rate

NUC	Next Unit of Computing
LTE	Long Term Evolution
TCP	Transmission Control Protocol
eNB	Evolved Node B
FC	File Creation
FR	File Retrieval

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. 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 [2, 3, 4, 5, 6]. 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 [7, 8, 9]. 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 [10, 11, 12]. 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 1.1).

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 [13, 14]. Device failure can occur due to hardware malfunction and battery depletion due to heavy use 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 [15], Google Drive [16], OneDrive [17] etc. are not designed for MEC and cannot operate in the absence of cloud. Although these services al-

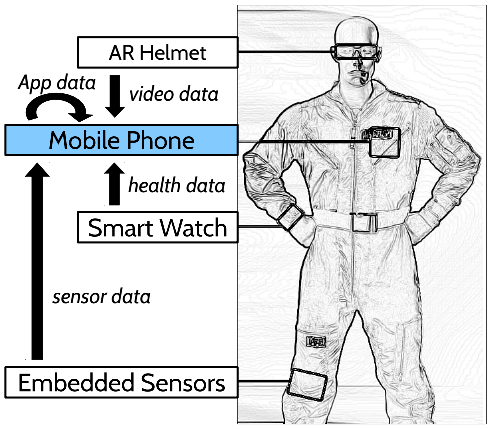


Figure 1.1: Next generation first responders equipped with wearable technologies including AR helmets, mobile phones, smart watches and embedded sensors, which generate large amounts of data that require resilient storage for processing, adapted from [1]

low 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 (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 thesis 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.

The rest of the thesis is structured as follows. In Chapter 2 we motivate the research and present state of the art solutions for the problem. In Chapter 3 we present the design and implementation of *R-Drive*. In Chapter 5 we evaluate the performance of our proposed solution and conclude in Chapter 6 with ideas for future work.

2. BACKGROUND AND STATE OF THE ART

In this chapter we present background on mobile edge computing, the necessity for data storage at the edge with the challenges faced when employing available solutions, and the requirements MEC poses to data storage and sharing.

2.1 Mobile Edge Computing for Disaster Response and Tactical Environments

Figure 2.1 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.

An example of a MEC system for disaster response is shown in Figure 2.2. The system DistressNet-NG , 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 DistressNet-NG is shown in Figure 2.3. As shown, several applications have been specifically designed for MEC. For example, R-MStorm (Resilient Mobile Storm) [18] 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 [19] 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,

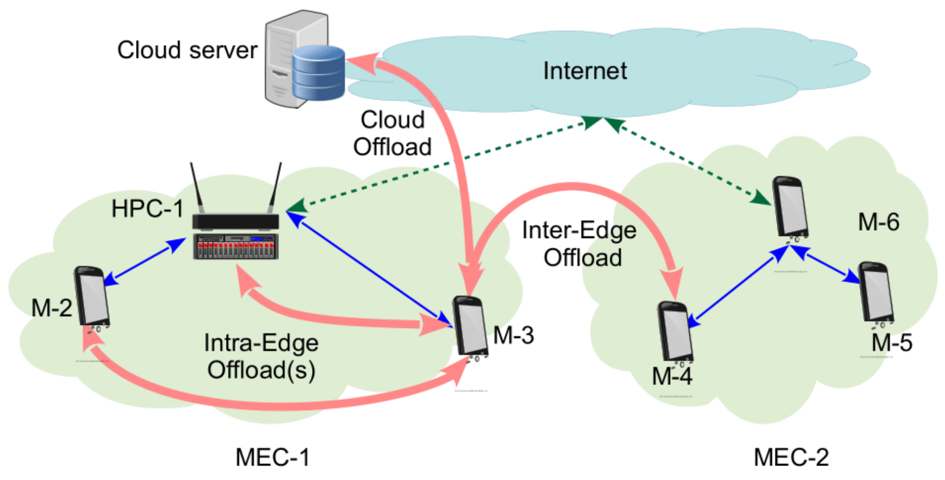


Figure 2.1: Mobile devices form two mobile edge networks (MEC-1 and MEC-2) and share resources among themselves, or with the cloud, to perform data storage/sharing and processing tasks

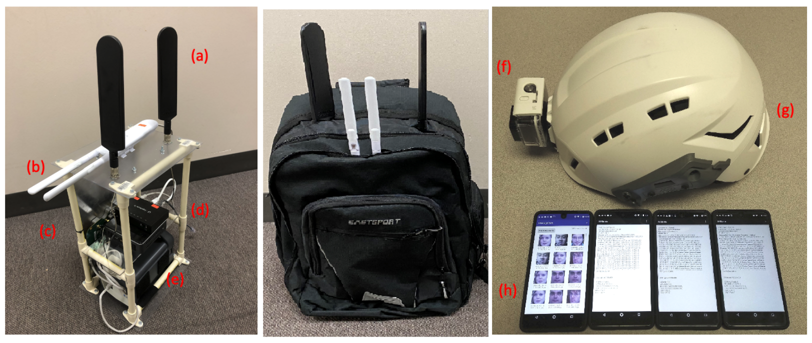


Figure 2.2: DistressNet-NG - a MEC platform for disaster response consisting of: a) LTE antenna, b) Wi-Fi AP, c) LTE eNB, d) Intel NUC that runs LTE EPC and HPC; e) Battery; f-g) Helmet with body camera; h) Android phones

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 [20] 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 2.3, two important components of the DistressNet-NG architecture are EdgeKeeper and RSock (Resilient Socket). **EdgeKeeper** [21] is a distributed coordination and

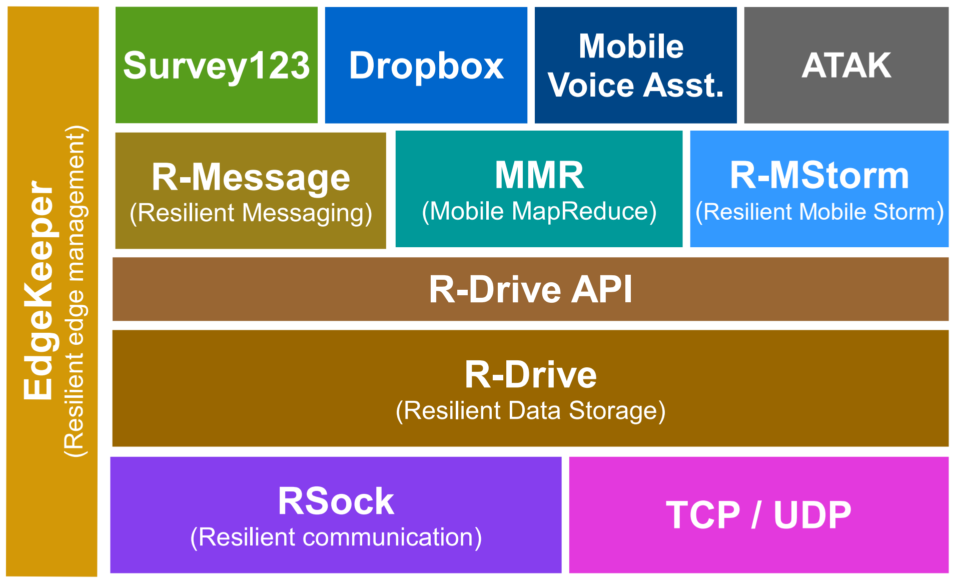


Figure 2.3: Software ecosystem for the DistressNet-NG mobile edge computing suit

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) [22] 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** [23] 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) [24], 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.

2.2 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 2.1 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 2.1: 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 [25], maintains a local cache during disconnected operations to store edited data and requires cloud connectivity to synchronize local cache with replica servers. OFS [26], carries heavy storage overhead since it keeps a copy of the same data to both local device and cloud to ensure data availability. MEFS [27] tried to retro-fit a cloud based file system to use for mobile edge, but the solution greatly relied on cloud communication. PFS [28], FogFS [29] rely in specific mobility models that makes them impractical for disconnected mobile edge. Although HDFS [30] and GFS [31] 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 [32] 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 [33] 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.

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

Table 2.2: Existing data sharing services cannot share data without cloud connectivity.

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.* Users can use file sharing applications (Google Files [34], SHAREit [35] 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 2.2 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 [36], GFS [31] 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 [33] 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 [37]. 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 storage devices have enough storage capabilities, which can be a big assumption for mobile edge computing environment where devices are heterogeneous. HACFS [38] 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 [39]. 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 [40].

R-Drive tackles above mentioned inefficiencies in existing solutions and eliminates the reliance on the cloud by bringing the cloud functions to the edge. *R-Drive* leverages opportunistic networking to perform disconnected data transfer without requiring constant connectivity to infrastructure networks. *R-Drive* applies erasure coding and encryption on files before storing in local devices so that an entire file is not exposed and only authenticated users/applications are allowed to access the files. *R-Drive*ś directory service provides a namespace to all applications/users to access and manage files. Lastly, data sharing in *R-Drive* employs opportunistic networking, hence data sharing is agnostic of infrastructure network connectivity and user mobility.

3. R-DRIVE SYSTEM

In this chapter we present the design and implementation of *R-Drive*.

3.1 R-Drive System Overview

3.1.1 System Components

Figure 3.1 shows the *R-Drive* system components and its integration with the DistressNet-NG 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 [41]. **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.

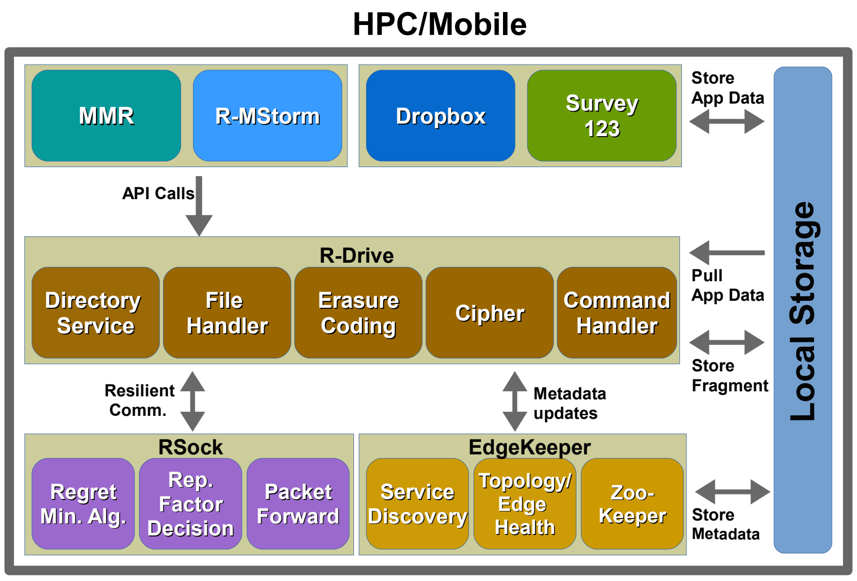


Figure 3.1: *R-Drive* components and integration with the DistressNet-NG software ecosystem

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.

3.1.2 R-Drive Authentication

R-Drive relies on EdgeKeeper for authentication and authorization. EdgeKeeper uses a Globally Unique Identifier (GUID) that is unique to each device. To obtain a GUID, a user must log into the EdgeKeeper using a X.509 certificate [42, 43]. The certificate contains user credentials such as account name, alias, certifying authority, and a private-public key pair. The certificate file is password-protected, and the user provides the password when logging in. The GUID is generated through one way hash function, thus, uniquely associated with each user. The public key associated with a GUID is stored in Global Naming Servers (GNS) and can be obtained by EdgeKeeper entities running on other devices.

3.1.3 Explicit Storage in R-Drive

Explicit 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);
```

The description and implementation of each of the API calls are presented in Sections 3.2 through 3.5.

3.1.4 Implicit Storage for R-Drive

R-Drive allows implicit data storage by monitoring files in user-selected directories on local storage, similar to Google Backup and Sync [44]. 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 and Dropbox; a user may allow *R-Drive* to access the *app data* of the above applications to periodically pull and store them in *R-Drive*.This feature is beneficial for hands-free usage by first responders who want to securely store mission critical data in *R-Drive* with minimal intervention.

3.1.5 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).

3.2 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 3.1). 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.

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. The Directory Service fetches from EdgeKeeper the target rnode corresponding to the target directory. The target directory rnode contains the list of all files and subdirectories of immediate lower level.

3.2.1 Access Control Management

R-Drive leverages ZooKeeper's access control for managing access and updating permissions for rnodes. ZooKeeper [45] 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.

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

Table 3.1: *R-Drive* rnode structure

3.3 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 3.2 shows the steps for a file creation process in *R-Drive* .

3.3.1 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) [46]. 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

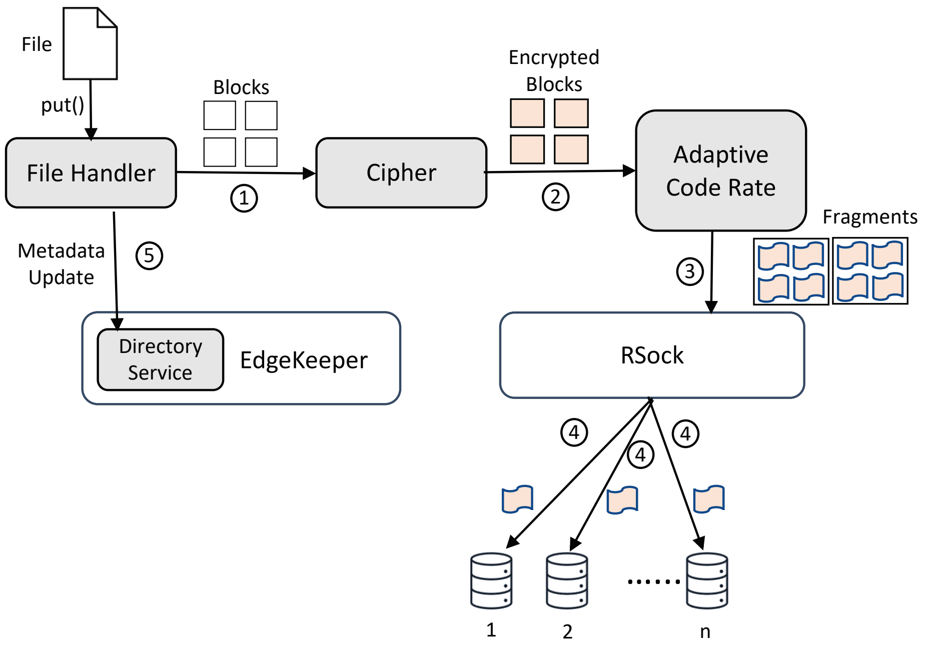


Figure 3.2: *R-Drive* file storage steps: partitioning the file into blocks, encrypting them, applying the adaptive erasure coding and distributing the fragments to best suitable n nodes

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.

3.3.2 Resilient Storage through Erasure Coding

R-Drive uses Reed-Solomon erasure coding for data redundancy [41]. 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.

3.3.3 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 3.3 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?*

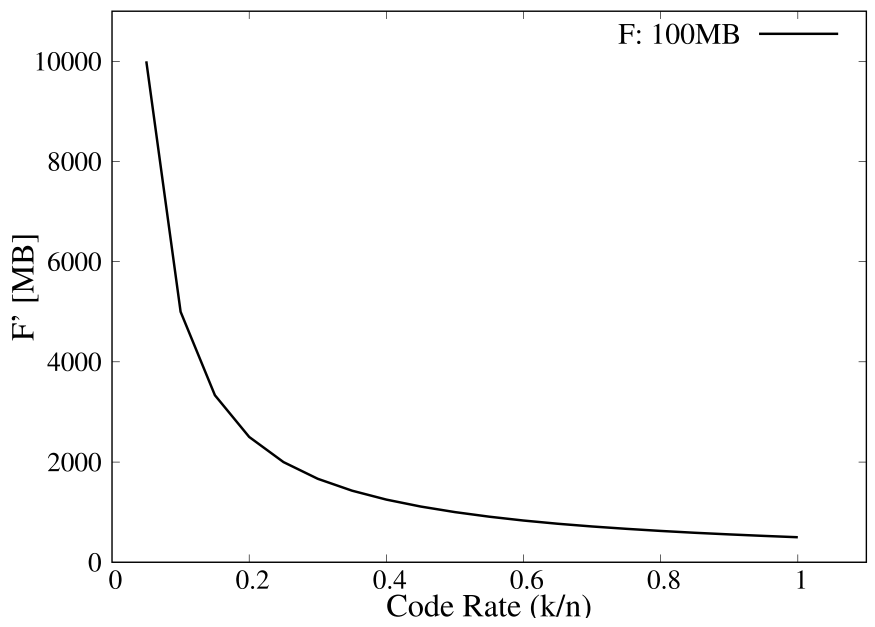


Figure 3.3: File size F' after erasure coding (applied to a file F of size 100MB) as a function of the code rate

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:

$$\begin{aligned} \underset{(k,n)}{\text{minimize}} \quad & C(k,n,w_a) = w_a * k/n + (1-w_a) * n/k && (3.1) \\ \text{subject to:} \quad & F/k \le S_n, && (3.2) \\ & T \le T_k, && (3.3) \\ & 1/N \le k \le n \le N, k, n \in Z^+ && (3.4) \\ & 0 \le w_a \le 1 && (3.5) \end{aligned}$$

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. More specific, for $w_a = 0$ (i.e., availability is not important) the objective is to reduce storage, thus $k = n$.

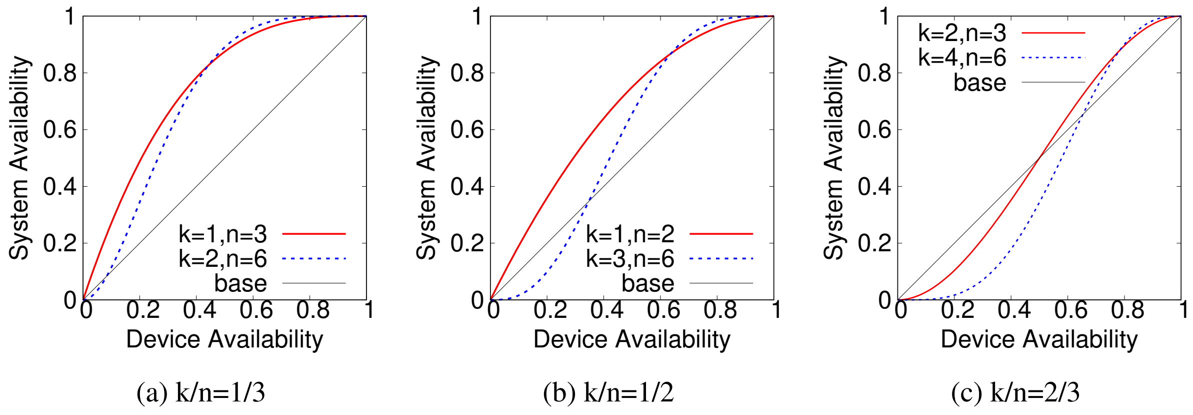


Figure 3.4: Examples showing how different (k, n) pairs determine different system availability. The example contains three small groups (a, b, c) and each group contains two (k, n) pairs of same ratio. The baseline in each group represents pure local storage

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:

$$A(k, n, p) = C_k^n p^k (1-p)^{(n-k)} + ... + C_n^n p^n \tag{3.6}$$

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 3.4, 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 (3.1). 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:

$$p_i = \begin{cases} 1, & T_i \geq T \\ T_i/T, & 0 < T_i < T \end{cases} \tag{3.7}$$

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.

3.3.4 Cost Function Lower Bound

Figure 3.5 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 3.1). Table 3.2 shows the optimal cost for variable w_a and the code rates for which the optimal cost is achieved.

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

A natural question may arise, if the cost for variable w_a is constant, why not use a look-up

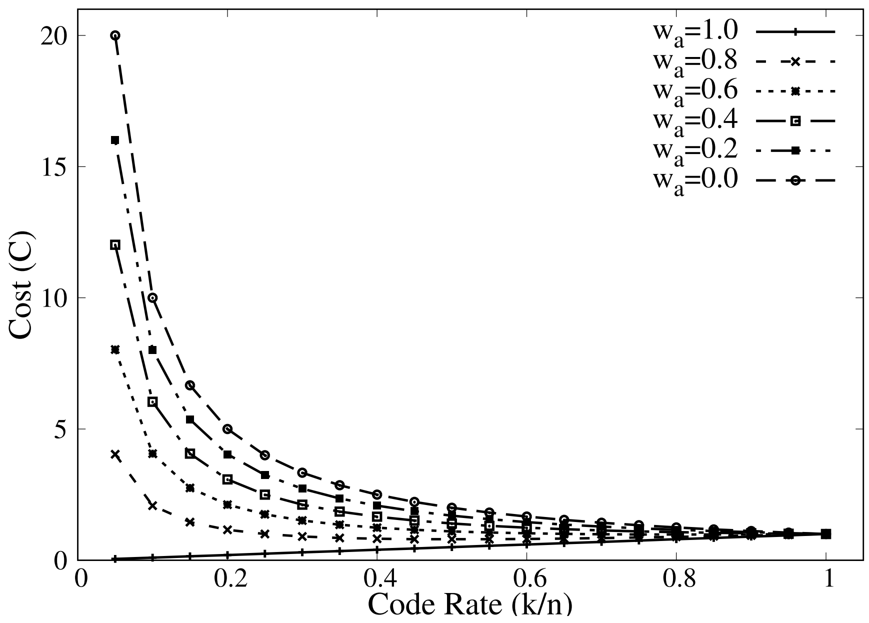


Figure 3.5: Cost as a function of code rate for different w_a

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 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.

3.4 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 3.6. 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

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

```
   Input  : $F$, $T$, $S_i$, $T_i$, $w_a$
   Output: (k,n) and n devices
 1 $(k, n) \leftarrow (1, 1)$
 2 $C_{min} \leftarrow 1$
 3 for $n' \in 1...N$ do
 4     for $k' \in 1...n'$ do
 5         if Satisfying Eq.(3.2)(3.3) then
 6             if $C(k', n') < C_{min}$ then
 7                 $(k, n) \leftarrow (k', n')$
 8                 $C_{min} \leftarrow C(k', n')$
 9             if $k'/n' = k/n$ then
10                 if $A(k, n, \overline{p}) < A(k', n', \overline{p})$ then
11                     $(k, n) \leftarrow (k', n')$
12 $V \leftarrow$ pick up devices with $S_i > F/k$
13 sort $V$ based on $T_i$ in descending order
14 $V_n \leftarrow$ choose top $n$ devices with the largest $T_i$
15 return $(k, n)$ and $V_n$
```

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.

3.4.1 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

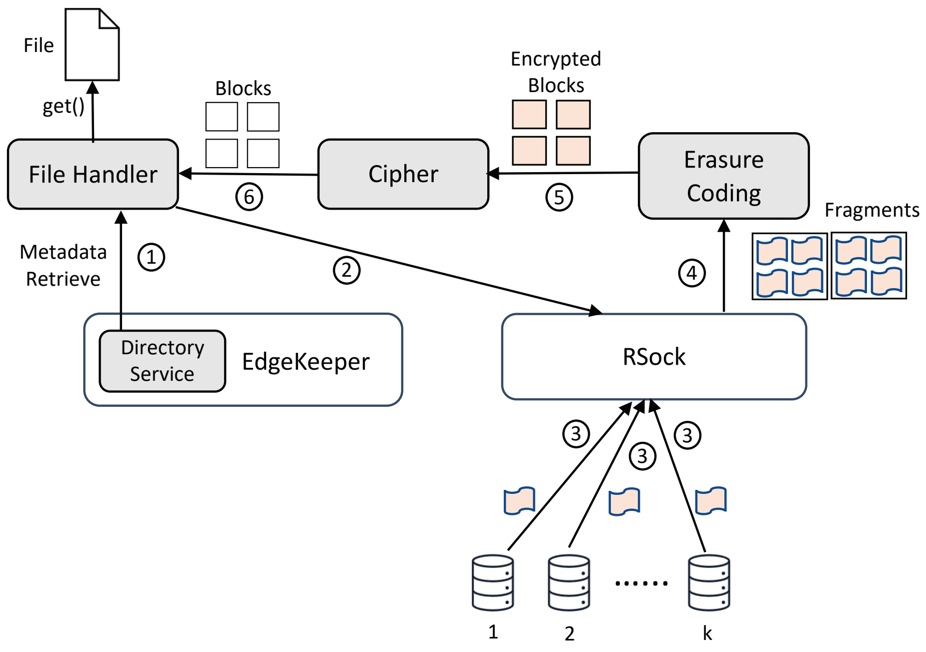


Figure 3.6: *R-Drive* file retrieval steps: obtaining from the directory service the location of fragments, deciding which k fragments to retrieve and asking RSock for their delivery, applying erasure coding and re-creating the file from the decrypted blocks

available replica device.

3.5 R-Drive Data Deletion

File and directory deletions are executed using the *rm()* API call or the *-rm* command. The Directory Service informs EdgeKeeper to delete the target rnode from ZooKeeper, and returns a copy of the rnode to the requester. For deleting a file, the File Handler learns the GUIDs for of all fragment (the *fragLocation* field) and sends fragment deletion requests to them, using RSock.

3.6 Inter-Edge Data Exchange

R-Drive supports inter-edge directory and file exchanges for enabling collaboration among teams during search and rescue operations. As shows in figure 2.1, when two edges MEC-1 and MEC-2 come within each other's network range, or when they discover each other over internet, their EdgeKeeper masters can exchange their *R-Drive* directory information. As example, a user in MEC-1 can browse through the directories and contents of MEC-2 with 'WORLD' permission tag. Any file retrieval request from MEC-1 is first submitted to EdgeKeeper master. EdgeKeeper master of MEC-1 will send a request to EdgeKeeper master of MEC-2 for the file fragments. Upon verifying the permission, EdgeKeeper master of MEC-2 will trigger fragment forwarding request to all fragment replica devices of MEC-2. Fragment replica devices of MEC-2 will directly send the fragments to the file requestor at MEC-1 through RSock.

3.7 R-Drive Consistency Model

R-Drive provides eventual and sequential consistency for file data and metadata, respectively. As discussed earlier, *R-Drive* leverages RSock to transfer file fragments in a disconnected environment. Although RSock attempts to deliver a file within TTL, the deliveries are not guaranteed to be instantaneous, due to network delays or disconnections. Thus, at some point in time, some replica devices may be under-replicated, since some target replica devices may not have received the file fragments yet. Thus, for file fragments distribution, *R-Drive* provides an eventual consistency model, i.e., given enough time, all replicas will eventually receive their corresponding fragment data. As discussed earlier, *R-Drive* metadata is stored in EdgeKeeper. EdgeKeeper provides a sequential consistency model for metadata storage; all updates from clients are applied in the order they are received. Consequently, the *R-Drive* metadata updates are also sequentially consistent.

3.8 R-Drive Sample Execution

This experiment description aims to illustrate the case when *R-Drive* provides eventual consistency with a demonstration of inter-edge data storage. Two DistressNet-NG 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 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.

4. R-DRIVE IMPLEMENTATION

We implemented *R-Drive* as an app for Android mobile devices and as a daemon process for Linux desktops (HPC nodes). Both the Android app and the Linux service share the same code base, except the fact that on Android we had to make *R-Drive* as an Android activity that runs in the background. The *R-Drive* Linux service allows control over a Command Line Interface (CLI). The Android app and Linux service are further described below.

4.1 Android App

The Android app (shown in Figure 4.1) is compatible with Android version 7.1, 8.0, 10.0. The app runs as an Android background service aiming for hands-free use; users such as first responders and military personnel can minimize the application and *R-Drive* services can still remain operational. Users have the option to set event notifications so that they can be notified for various task completions.

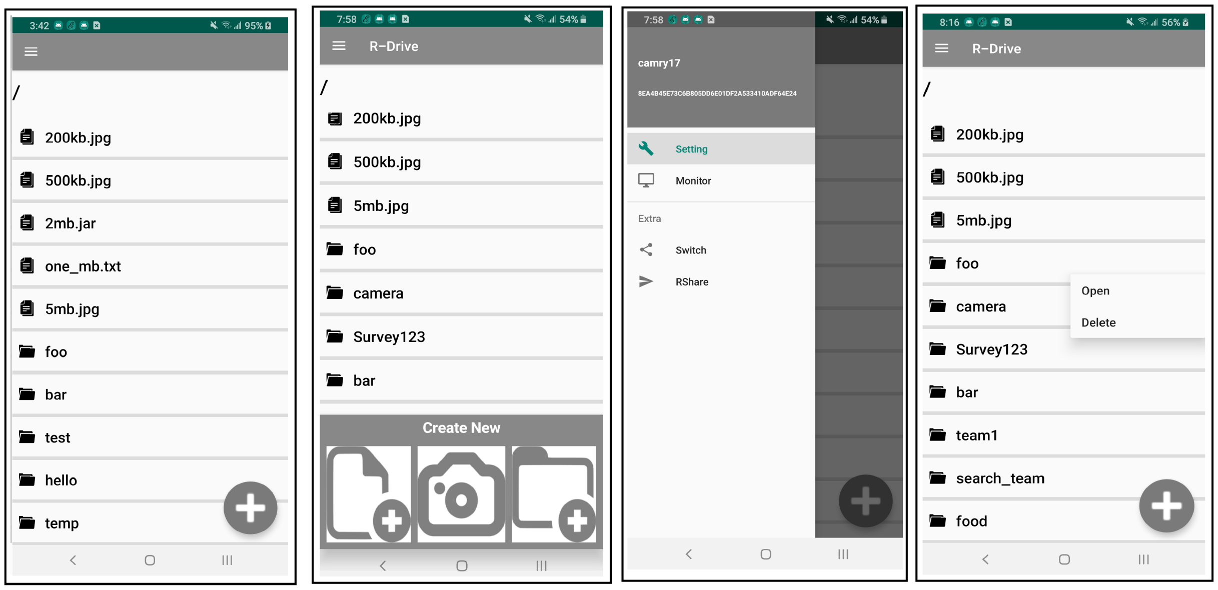


Figure 4.1: *R-Drive* Android Application

The implementation of *R-Drive* has approximately 10,000 lines of Java code. *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. A user can press the *add* floating button in home page to either select a file from the local storage or take an image using camera and store in *R-Drive*. On the top left, the hamburger icon opens a navigation drawer where users can find options to change application setting, monitor device local directory, browse neighboring edge directory and to open RShare messaging app.

4.2 Linux Service and CLI

R-Drive provides command line interface (CLI) for Linux desktop users, to perform storage operations on remote devices if the device operators allow permissions. All commands are sent from the CLI to mobile devices via RSock. *R-Drive* commands are interpreted by the Command Handler. *R-Drive* commands are similar to Hadoop hdfs commands such as *-put, -get, -mkdir, -ls, -rm* [47]. 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.

4.3 R-Drive Components Implementation

In this section we present how the core components of *R-Drive* as shown in Figure 3.1 are implemented.

R-Drive interacts with RSock and EdgeKeeper through Java client packages abstracting the APIs for the two services. Internally, the Java client packages employ TCP sockets.

The File Handler module (shown in Figure 3.1) 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* [48], an open source library implementing the LaGrange Interpolating Polynomial Scheme [49]. The File Handler module also employs a Reed-Solomon erasure coding library, BackBlaze [50], an open source implementation available for both academic and commercial use.

For executing the Adaptive Code Rate algorithm, the File Handler obtains from EdgeKeeper edge topology information: which nodes are available, their available storage (S_k in Equation 3.1) and their available battery levels (T_k in Equation 3.1). Based on the the topology information, the File Handler computes the individual availability scores, the optimal code rate n/k and most suitable n devices, as presented in Section 3.

Once the decision on which n nodes will store the fragments, EdgeKeeper is updated with file meta-data information and the fragments are distributed using RSock to the corresponding nodes.

5. 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 can be powered by a portable generator and can be rapidly deployed to a disaster zone on a pickup truck. For 10MHz downlink and uplink channels, the observed LTE data rates are about 95 and 20Mbps respectively. 2) DistressNet-NG 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 [51]. All the components are directly powered by inboard batteries inside the manpack. For both systems, the Wi-Fi are capable of providing around 100Mbps data transfer rate. We used a total of 15 Essential PH-1, Samsung S8 and Sonim XP8 devices with Android versions 7.1, 8.0 and 10.0.

5.1 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.

5.1.1 Achieved cost for variable w_a

Table 5.1 shows the average achieved cost for variable w_a and network size. For almost all w_a, the average achieved cost leans more towards the optimal cost with larger network size. This is due to the fact that, 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.

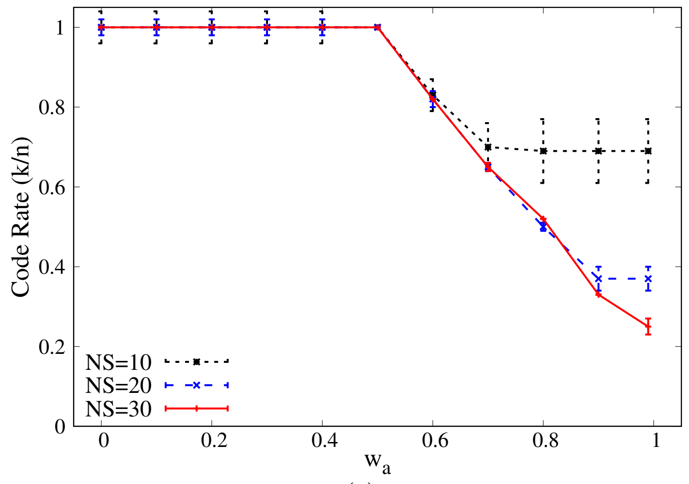
w_a	Lower	Achieved Cost		
	Bound	NS=30	NS=20	NS=10
1.0	0.00	0.2402	0.3613	0.66
0.9	0.6	0.6	0.6048	0.6782
0.8	0.8	0.8	0.8121	0.8360
0.7	0.9165	0.9165	0.9166	0.9183
0.6	0.9797	0.9797	0.9799	0.9807
0.5	1.0	1.0	1.0	1.0

Table 5.1: Achieved cost for variable w_a and Network Size (NS)

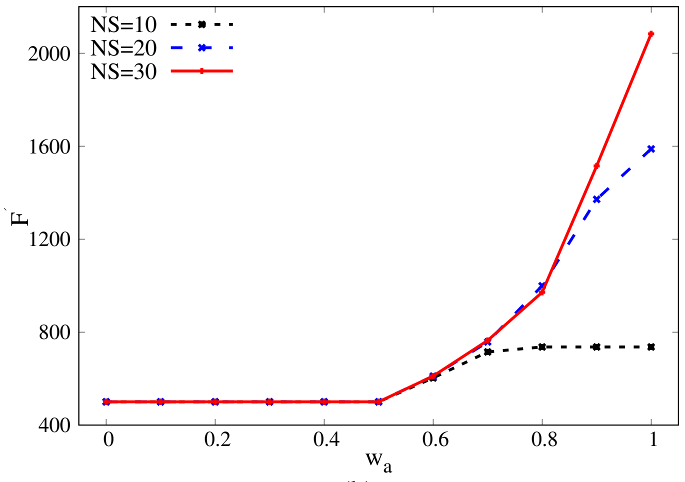
5.1.2 Impact of w_a and Network Size on Code Rate and F'

Figure 5.1a 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. If w_a is higher, the algorithm chooses a larger n in an attempt to provide more data redundancy, hence the code rate decreases. For network size 10, the chosen code rate is much higher compared to network size of 20 and 30. This is due to the fact that, for smaller networks size, several runs could not produce a solution due to not having nodes with enough storage and/or remaining battery time. Figure 5.1b shows the F' as a function of w_a. F' increases exponentially with higher w_a. Again, since chosen code rate is higher for network size 10, F' is higher compared to network size 20 and 30.

Figure 5.2 shows the averages of chosen n and k values for variable network size over 30 iterations. As discussed earlier, for higher w_a, the algorithm chooses larger n value to provide data redundancy. The cost function aims to reach towards the minimum cost, regardless of the choice of (n, k) values. In Figure 5.2c, 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).



(a)

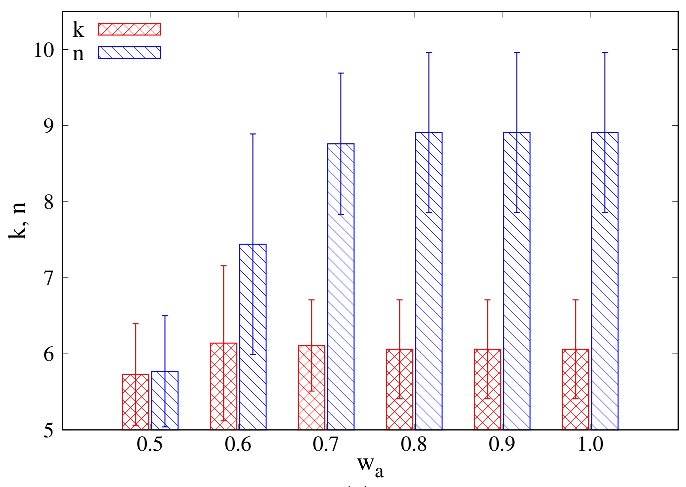


(b)

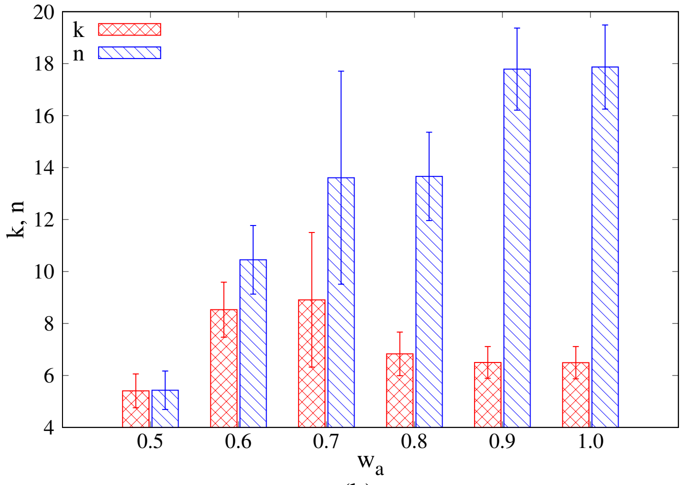
Figure 5.1: Effect of w_a on: a) code Rate (k/n); and b) file size F', for different network sizes, NS=10, 20 and 30

5.1.3 Impact of w_a and Network Size on selected storage and battery remaining time

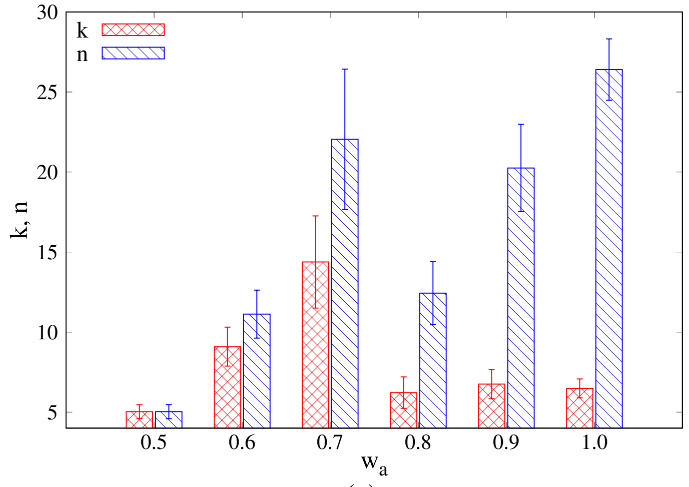
Figure 5.3 shows the average storage capacity and battery remaining time of the selected nodes. Figures 5.3a and 5.3b illustrate the fact that, on average the algorithm chooses nodes with at least minimum required storage capacity, but higher battery remaining time. This is due to the fact that, on the occasion of two different solutions having same cost, the algorithm has provision to choose the nodes with higher device availability, as mentioned in equation 3.7.



(a)

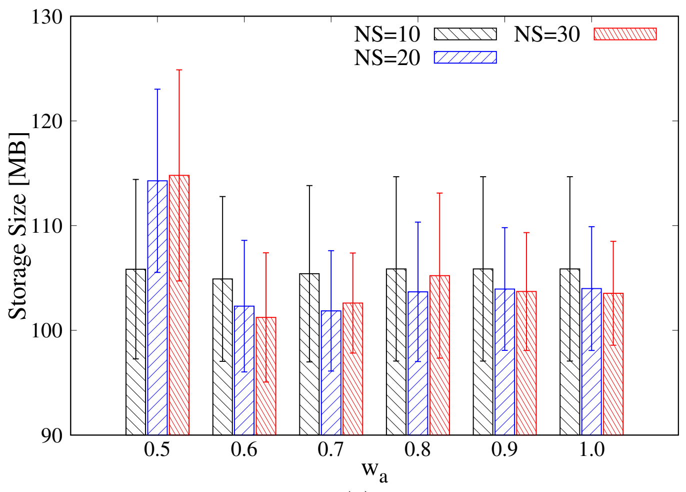


(b)

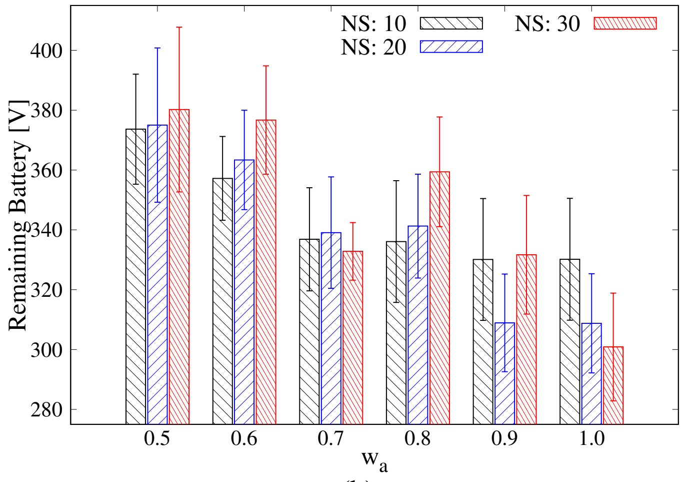


(c)

Figure 5.2: Average (k, n) for different network sizes NS: a) 10; b) 20; and c) 30



(a)



(b)

Figure 5.3: Impact on w_a on: a) storage size; and b) battery remaining time, for different network sizes NS=10, 20, 30

5.2 Directory Service Resilience

We measured resilience of *R-Drive* Directory Service based on how fast Directory Service becomes operational after failure event takes place. Directory Service becomes inoperational when EdgeKeeper ensemble is broken. Ensemble can break due to several reasons such as configuration changes, device or network failures etc. EdgeKeeper initiates to reform a new ensemble with available devices as soon as it detects that previous one is broken. We performed experiments with EdgeKeeper replica configurations of 3, 5, and 7 to measure average latency of edge reformation delay by introducing changes in a stable ensemble. The experiment was conducted at NIST testbed with Samsung S8 phones and LTE networking backbone. Each bar in Figure 5.4 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. Note that, regardless of some replica device leaving and new devices joining as replicas, *R-Drive* metadata are replicated into new replica devices to provide maximum fault tolerance.

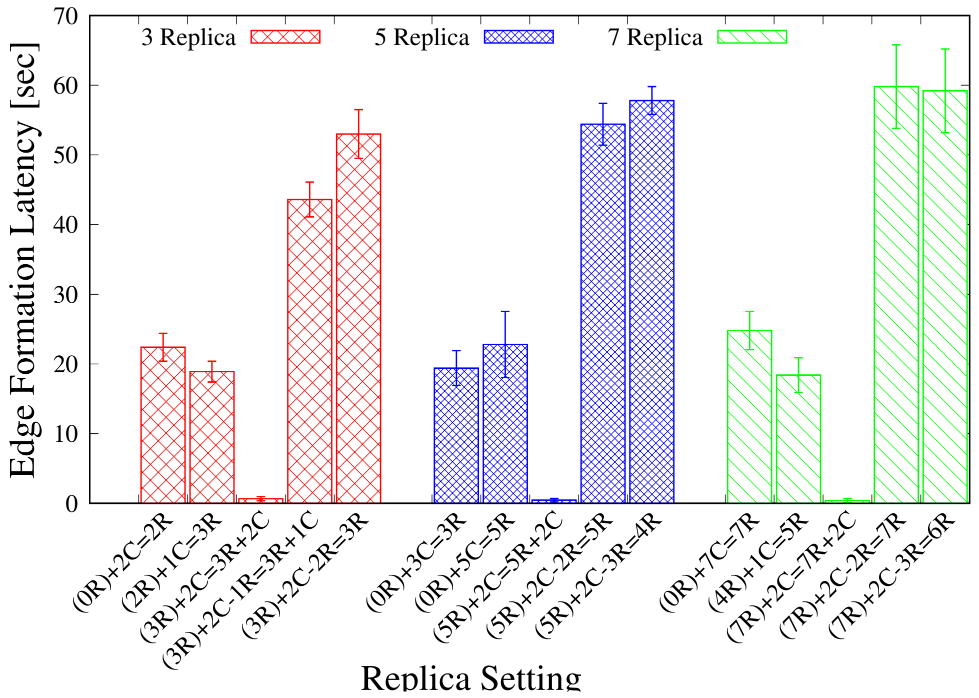


Figure 5.4: Edge formation latency for variable EdgeKeeper replica settings. Here **R** and **C** denote for replica and client respectively

5.3 Data Resilience

We evaluate data resilience of *R-Drive* system as file retrieval success rate as a function of device availability. We controlled device availability by assigning a probability value for each device. Devices mimicked unavailability simply by not replying to the fragment requests from other devices. As example, if a device availability value is set to 0.5, it randomly replied to only 50% of all fragment requests it received. The experiment was conducted in a purely connected Wi-Fi network to ensure that network connectivity had no effect on the experiment, and only device availability had effect on the experiment. We first stored a total of 10 files, each of size 10MB, at 10 different devices [P1, P2, P3, P4, P5, P6, P7, P8, P9, P10] with (n, k) values of (10, 5). Then using a different phone p11, we tried to retrieve the files. As shows in Figure 5.5, the retrieval success rate gradually increases along with device availability. Especially, when device availability value exceeds the code rate (in this case 0.5), the file retrieval success rate increases rapidly, which can be observed for availability value 0.8. Note that, the file retrieval success rate is 100% for device availability 1.0 in a purely connected network.

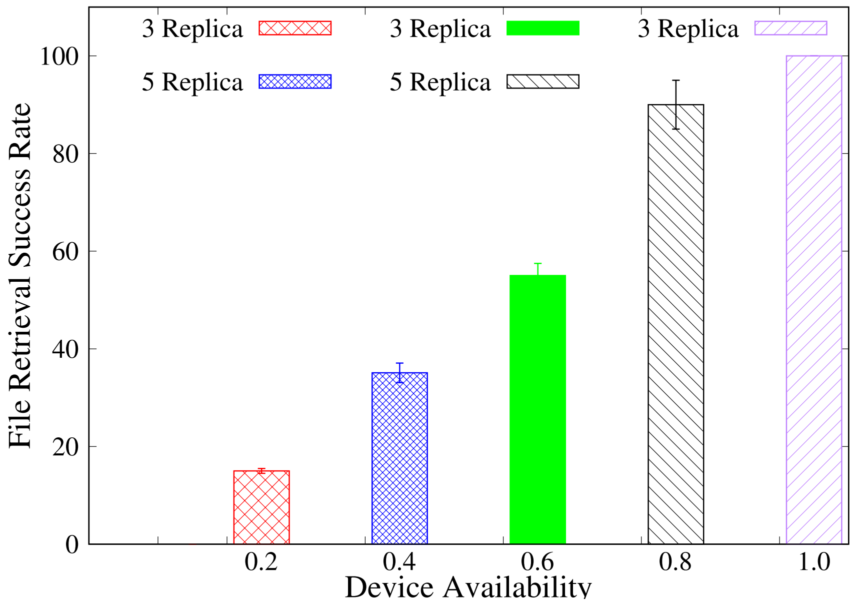
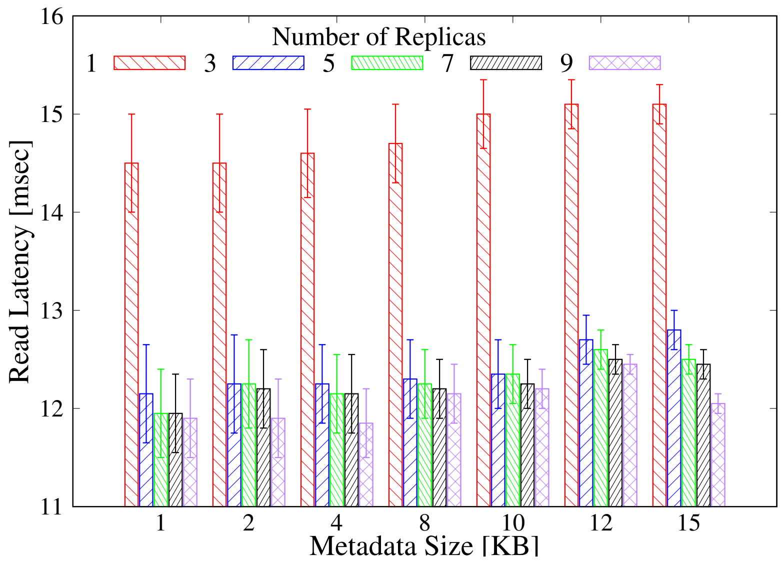
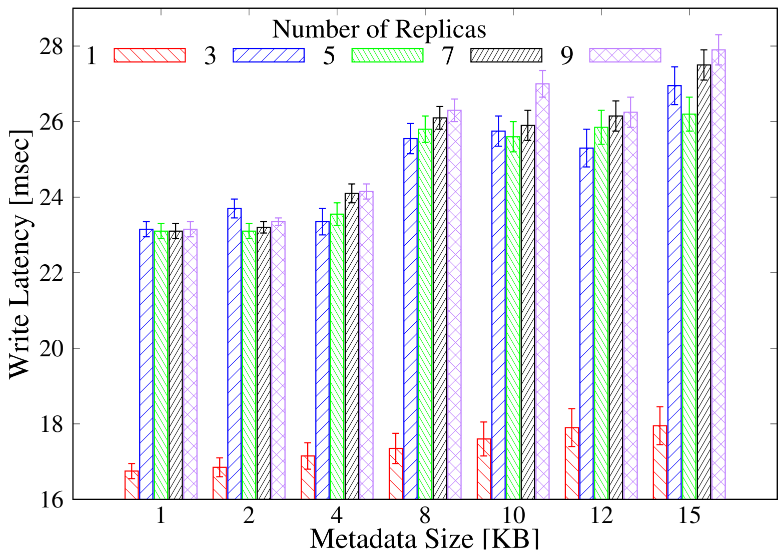


Figure 5.5: *R-Drive* file retrieval success rate as a function of device availability



(a)



(b)

Figure 5.6: Metadata read (a) and write (b) latencies as a function of metadata size, for link availability 1.0

5.4 Directory Service Latency

As mentioned earlier, EdgeKeeper stores *R-Drive* metadata in ZooKeeper for resilient storage. Hence, for different replica configurations and metadata sizes, metadata read/write performances can vary significantly. Figure 5.6a and 5.6b show the average metadata read and write latencies, respectively for variable metadata sizes and number of EdgeKeeper servers. Each result represents the average latency of 1,000 read or write operations using maximum of 9 Android devices in a purely connected network including both Wi-Fi and LTE. All 9 devices, regardless of acting as a server, performed read or write operations simultaneously. Figure 5.6a shows that for each metadata size group, as the number of EdgeKeeper servers increases more than 1, metadata retrieval latency drops. This is due to the fact that, 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 small, usually within 1 to 15KB, the average cost to fetch most metadata is almost the same. Figure 5.6b also shows that, as number of servers increases more than 1, write latency increases significantly. This is 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.

5.5 Data Throughput

Figures 5.7 and 5.8 show the average data read and write throughput for variable code rates, block sizes and link availability. The experiments were conducted with DistressNet-NG testbed, with a maximum of 9 Android devices with Wi-Fi and LTE connectivity. Each phone stored and retrieved 3GB of data simultaneously, comprising of file sizes ranging between 10 to 200MB. Write time was measured as the time it took for processing all fragments and distributing them to destination devices. Read time was measured as the time it took for all fragments to be retrieved and constructed as original file. We calculated throughput by dividing the data size with the time it took for distribution or retrieval. We controlled the link availability using another android application that can turn on/off networking based on a presetting probability. The experiment was conducted in a purely connected network (link availability 1.0), as well as loosely connected network (link availability 0.5). 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. This is due to the fact that, higher block size ensures lower block count, resulting in lower number of total fragments that requires distribution or retrieval over the network. Moreover, for most block size groups, throughput slightly drops with lower code rates. This is due to the fact that, lower code rate comes with higher n and k values, resulting in more fragments to be distributed or retrieved respectively.

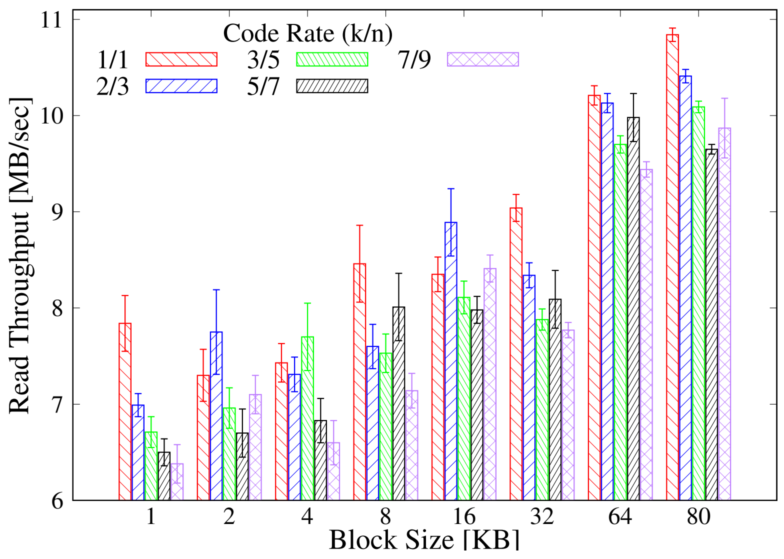
We compared data read/write throughput of *R-Drive* with MDFS [33], which resembles the closest with *R-Drive* in terms of design paradigm. For 2MB files and (n, k) parameters as $(7, 3)$, MDFS provided 2.3MB/sec and 2.0MB/sec of read and write throughput respectively, whereas *R-Drive* provides 11.5 and 6.5 MB/sec for read and write respectively. We dug into MDFS implementation and found major design and implementation flaws that are primarily responsible for such low performance. MDFS employs a topology discovery mechanism which is triggered before every file creation and retrieval operations. Any data communication takes place via individual TCP session that requires time to initiate.

5.6 Data Sharing with Opportunistic Network

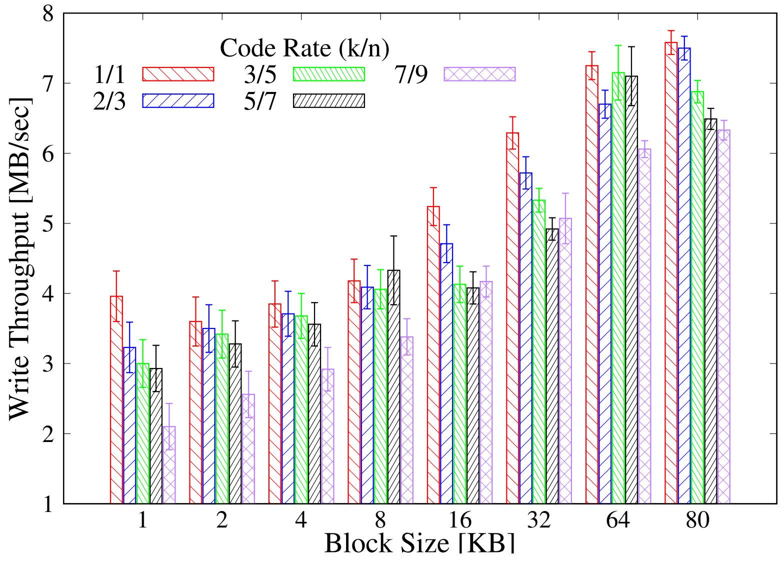
5.6.1 Data Sharing with Opportunistic Network

Figure 5.9 shows the average data sharing delay over variable link availability over Wi-Fi.

We set up a topology of 10 devices connected to the same Wi-Fi network. Each device shared a file of 10MB to every other phones. Hence each device stored a total of 90MB of data in *R-Drive* , and all devices stored a total of 900MB data in *R-Drive* . We controlled the Wi-Fi link availability as (0.2, 0.4, 0.6, 0.8, 1.0) of devices using a synthetic application that periodically turned Wi-Fi links on/off based on previously set probabilities. As example, if the link failure probability is 0.8, the device will be randomly connected to Wi-Fi for 20% of the experiment



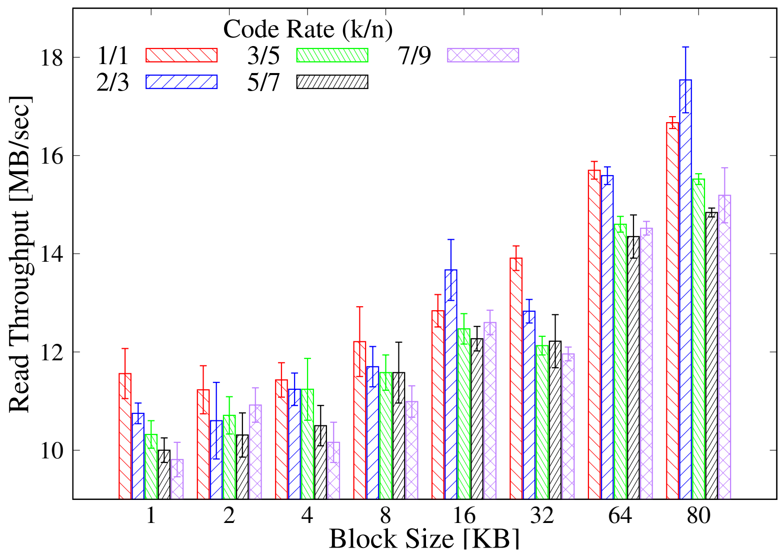
(a)



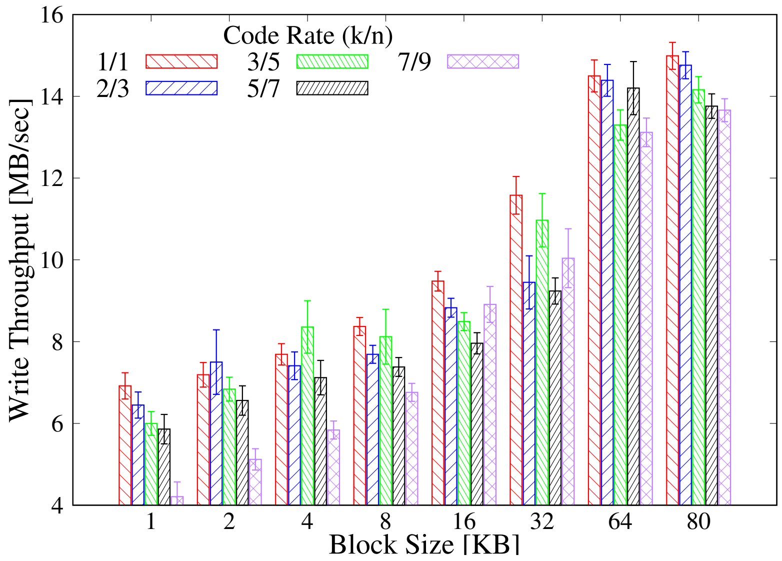
(b)

Figure 5.7: Data read and write throughput as a function of block size, for 0.5 link availability

time. During sending data, if a link was turned off, RSock running in sender device caches the data until the Wi-Fi connectivity is reestablished. We compared *R-Drive* data sharing against pure implementation of TCP in Android devices. We used SocketChannel class in *java.nio.channels* package to implement a TCP client-server application for Android. The implementation ensured that when sender side detected connection termination, it immediately tried to reconnect to receiver



(a)



(b)

Figure 5.8: Data read and write throughput as a function of block size, for 1.0 link availability

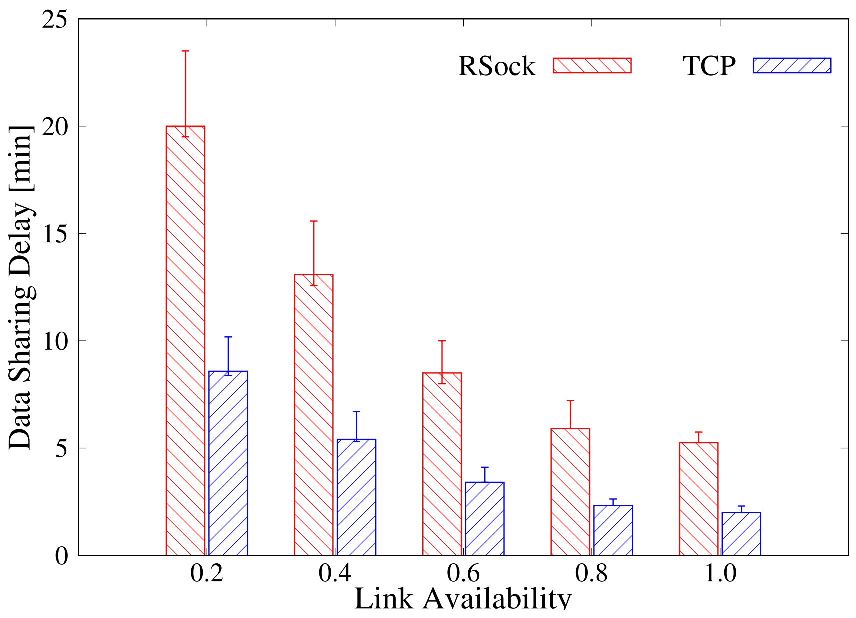


Figure 5.9: *R-Drive* Data sharing delay for total of 900MB data over RSock for variable link availability

indefinitely to finish sending remaining data. The experiment shows that compared to TCP, RSock data sharing delay is quite significant with high upper error when link availability is low. This is due to the fact that, RSock network topology learning time increases dramatically for highly sparse network connectivity. However, for higher link availability, RSock data sharing delay reduces significantly.

5.7 R-Drive Overhead

5.7.1 Memory Footprint

We traced real-time memory footprint for *R-Drive* using Android Profiler [52] during file storage and retrieval process. The purpose behind memory tracing is to identify memory leak in *R-Drive* Android application due to repeated memory allocation and deallocation. *R-Drive* is a write-heavy storage system, so it is important to observe whether the *R-Drive* application causes memory leak over extended runtime. For this experiment, we used a 20MB file and 1MB block size with (k, n) values of (10,20). Table 5.2 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 5.2: Number of allocated and de-allocated objects for different numbers of file creation and retrieval

5.7.2 Energy Consumption

Table 5.3 shows *R-Drive* application average energy consumption for continuous workload in different Android devices.

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

For this experiment, we started with 100% energy in each phone, ran DistressNet-NG application suit until the phones turned off due to battery exhaustion. We used Battery Historian [53] to pull Android battery usage data from Android devices after each experiment run. In each device, we ran EdgeKeeper, RSock and *R-Drive* along with a fourth client application that continuously performed random file creation and retrieval in *R-Drive* system. We deduce that, if similar devices are used in field, first responders may need to switch the device battery after approximately 3.5 hours. However, with newer Android devices with higher battery capacity, the runtime may increase.

5.7.3 Processing Overhead

We measured processing time for components responsible for encryption key generation, data encryption and data erasure coding, as shown in Table 5.4. We conducted experiments for variable block sizes such as 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 80MB and presented the percentage of average delay for each component. We observed that, data encryption takes the majority amount of processing time. For future implementation we plan to minimize data encryption delay by incorporating more robust library.

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

Table 5.4: Processing overhead as percentage of the total delay that different components are responsible for

5.7.4 Algorithm Execution Time

Table 5.5 shows the average algorithm execution time in milliseconds on a Samsung S8 Android device. The average is taken over 1000 iterations for each network size.

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

Table 5.5: Execution time of the Adaptive Coding algorithm in Samsung S8 for different network sizes (NS)

As shown, the execution time of the algorithm is rather negligible.

6. CONCLUSIONS AND FUTURE WORK

We have developed a device and network failure resilient data storage and sharing framework for disconnection oriented mobile edge that can operate in a wide variety of network connectivity. We presented the design of R-Drive, with detailed explanation of how R-Drive resiliently store and share data leveraging edge devices in an environment where network connectivity 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 how to transfer fragments from one vulnerable device to a safe one over opportunistic network before device failure takes place and data becomes lost/unavailable.

REFERENCES

[1] G. Otto, “DHS sees wearables as the future for first responders.” `https://www.fedscoop.com/dhs-wearables-first-responders/`, 2014. 2021-10-25.

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

[3] 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, pp. 1275–1284, 2018.

[4] X. Sun and N. Ansari, “EdgeIoT: Mobile edge computing for the internet of things,” *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

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

[6] N. K. Ray and A. K. Turuk, “A framework for post-disaster communication using wireless ad hoc networks,” *Integration*, vol. 58, pp. 274–285, 2017.

[7] R. Olaniyan, O. Fadahunsi, M. Maheswaran, and M. F. Zhani, “Opportunistic edge computing: Concepts, opportunities and research challenges,” *Future Generation Computer Systems*, vol. 89, pp. 633–645, 2018.

[8] 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.

[9] Z. Lu, X. Sun, and T. La Porta, “Cooperative data offload in opportunistic networks: From mobile devices to infrastructure,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3382–3395, 2017.

[10] 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.

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

[12] D. Reina, M. Askalani, S. Toral, F. Barrero, E. Asimakopoulou, and N. Bessis, “A survey on multihop ad hoc networks for disaster response scenarios,” *International Journal of Distributed Sensor Networks*, vol. 11, no. 10, p. 647037, 2015.

[13] F. Calabrese, L. Ferrari, and V. D. Blondel, “Urban sensing using mobile phone network data: a survey of research,” *ACM Computing Surveys (csur)*, vol. 47, no. 2, pp. 1–20, 2014.

[14] A. Rahman, E. Hassanain, and M. S. Hossain, “Towards a secure mobile edge computing framework for hajj,” *IEEE Access*, vol. 5, pp. 11768–11781, 2017.

[15] “Dropbox.” `https://www.dropbox.com/?landing=dbv2`. 2021-10-25.

[16] “Google drive.” `https://www.google.com/drive/`. 2021-10-25.

[17] “Microsoft OneDrive.” `https://www.microsoft.com/en-us/microsoft-365/onedrive/misc-cloud-storage`. 2021-10-25.

[18] 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.

[19] “ArcGIS Survey123.” `https://survey123.arcgis.com/`. 2021-10-25.

[20] CivTak, “Civtak/atak.” `https://www.civtak.org/`. 2021-10-25.

[21] S. Bhunia and R. Stoleru, “EdgeKeeper: Resilient and lightweight coordination for mobile edge computing systems.” `https://github.com/msagor/EdgeKeeper`, 2021. 2021-10-25.

[22] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, “A global name service for a highly mobile internetwork,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 247–258, 2014.

[23] A. Altaweel and R. Stoleru, “Rsock: A resilient routing protocol for mobile fog/edge networks.” `https://github.com/msagor/RSock`, 2021. 2021-10-25.

[24] C. Yang and R. Stoleru, “Hybrid routing in wireless networks with diverse connectivity,” in *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pp. 71–80, 2016.

[25] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[26] 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.

[27] 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)*, pp. 1–9, IEEE, 2019.

[28] D. Dwyer and V. Bharghavan, “A mobility-aware file system for partially connected operation,” *ACM SIGOPS Operating Systems Review*, vol. 31, no. 1, pp. 24–30, 1997.

[29] 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.

[30] 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.

[31] 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.

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

[33] C. 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.

[34] "Google files." `https://files.google.com/`. 2021-10-25.

[35] "Shareit." `https://play.google.com/store/apps/anyshare`. 2021-10-25.

[36] Z. Zhang, A. Wang, K. Zheng, G. U. Maheswara, and B. Vinayakumar, "Introduction to HDFS erasure coding in apache hadoop," *Cloudera Engineering Blog*, 2015.

[37] 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.

[38] 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)*, pp. 213–226, 2015.

[39] 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)*, pp. 212–215, IEEE, 2020.

[40] 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.

[41] 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.

[42] R. Housley, “Public key infrastructure certificate and certificate revocation list (crl) profile,” *RFC 3280-Internet X. 509*, 2002.

[43] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. T. Polk, *et al.*, “Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile.,” *RFC*, vol. 5280, pp. 1–151, 2008.

[44] “Google backup and sync.” `https://support.google.com/drive/answer/2374987`. 2021-10-25.

[45] “ZooKeeper programmer’s guide.” `https://zookeeper.apache.org/doc/r3.4.6/zookeeperProgrammers.html`, 2013. 2021-10-25.

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

[47] “Hadoop hdfs commands.” `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html#dfs`. 2021-10-25.

[48] “secretsharejava.” `https://sourceforge.net/projects/secretsharejava/`. 2021-10-25.

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

[50] B. Beach, “Backblaze open sources reed-solomon erasure coding source code.” `https://www.backblaze.com/blog/reed-solomon/`, 2015.

[51] “Baicells nova 227 enb.” `https://www.doubleradius.com/baicells-nova-227-outdoor-tdd-enb-basestation`. 2021-10-25.

[52] M. Jordan, “Android profiler,” in *`https://developer.android.com/studio/profile/android-profiler`*, 2022. Accessed Oct. 25, 2021 [Online].

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