DistressNet-NG: A Resilient Data Storage and Sharing Framework for Mobile Edge Computing in Cyber-Physical Systems

MOHAMMAD SAGOR, AMRAN HAROON, and RADU STOLERU, Department of Computer Science and Engineering, Texas A&M University, USA

SUMAN BHUNIA, Department of Computer Science and Software Engineering, Miami University, USA

ALA ALTAWEEL, Department of Computer Engineering, University of Sharjah, UAE

MENGYUAN CHAO and LIUYI JIN, Department of Computer Science and Engineering, Texas A&M University, USA

MAXWELL MAURICE and ROGER BLALOCK, National Institute of Standards and Technology (NIST), USA

Mobile Edge Computing (MEC) has been gaining a major interest for use in Cyber-Physical Systems (CPS) for Disaster Response and Tactical applications. These CPS generate a very large amount of mission-critical and personal data that require resilient and secure storage and sharing. In this article, we present the design, implementation, and evaluation of a framework for resilient data storage and sharing for MEC in CPS targeting the aforementioned applications. Our framework is built on the resiliency of three main components: EdgeKeeper, which ensures resilient coordination of the framework's components; RSock, which provides resilient communication among CPS's nodes; and R-Drive/R-Share which, leveraging EdgeKeeper and RSock, provides resilient data storage and sharing. EdgeKeeper employs a set of replicas and a consensus protocol for storing critical meta-data and ensuring fast reorganization of the CPS; RSock decides an optimal degree for replicating data that is communicated over lossy links. R-Drive employs an adaptive erasure-coded and encrypted resilient data storage; R-Share, leveraging RSock provides resilient peer-to-peer data sharing. We implemented our proposed framework on rapidly deployable systems (e.g. manpacks, testMobile Edge Clouds) and on Android devices, and integrated it with existing MEC applications. Performance evaluation results from three real-world deployments show that our framework provides resilient data storage and sharing in MEC for CPS.

ACM Reference Format:

Mohammad Sagor, Amran Haroon, Radu Stoleru, Suman Bhunia, Ala Altaweel, Mengyuan Chao, Liuyi Jin, Maxwell Maurice, and Roger Blalock. 2023. DistressNet-NG: A Resilient Data Storage and Sharing Framework for Mobile Edge Computing in Cyber-Physical Systems. 1, 1 (December 2023), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cyber Physical Systems (CPS) are being deployed in Disaster Response and Tactical environments to improve situational awareness, coordination, and decision-making in high-stress situations. As shown in Figure 1, CPS designed for disaster

Authors' addresses: Mohammad Sagor, msagor@tamu.edu; Amran Haroon, amran.haroon@tamu.edu; Radu Stoleru, stoleru@cse.tamu.edu, Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA; Suman Bhunia, bhunias@miamioh.edu, Department of Computer Science and Software Engineering, Miami University, Oxford, OH, USA; Ala Altaweel, aaltaweel@sharjah.ac.ae, Department of Computer Engineering, University of Sharjah, Sharjah, UAE; Mengyuan Chao, chaomengyuan@tamu.edu; Liuyi Jin, liuyi@tamu.edu, Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA; Maxwell Maurice, maxwell.maurice@nist.gov; Roger Blalock, roger.blalock@nist.gov, National Institute of Standards and Technology (NIST), Boulder, CO, USA.

Manuscript submitted to ACM

response, can be used to monitor, collect data, process the data, and provide analytics to disaster responders in Search and Rescue or Firefighting missions. By using sensors, drones, and other technologies, CPS can help emergency responders locate survivors, assess the damage, and coordinate resources. For example, CPS can help monitor the condition of bridges, roads, and other infrastructures to ensure their safety and functionality. In Tactical Environments, CPS can be used in military or law enforcement settings to improve situational awareness, decision-making, and coordination. For example, CPS can be used to monitor the movements of enemy forces, track the location of friendly forces, and coordinate the deployment of resources. CPS can also be used to detect and respond to threats such as chemical or biological attacks. However, it is important to ensure that these systems are reliable and secure to prevent any potential failures or cybersecurity risks. One important aspect of the reliability of such CPS is the resilient data storage and sharing, by adaptively storing data, when considering devices' reliability, device mobility, and the locations for computation functions (i.e., analytics).

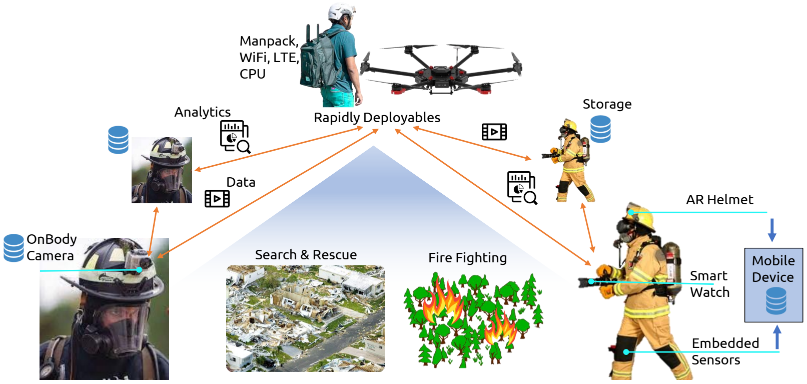


Fig. 1. Cyber-Physical Systems (CPS) collecting and processing essential data for first responders deployed in Search and Rescue or firefighting missions. Disaster Responder's equipment consists of a set of mobile devices (AR helmet, on-body/on-drone cameras, embedded sensors, mobile phones) [34]. Rapidly deployable units enable communication, sharing of data, and the results of computation (i.e., "analytics" in the figure).

Mobile Edge Computing (MEC) has been gaining significant interest for CPS deployed in Disaster Response and Tactical applications. As shown in Figure 2, several spatially close mobile edge devices form an edge cluster under effective coordination [11]. 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 (High-Performance Computing) node that manages communications (e.g., LTE, WiFi, WiFi Direct), allocates IPs, or provides DNS and device naming services. Data can be offloaded to HPC and other connected devices for processing and storage. As shown in Figure 2, two MECs (where nodes "HPC-1" and "M-6" serve as master nodes for edges 1 and 2, respectively) can provide cloud-like services intra-edge as well as inter-edge.

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 [37]. Existing commercial data storage services, e.g., Dropbox [18], Google Drive [22], OneDrive [33], 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

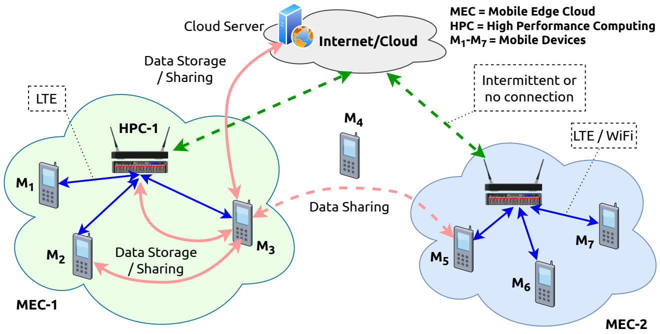


Fig. 2. MEC architecture, where mobile devices form two Mobile Edge Cloud (MEC) networks MEC-1 (with HPC infrastructure) and MEC-2 (ad-hoc), and share resources among themselves, or with the cloud

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, WiFi 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.

Users can use file-sharing applications (Google Files [23], SHAREit [31] etc.) that do not require cloud connectivity and can operate over ad-hoc networks such as WiFi 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 divided into groups and multiple groups are deployed at disjoint geographical locations, so it will be impractical for team members to sacrifice their mobility and stay within each others network range for the sake to exchanging data. Consequently, MEC platforms for disaster response should support network opportunistic data sharing over multiple hops.

To address the mentioned limitations, we present DistressNet-NG, a framework for resilient data storage and sharing for MEC in Disaster Response and Tactical CPS. The DistressNet-NG software and hardware ecosystem has been developed and field tested for the purpose of aiding search and rescue operations for disaster response. This research is an extension of preliminary conference versions appearing in the Proceedings of the IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS'22) [11] [39] [3], with the following contributions:

- **R-Drive**, a resilient data storage framework for MEC in CPS. R-Drive employs a novel adaptive erase coding technique that is implemented and deployed on real systems, validating its suitability. R-Drive's resilience is supported by its integration with a resilient edge coordination service (EdgeKeeper) and a resilient communication framework for dynamic, disconnected MEC environments (RSock).
- **R-Share**, a resilient message and file sharing framework for MEC in CPS. R-Share's resilience is supported by its integration with a resilient edge coordination service (EdgeKeeper) and a resilient communication framework for dynamic, disconnected MEC environments (RSock).
- The **implementation**, **integration** and **real-world deployments** of DistressNet-NG (i.e., R-Drive, R-Share, EdgeKeeper, and RSock). We present performance evaluation of both indoor and outdoor experiments with different hardware configurations in variable network conditions to demonstrate resiliency of data storage and sharing.

The rest of the article is structured as follows. In Section 2, we present state-of-the-art research and commercial solutions for data storage and sharing. In Section 3, we present the design of DistressNet-NG components for resilient data and sharing (R-Drive and R-Share), and their integration with EdgeKeeper and RSock. In Section 4, we present details about the hardware and software implementations for DistressNet-NG components, while in Section 5 we present the performance evaluations from real system implementations and deployments. We conclude in Section 6 with a summary of contributions and ideas for future work.

2 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 is distributed among devices that are prone to frequent disconnections and failures. This raw and processed data needs to be readily available to the MEC 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, hence when device storage runs out, these services become inoperational, despite other devices in same network having large amounts of available storage.

Data sharing is equally important for MEC. Applications like Google Files and SHAREit, allow users share files over ad-hoc networks (WiFi 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. 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. 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 1 summarizes the limitations of existing storage and sharing solutions that make them impractical for MEC environments.

	Distributed Offline Data Storage	**Opportunistic Data Share**	**Cloudless Namespace Sync**
Dropbox	×	×	×
OneDrive	×	×	×
Google Drive	×	×	×
Google Files	×	×	×
SHAREit	×	×	×
R-Drive	✓	✓	✓

Table 1. Existing data storage and sharing services cannot fulfill edge requirements

Other solutions that target resilient data storage primarily in the cloud do not apply to MEC. OFS [35], HDFS [45] and GFS [20] are too heavy-weight either for storage overhead, memory footprint, or computation overhead. MEFS [41] does not work in absence of the cloud. PFS [19], FogFS [36] rely on specific mobility models that may be impractical for

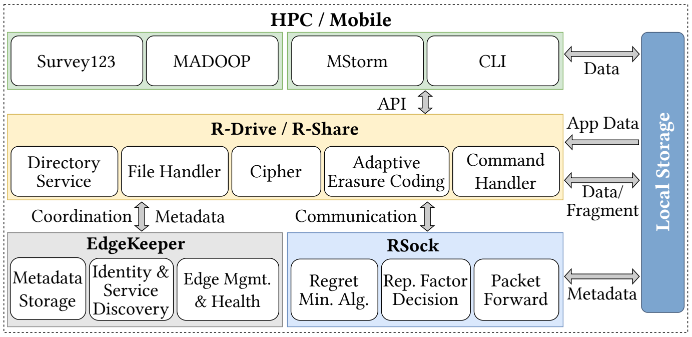


Fig. 3. R-Drive/R-Share: components and their integration with the DistressNet-NG software ecosystem, including EdgeKeeper, RSock, and MEC applications for disaster response: MStorm [13], Survey123 [7] among others.

MEC that is disconnected from the cloud for long periods of time. Hyrax [32] ports HDFS to Android but shows poor performance for CPU-bound tasks. While MDFS [14, 15], the earlier version of R-Drive, 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.

Reed–Solomon erasure coding [38] 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 note that (n, k) should be decided dynamically depending on resource availability in MEC and on the 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. [49] presented an online adaptive code rate selection algorithm for cloud storage that considers real-time user demands optimum (n, k). However, this solution assumes that all candidate storage devices have enough storage capabilities. HACFS [46] implements an extension to HDFS to adaptively choose between two (fast and compact code) coding schemes but their solution involves fixed coding parameters for each of the coding schemes. Other researchers [44, 48] also proposed solutions for erasure coding-based data storage, yet they do not address how to choose n and k dynamically.

The main technical challenge that the aforementioned solutions face when considering resilient data storage and sharing is that it is not clear how to embed "resiliency" into a complete MEC for CPS. Individual components that were designed with resiliency in mind do not automatically (and equally important optimally) integrate well together. In this manuscript we propose an integrated design and implementation of a complete solution. To the best of our knowledge, no solution exists that is end-to-end/complete (i.e., takes into account all aspects of resiliency, e.g., coordination, and communication) and that provides resilient data storage and sharing in MEC for highly dynamic environments (such as disaster response or tactical), where device failures/disconnections are frequent.

3 DISTRESSNET-NG FRAMEWORK DESIGN

DistressNet-NG [3, 11, 13, 16, 24] is a next-generation MEC system for disaster response. It provides cloud-like functions (e.g., resilient data storage, real-time stream processing and batch processing) to MEC applications.

The software architecture for the DistressNet-NG's components that provide resilient data storage and sharing (R-Drive and R-Share) is shown in Figure 3.

Two important components of the DistressNet-NG architecture are EdgeKeeper and RSock (Resilient Sockets). EdgeKeeper [11] ensures coordination among all the devices in the edge network in a distributed manner and provides services like identity and naming, authentication, service discovery, metadata storage, and edge status monitoring. RSock [3] is a resilient transport protocol designed for sparsely connected network environments aiming to make efficient use of available network bandwidth and to ensure timely data delivery. RSock performs multipath packet routing over available network interfaces such as LTE, WiFi-Direct, and WiFi.

DistressNet-NG also implements data processing frameworks (not presented in this article): a) a real-time stream processing, through MStorm [13], similar to what Apache Storm [6] provides in the cloud; and b) batch processing, though MADOOP, similar to Apache's HADOOP platform in the cloud. These data processing frameworks may employ resilient data storage and sharing functions of DistressNet-NG. Survey123 [7], a GIS application used by disaster responders, was integrated with R-Drive and R-Share. A Command Line Interface (CLI) also provides file system-like functions for managing data in MEC.

In the remaining part of this section, we present the design of the main components that ensure resilient data storage and sharing in DistressNet-NG: EdgeKeeper, RSock, R-Drive, and R-Share.

3.1 EdgeKeeper - Resilient Coordination

All devices run EdgeKeeper as a background process that coordinates with other devices in the edge network (as shown in Figure 3). Other edge applications such as R-Drive, RSock, and MStorm interact with EdgeKeeper running on the same device through predefined Application Programming Interface (API) calls. Unfortunately, conventional coordination services such as Apache ZooKeeper [5] fail to operate in mobile edge environments. ZooKeeper is designed for a cloud-like environment where the IP addresses of the consensus-maintaining servers remain static, and the link quality between them is good. In a deployable CPS environment, nodes face frequent changes in IP address and node and link failures. Moreover, any consensus mechanism requires frequent message exchanges, which may be hindered by the quality degradation of wireless links in a deployable environment. In ZooKeeper, the server node configuration, which runs the consensus, must be static. If the ZooKeeper ensemble loses the majority of the server nodes, the whole ensemble fails to work. Thus, the ZooKeeper at its current state is not suitable for distributed edge computing at the edge. We developed a resilient, distributed coordination service for edge computing, namely EdgeKeeper. EdgeKeeper is an application that runs on all the devices in the background and provides resilient coordination for other applications. EdgeKeeper runs a peer node discovery and link quality monitoring system. When some nodes leave the network, the EdgeKeeper ensemble dynamically chooses new nodes to participate in the consensus and reconfigure the edge network.

In this section, we briefly present each module of EdgeKeeper below, as shown in Figure 3. For a detailed description of the four modules, we refer the readers to [11].

3.1.1 Resilient Metadata Storage. Many distributed applications running on an edge network require a database to store critical coordination information. For example, R-Drive needs to store the directory structure information, and RSock needs to store the device-ID-to-address mapping. In an edge network, storing this critical information on a single node is

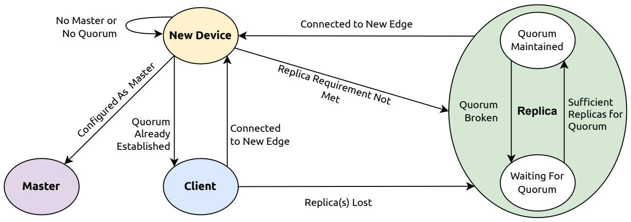


Fig. 4. EdgeKeeper state transition diagram, from when a device is starting to when consensus/quorum is achieved or broken, through devices leaving or joining the edge.

undesirable as devices frequently become inoperable. To safeguard against these failures, EdgeKeeper stores critical data over multiple replicas and maintains consensus among them. It uses three types of device roles: *master*, *replica*, and *client*. A master node acts as the gateway node for forming the edge network. Often the HPC or the WiFi group owner/leader (in the case of WiFi Direct) acts as the master node. As shown in Figure 2, there are two edge networks. Here HPC-1 and HPC-2 act as masters for their respective networks. By default, all other devices act as client devices. The clients read and write critical edge coordination data from one of the replicas that participate in consensus. An administrator decides the required number of replicas and, implicitly, the size of the quorum (50% or more available replicas needed for quorum). Depending on the network conditions (link and device state), the master node decides which devices are replicas and take part in the consensus. For a network with a replica configuration of r, a consensus could be reached as long as more than $r/2$ replicas are operational. The more replicas, the higher the fault tolerance. However, increasing the number of replicas also requires more devices to participate in the quorum. Thus, the administrator must analyze the deployment scenario and decide on the replica configuration, r.

Figure 4 shows the states of a device after joining an edge network. An administrator chooses the master and configures it before forming an edge network. The master runs a local DNS server for the edge network to function without connectivity to the cloud. Any node that wishes to join an Edge cluster first finds the edge master (through a specially crafted DNS query) and initiates the coordination to join the edge cluster. The master also observes the edge network topology and dynamically decides which device to serve as replicas based on link quality. A master dynamically tries to meet the number of replicas the administrator sets. When a new device joins an edge network, it stays inoperable until the replica ensemble reaches the quorum. The new device acts as a replica or client based on the master's decision.

R-Drive stores the file and directory metadata on an EdgeKeeper replica ensemble through a local to-the-node EdgeKeeper process. This metadata contains information about a file (on which devices the fragments of a file are stored), and the directory structure. The directory structure is stored in a hierarchical structure of node objects (similar to Linux iNodes) where each node represents a specific directory and the root node of the tree represents the root directory for the local edge network.

3.1.2 Identity and Service Discovery. EdgeKeeper provides resilient device naming for edge networks using a Global Naming Service (GNS) [43] and assigns a globally unique identifier (GUID) to each device. EdgeKeeper maintains a cache of GUID records on the replica ensemble to preserve the name resolution service when an edge network gets disconnected from the federated GNS servers. The name record updates are committed to the local cache and lazily updated to the GNS server whenever the connection to the Internet is restored.

After joining an edge network, applications such as R-Drive/R-Share first try to discover other devices running the same applications by querying EdgeKeeper. If a device offers a service, the service name and the role are stored as part of the GUID record. Each GUID record contains an associative array of key-value pairs, as: `GUID: <own GUID>, alias: <host name for DNS>, netaddress: [<node's IP-1> , <node's Ip-2>], <application-name 1>: <application-role 1>, last-update: <node's system time>`. Any node wishing to find a list of nodes offering a particular service will query EdgeKeeper to retrieve a list of GUIDs containing the key-value pair as `service:role`. The API for service discovery is as follows: `addService(ownService, ownDuty), removeService(targetService), getPeerGUIDs(targetService, targetDuty)`.

3.1.3 Edge Monitoring and Management. EdgeKeeper runs a topology discovery service to learn the network topology and link statuses. Each device periodically pings other devices in the network and assesses device-to-device link qualities by measuring the expected number of transmissions required for a packet to be successfully transmitted and acknowledged (ETx). The ETx of a path between nodes A and B is calculated as follows: $ETx_{AB} = \frac{1}{(1-PDR_{A\to B})(1-PDR_{B\to A})}$ where $PDR_{A\longrightarrow B}$ is the packet loss rate from A to B and $PDR_{B\longrightarrow A}$ is the packet loss rate from B to A. EdgeKeeper maintains a network graph that contains available nodes and the links among the nodes. When multiple wireless links are possible between two devices (e.g., WiFi and LTE), EdgeKeeper maintains information about all links. R-Drive fetches the topology graph from EdgeKeeper and uses this link quality information to decide where to store the data fragments intelligently. The API for obtaining network topology information is as follows: `getNetworkInfo(), getAllLocalGUID()`.

In addition to network topology, edge applications require the capabilities of peer devices, such as the number of functioning processors, available memory, remaining battery, and available storage. EdgeKeeper running on each device periodically measures the device resources and reports them to the Edge master. Any client application can obtain edge health status using the API calls: `putAppStatus(appName, appStatus), getAppStatus(targetGUID, appName ), getDeviceStatus(targetGUID)`.

3.2 RSock - Resilient Communication

Resilient Sockets (RSock) is a hybrid routing protocol that investigates the benefit of packet replication in terms of packet delay reduction, as it is the key to decide when and how much replication should be used [3]. Based on real-world exercises of first responders during a wide area search, we observed that the devices, in Disaster Response and Tactical applications, often exhibit diverse connectivity characteristics [3]. These diverse characteristics have not been addressed by previous routing protocols [2, 10, 12, 17, 25–27, 29, 30]. Also, these protocols have not been widely deployed due to the middleboxes and firewalls in today’s wireless networks architecture. For instance, the latest releases of OLSR and Prophet [1, 10, 30, 40] require OS upgrades and/or root privileges to be able to run. Hence, RSock is designed as a user-space protocol to handle packets routing under diverse connectivity scenarios in CPSs and it leverages UDP for its data and control packets (i.e., to facilitate its deployment, iterative modifications, and future releases). Before we proceed in describing the main modules of RSock, we introduce three main parameters that were used in its design:

- *Inter-contact time (ICT)*, which is the time duration between two contact events between a pair of nodes.
- *Replication factor, rf*, which is the total number of data copies created at the source for a given packet.
- *Replication gain, γ_{rf}* $= E[D_1]/E[D_{rf}]$; whereas, D_1 and D_{rf} are the random variables for routing delay when the replication factor is 1 and rf, respectively, and E represents the expected routing delay.

RSock mainly consists of four modules, as shown in Figure 3, *Regret Minimization Algorithm*, *Replication Factor Decision*, *Packet Forwarding*, and *Communication API*. In this article we briefly present each module; for a detailed description of the four modules, we refer the readers to [3].

3.2.1 Regret Minimization Algorithm. We presented in [3] a mechanism to capture any potential correlation among the ICT of different paths between the sender node and destination node and how this correlation can affect the replication gain, γ_{rf}. Basically, RSock estimates the ICT among the nodes as well as the path delay correlation and feeds them into the Regret Minimization Algorithm, which dynamically decides the appropriate value of rf, i.e., the packet replication. The Regret Minimization Algorithm aims to obtain the best rf while it gains more information about the network. That is, it uses a probability distribution to represent the preference of choosing a particular value for rf. Afterward, and based on the probability distribution, the algorithm draws the value of rf such that it is able to seamlessly switch between a single-copy to a multi-copy protocol according to the current connectivity conditions in the network.

3.2.2 Replication Factor Decision. This module aims to calculate and maintain the replication factor probability distribution. It relies on the packet delays for any given rf in order to derive its corresponding γ_{rf}. The packet delays are collected through the Round Trip Time (RTT) from the direct feedback of destination nodes. That is, based on the *ack* the destination node sends upon receiving its packets. The delay information is collected for different values of rf. Once two Replication Factor Decision modules in two different nodes discover each other, they start to periodically exchange probes to collect delay information for different values of rf. The probes are sent via rf paths specified by Yen's algorithm [47], which is employed by the Replication Factor Decision module to find multiple shortest paths to the destination nodes. Once the module in the receiver node receives a probe, it immediately sends back a reply. Upon the reception of the reply, the Replication Factor Decision module in the sender node uses the RTT to calculate the replication factor distribution.

3.2.3 Packet Forwarding. This module implements the forwarding mechanism based on the measured delivery capability, which is built based on the potential ICT correlation between different nodes in the network. It draws the values of rf using the replication factor probability distribution that is produced by the Replication Factor Decision module and then forwards the number of packet copies specified to their appropriate packet carriers. That is, by augmenting the packet header with the following four fields: 1) the $nrofCopies$ field, which represents the remaining number of data copies. 2) the $nextCarrier$ field, which specifies the intermediate destination for the packet (i.e., this field is set by the current carrier when it forwards the packet). 3) the $carriers$ field, which contains a list of packet carriers that currently hold the packet. 4) the $path$ field, which contains a list of nodes that specifies the shortest path leading to the $nextCarrier$ in the network. When any node in the network receives a packet, it checks if it is the destination. If that is the case, it sends back an *ack* that contains the delay of the packet. Otherwise, it adds itself to the $carriers$ field (if it is the $nextCarrier$ field) or it forwards the packet to the next hop according to the $path$ field (if it is not the $nextCarrier$ field).

3.2.4 Communication API. RSock Communication module handles the interaction between RSock and its applications (such as R-Drive or R-Share) in the same way as a TCP socket is used, through the following API calls:

1. Registration API call, in which R-Drive or R-Share registers with RSock. The registration is mainly performed via $Local_{ID}$ and TTL fields. $Local_{ID}$ is a 40-char string ID obtained from EdgeKeeper [11] and used in the sender field of data packets. TTL is a QoS parameter for the packets. TTL indicates the maximum Time-to-Live, in seconds/hours, that the sender is willing to tolerate before the packets get received by the final destination. If any packet's TTL expired before it reaches the final destination, RSock drops it.
2. Transmission API call that is used when R-Drive or R-Share has some data to be sent. This API call contains the bytes of the application data to be sent, the data size, as well as the ID of the destination device, which is obtained from EdgeKeeper.
3. Receiving API call. A zero-parameter blocking call in which R-Drive or R-Share waits to obtain any receiving data from RSock. This call mainly returns the received data (and its size) as a byte array.

3.3 R-Drive/R-Share - Resilient Data Storage and Sharing

The R-Drive/R-Share system architecture (with its five major components and their integration with the DistressNet-NG software ecosystem) is shown in Figure 3. 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. In the remaining part of this section, we present the detailed designs for each of the components.

3.3.1 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 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 [21]. A user can select application directories that 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 [7]. In the following sections, we present in detail the design of the core components of R-Drive.

3.3.2 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*. The rnode data structure is shown in Figure 5a. An rnode represents either a file or a directory. After creating an rnode, the Directory Service stores it in EdgeKeeper which uses multiple replicas and consensus for resilience, as presented in Section 3.1.1. A directory creation also can take place when a client invokes the `mkdir()` API function or when the command `-mkdir` is executed in the CLI. Directory retrieval 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 [5] 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.

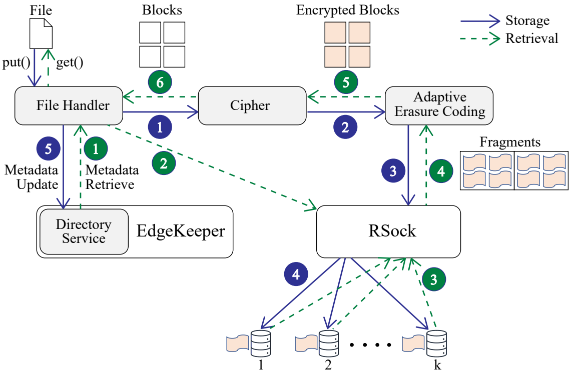
3.3.3 R-Drive: Resilient Data Storage. Data is stored in R-Drive as files. File creation involves copying a file from the local file system to R-Drive using the `put()` API or the `-put` command. Figure 5b shows the steps for file creation and retrieval processes in R-Drive. For file creation, a local 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 communication API. All fragments contain a timestamp that acts as a version number for fragments. A receiver device only accepts fragments with the same or higher timestamps. The Directory Service communicates with EdgeKeeper to create an rnode for the new file.

We are now presenting in detail the cyphering and adaptive erasure coding technique that R-Drive uses.

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) [42]. 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.

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)



(b)

Fig. 5. (a) R-Drive rnode structure; and b) R-Drive file storage and retrieval steps: partitioning the file into B blocks, encrypting them, applying the adaptive erasure coding, and distributing the fragments to the best suitable n nodes for storage. The retrieval process starts with a device requesting fragments from other devices, then restructuring all blocks when enough fragments are received.

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. 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 6a 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.

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 too?* and 3) *How to obtain the system parameters used in answering 1) and 2)?*

Q1: What k and n values? 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 express the cost of availability and storage C as a weighted sum and include them in the following minimization problem:

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

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

$$T \leq T_k, \tag{3}$$

$$1/N \leq k \leq n \leq N, k, n \in Z^+ \tag{4}$$

$$0 \leq w_a \leq 1 \tag{5}$$

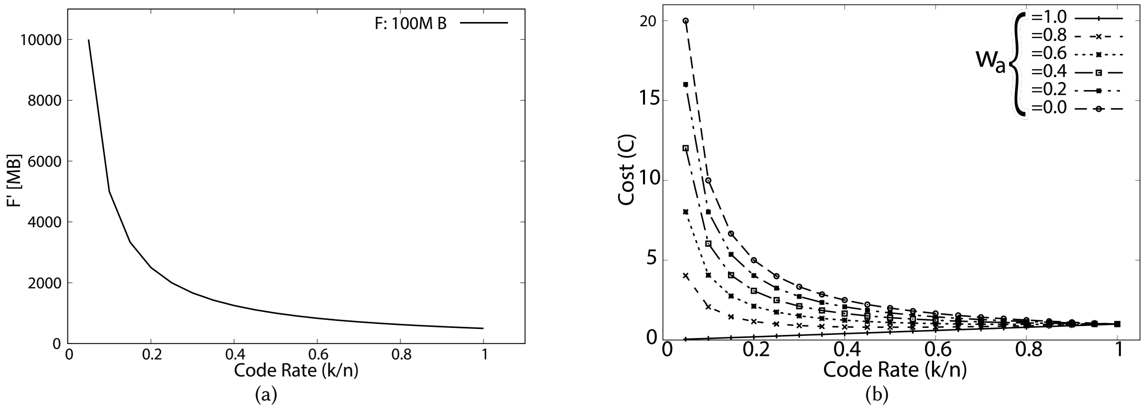
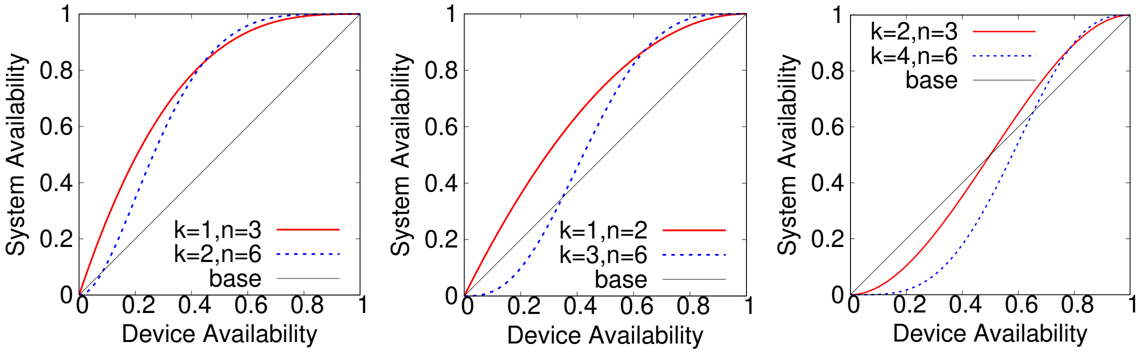


Fig. 6. a) File size F' after erasure coding (applied to a file F of size 100MB) as a function of the code rate (k/n); and b) Cost as a function of code rate for different w_a



(a) k/n=1/3

(b) k/n=1/2

(c) k/n=2/3

Fig. 7. Example of different (k, n) pairs determining different system availability. Each group of a, b, and c contains two (k, n) pairs of the same ratio. The baseline in each group represents pure local storage

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 choosing 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:

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

Figure 7a, 7b, 7c each contains two (k, n) pairs with the same ratio. As shown, when the code rate increases from 1/3 to 1/2 and then to 2/3, the system availability gradually decreases. Meanwhile, in each group, when the device

Table 2. 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

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.

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

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 (i.e., 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. For example, one client of R-Drive is MStorm, that takes real-time video recording by first-responders as input and performs face detection and recognition to produce victims' images. MStorm will save the victims' images in R-Drive with high w_a value since identifying victims by their facial images is crucial in disaster response and casualty estimation. Additionally, users can specify an approximate T because some data is time sensitive and does not serve any valuable purpose after a certain time has elapsed.

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

```
Input   : $F$, $T$, $S_i$, $T_i$, $w_a$
Output : (k,n) and n devices
$(k, n) \leftarrow (1, 1)$
$C_{min} \leftarrow 1$
for $n' \in 1...N$ do
    for $k' \in 1...n'$ do
        if Satisfying Eq.(3.2)(3.3) then
            if $C(k', n') < C_{min}$ then
                $(k, n) \leftarrow (k', n')$
                $C_{min} \leftarrow C(k', n')$
            end
            if $k'/n' = k/n$ then
                if $A(k, n, \overline{p}) < A(k', n', \overline{p})$ then
                    $(k, n) \leftarrow (k', n')$
                end
            end
        end
    end
end
$V \leftarrow$ pick up devices with $S_i > F/k$
sort $V$ based on $T_i$ in descending order
$V_n \leftarrow$ choose top $n$ devices with the largest $T_i$
return $(k, n)$ and $V_n$
```

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 6b. 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 2 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

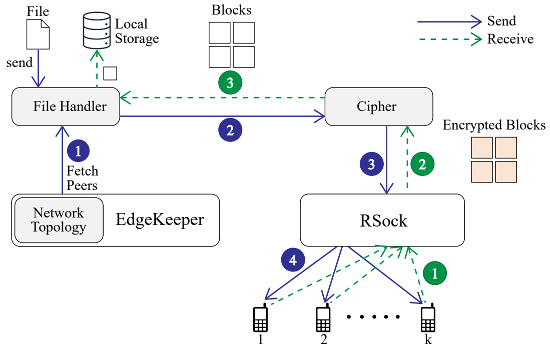


Fig. 8. R-Share file sharing steps: A file is divided into encrypted blocks and sent to other devices over RSock

and k 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 the code rate with the lowest cost (hence n and k) but also chooses devices with the minimum required storage and battery remaining time.

3.3.4 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 5b. 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 that 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, and sends fragment requests to the k devices. When k fragment replies are received, File Handler employs Erasure Coding and Ciphering for block decoding and decryption, respectively.

3.3.5 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. Command Handler consists of a hand-written lexer and parser. Lexer takes an input command as a text stream, converts it into a series of tokens and the parser converts the tokens into a parse tree. The parse tree enables Command Handler to identify the type of command.

3.3.6 R-Share: Resilient Data Sharing. R-Share is a peer-to-peer message/voice recording/file-sharing application that relies on RSock for resilient communication and on EdgeKeeper for service discovery. R-Share's architecture, shown in Figure 8, shares some common components with R-Drive such as File Handler, and Cipher but it does not rely on any Directory Service. File Handler runs as a background service and interacts withh RSock for continuously receiving new data packets and with EdgeKeeper's Identity and Service Discovery component for periodically fetching peers that are also running R-Share. To ensure a non-blocking experience for users, R-Share maintains two separate data queues for sending and receiving data to/from RSock; users can push data (e.g., message, voice recording, file) to the application without being blocked for job completion.

4 DISTRESSNET-NG FRAMEWORK IMPLEMENTATION

In this section, we present the hardware and software implementations for the DistressNet-NG framework. These implementations were used in four real-world deployments. Details for the real-world deployments and performance evaluations obtained from them are presented in Section 5.

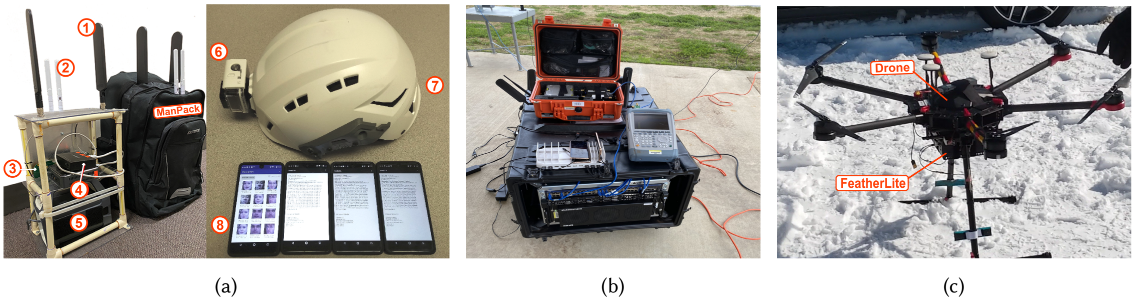
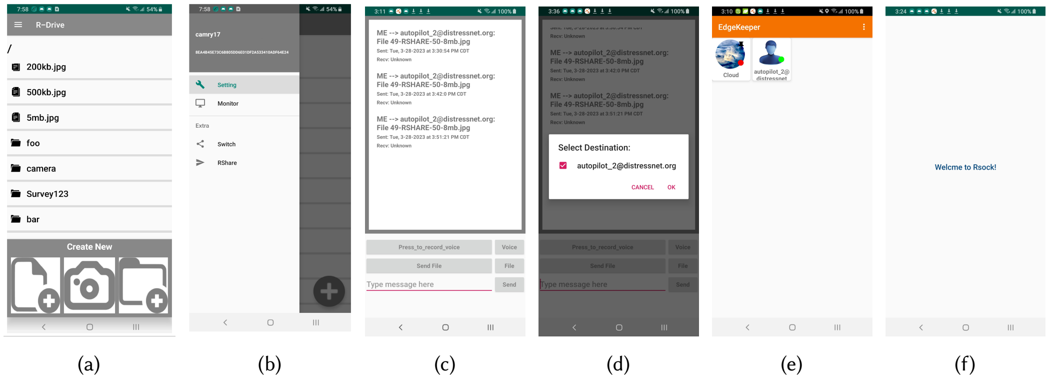


Fig. 9. DistressNet-NG hardware implementations: (a) Texas A&M manpack and edge computing equipment: from 1 to 8, they are LTE antenna, WiFi AP, LTE eNB, Intel NUC with LTE EPC and HPC, battery, camera, helmet, mobile phones; (b) NIST Rapidly Deployable; (c) VirtualNetCom Featherlite eNB mounted under a UAV (drone)



ME --> autopilot_2@distressnet.org:
File 49-RSHARE-50-8mb.jpg
Sent: Tue, 3-28-2023 at 3:30:54 PM CDT
Recv: Unknown

ME --> autopilot_2@distressnet.org:
File 49-RSHARE-50-8mb.jpg
Sent: Tue, 3-28-2023 at 3:42:0 PM CDT
Recv: Unknown

ME --> autopilot_2@distressnet.org:
File 49-RSHARE-50-8mb.jpg
Sent: Tue, 3-28-2023 at 3:51:21 PM CDT
Recv: Unknown

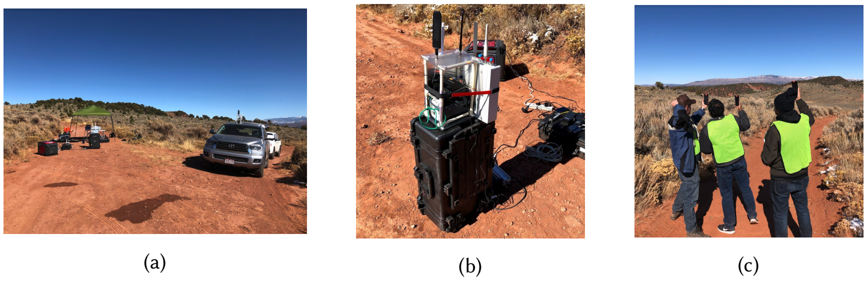
Select Destination:
☑ autopilot_2@distressnet.org
CANCEL OK

Welcme to Rsock!

Fig. 10. DistressNet-NG Android applications: a-b) R-Drive, allowing navigation of the distributed file system and providing capabilities to add/remove files and directories; c-d) R-Share showing peer selection window and file transfer report; e) EdgeKeeper showing available peers; and f) R-Sock, an application that runs in the background.

We implemented DistressNet-NG on three hardware platforms, as shown in Figure 9. The details for each platform are the following:

- **Texas A&M - Manpack**: The manpack, shown in Figure 9a, can be carried as a backpack by a first responder or a soldier and consists of communication, computation, and interconnect components. For communication, 4G LTE is provided by a Baicells Nova-227 eNB and WiFi by a Ubiquity UniFi Mesh Access Point. The computation platform is provided by an Intel NUC server. Communication among all components is enabled by a Ubiquity EdgeX router, while power for the components, is provided, through PoE, by onboard batteries. The LTE downlink and uplink data rates are 110 and 20Mbps, respectively [8]. The WiFi module is capable of providing around 100Mbps data transfer rate. The Intel NUC runs edge applications, including Open5GS, a 4G LTE EPC. Figure 9a also depicts typical user equipment that was used in all real-world deployments. The equipment consisted of on-body cameras (mounted on helmets) and Essential PH-1 Android mobile devices.
- **NIST - Rapidly Deployable**: Figure 9b shows NIST Public Safety Communications Research (PSCR) deployable system, which is equipped with LTE and WiFi communication capabilities. This system is 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.



(a) (b) (c)

Fig. 11. Gypsum, CO deployment (2018): a) deployment site; b) TAMU deployable; c) NIST and TAMU members performing experiments

- **VirtualNetCom - Featherlite LTE-in-a-Box** Figure 9c shows the Featherlite eNodeB deployed on a DJI M600 Pro multi-rotor drone. An Android Essential PH-1 mobile device configured with the DistressNet-NG software ecosystem is attached to Featherlite.

The DistressNet-NG software ecosystem, depicted in Figure 3, was implemented on Android mobile devices and Linux servers. MEC applications that were integrated with our DistressNet-NG software suite were Survey123 [7] and MStorm [13]. The developed DistressNet-NG software components are depicted in Figure 10. All DistressNet-NG software components were released in the public domain [28].

R-Drive and R-Share are implemented in about 10,000 lines of Java code and run, as shown, as Android apps and also as Linux servers. We used BackBlaze [9] Reed-Solomon erasure coding library and *javax.crypto* as the Cipher. As presented in Section 3, R-Share employs a simpler design than R-Drive; it does not incorporate Erasure Coding, Shamir, or Directory Service. Instead, it maintains a local database for indexing all sent and received messages and files. R-Drive/R-Share uses Java client libraries that implement the APIs for EdgeKeeper and RSock, thus enabling communication with the EdgeKeeper and RSock processes through JSON-based RPC over a local TCP socket.

The R-Drive and R-Share applications, shown in Figures 10a-10b and 10c-10d, respectively, provide resilient and secure data storage and data sharing. Figure 10a shows the file system navigation capabilities and adding/removing files/folders. Figure 10b shows configuration capabilities for R-Drive, including accessing R-Share. Figure 10c shows R-Share's interface for peer-to-peer sharing of files or text messages, while Figure 10d the GUID addressing for the destination node of R-Share.

The EdgeKeeper and RSock Android applications, shown in Figures 10e and 10f run on all devices of an edge cluster and are responsible for edge coordination, cluster formation and communication, respectively. Figure 10e shows the edge devices, their availability status as well as the status of the cloud connectivity. Figure 10f shows that RSock is just a daemon process.

5 PERFORMANCE EVALUATION

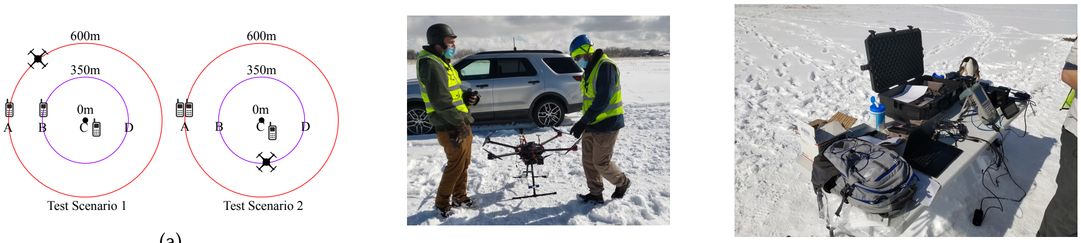
In this section, we present the evaluation for the implemented DistressNet-NG framework components that provide resilient data storage and sharing. The performance evaluation is based on three real-world deployments and indoor experiments. We also present simulation results that validate the analysis on which R-Drive's adaptive erasure coding technique is based on.

The real-world deployments where the R-Drive, R-Share, EdgeKeeper and RSock were evaluated were as follows:

- **Gypsum, CO - Wildfire Fighting (2018):** As shown in Figure 11, for a wildland-firefighting scenario, we deploy our DistressNet-NG in an open mountainous region near Gypsum, CO. During this deployment, TAMU deployable (Figure 9a) was used for assessing DistressNet-NG system's distance coverage by both WiFi and LTE

(a) (b) (c)

Fig. 12. Disaster City, TX, deployment (2020): a) Pre-deployment team preparation with initial TAMU deployable; b) TAMU deployable used in the simulated scenario; c) Victim images placed for victim's face detection as a part of search and rescue operation.



(a) (b) (c)

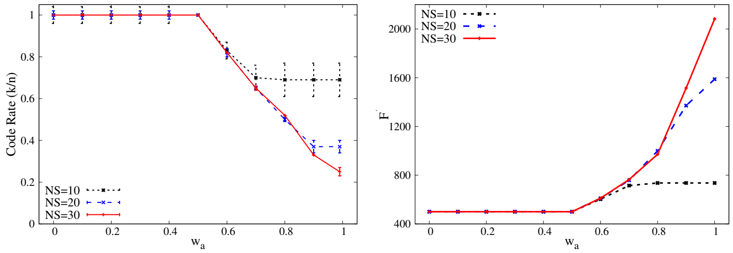
Fig. 13. Christman Airfield, CO deployment (2021): a) deployment scenarios for the mobile devices and drone's flight path; b) drone carrying eNodeB and Master Node mobile device; and c) setup station for system initialization and troubleshooting

networks, where the LTE was provided by the NIST deployable (Figure 9b). From this experiment, we obtained edge connectivity performance when distances among edge devices vary. Several components of DistressNet-NG architecture were improved based on the findings of this deployment.

- **Disaster City, College Station, TX - Wide Area Search and Rescue (2020):** We deployed DistressNet-NG in Disaster City, TX as part of a wide area search and rescue disaster response exercise, as shown in Figure 12. Our applications were tested in a simulated massive earthquake scenario by different Texas Task Fore 1 (TTF1) first responders to search for any survivors and assess the casualties. One first responder carried our manpack that provided wireless communication (LTE and WiFi) and a high-performance computing (HPC) unit for the team (Figure 12b). During this deployment, all processed data (e.g., victim's face data produced by an MStorm Face Detection application) were resiliently stored by R-Drive (i.e., in a local folder and successfully replicated in the cluster). During the 2020 deployment in Disaster City, the DistressNet-NG ecosystem collected a total of 14.4GB data from face detection and recognition that needed resilient and secure storage.
- **Fort Collins, CO - Drone Assisted Wildfire Fighting (2021):** To simulate a large-scale disaster response scenario where disconnected mobile devices on the ground would not be able to collaborate, we deploy our DistressNet-NG at the Christman Airfield in Fort Collins, CO where a UAV provides wireless connectivity and computation resources [24]. The VirtualNetCom Featherlite LTE-in-a-Box in Figure 9c is used for this deployment. We employed additional 12 mobile devices (Samsung Galaxy S9 and Sonim XP8), all running DistressNet-NG. Six additional Bittium Tough phones were used for logging LTE Reference Signal Received Power (RSRP). The drone followed two circular paths with a radius of 350m and 600m, respectively (Figure 13a). There are 12 flights in total, each flight took approximately 20min. From this deployment, we learned the feasibility of UAV-based deployment and understand the performance of our R-Drive/R-Share. We evaluated the *data read/write success rate, latency* for R-Drive and the *data delivery success rate, latency* for R-Share in a hostile network environment. We conducted

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) (c)

Fig. 14. a) Achieved cost (function $C(k, n, w_a)$ is dimensionless) for variable w_a and Network Size NS ; b) Effect of w_a on: a) code Rate (k/n); and b) file size F', for different network sizes, NS=10, 20 and 30

the deployment where four first responders were deployed in a wide area for a search and rescue mission. Each first responder carried a body-mounted camera that captures videos for face detection and recognition and transfers them through a node that was carried by a drone. In this experiment DistressNet-NG system generated a total of 4.22GB of data that were pushed into R-Drive.

Real-world deployments pose many challenges to conducting thorough tests varying several parameters which were hard to control. We also conducted several tabletop experiments to thoroughly analyze the performances with varying parameters such as link availability, data size, and EdgeKeeper configuration. We used a custom-coded application on Android devices to turn off the WiFi and LTE links to change the link availability parameter.

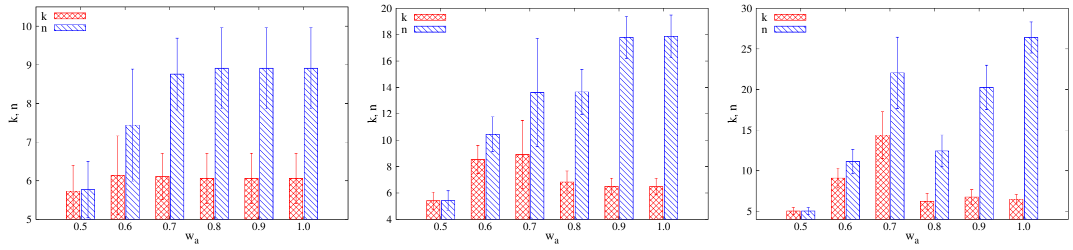
In the remaining part of this section, we present the performance evaluation for resilient data storage and sharing, edge coordination and communication in DistressNet-NG.

5.1 R-Drive Resilience through Adaptive Erasure Coding

In this section, we provide an in-depth analysis for 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 its impact on the cost function. We performed experiments on variable network sizes (10, 20, 30), for a file size F of 500MB, and an expected file availability time T of 300min. The storage S_i and expected battery remaining times T_i for nodes were generated using a pseudo-random value generator with mean-variance of $(100, 20)$ and $(300, 80)$, respectively. The experiments were conducted for 30 runs before the results were averaged.

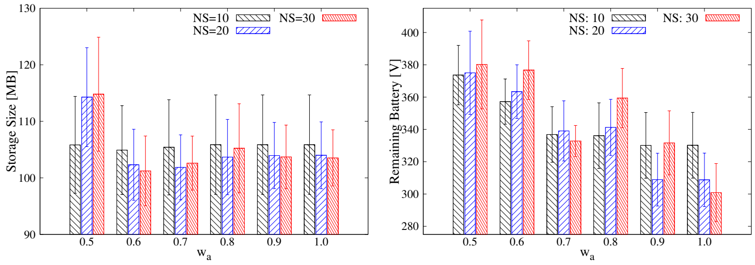
5.1.1 Achieved cost for variable w_a. Table 14a shows the average achieved cost for variable w_a and network size. For almost all w_a, the average achieved cost approaches the optimal cost with a larger network size. This is due to the fact that, with a larger network size the cost function is computed over more combinations of (n, k) values, hence, the algorithm achieves a cost value closer to an optimal value.

5.1.2 Impact of w_a and Network Size on Code Rate and F'. Figure 14b 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 sizes 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 14c shows 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 sizes 20 and 30.



(a) (b) (c)

Fig. 15. Average (k, n) for different network sizes NS: a) 10; b) 20; and c) 30



(a) (b)

Fig. 16. Impact of w_a on: a) storage size; and b) battery remaining time, for different network sizes NS=10, 20, 30

Figure 15 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 a larger n value to provide data redundancy. The cost function aims to reach the minimum cost, regardless of the choice of (n, k) values. In Figure 15c, 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 chose (n, k) values of (10,5), (12,6), (14,7) over 30 runs that averaged to (13.07, 6.53).

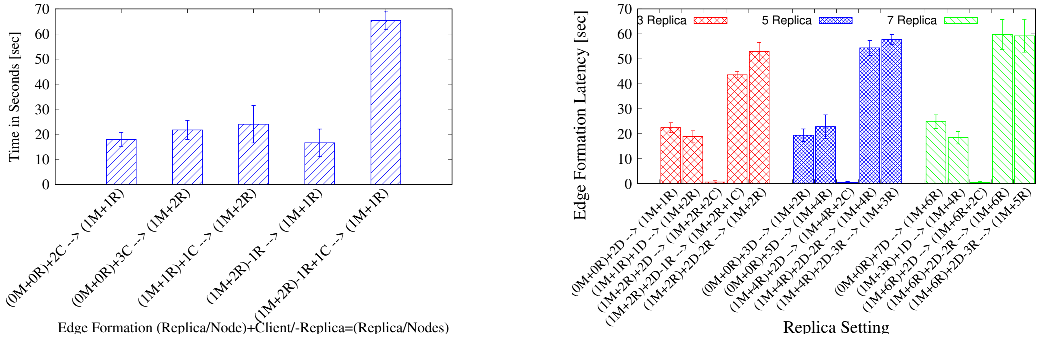
5.1.3 Impact of w_a and Network Size on selected storage and battery remaining time. Figure 16 shows the average storage capacity and battery remaining time of the selected nodes. Figures 16a and 16b illustrate the fact that on average the algorithm chooses nodes with at least minimum required storage capacity, but higher battery remaining time. This is due to the fact that, on the occasion of two different solutions having the same cost, the algorithm has provision to choose the nodes with higher device availability, as presented in Equation 7.

5.2 Directory Service Resilience

As discussed in Section 3.1, R-Drive relies on EdgeKeeper for storing file and directory metadata. This section evaluates the resiliency of directory service first in terms of its tolerance to replica faults and then latency in accessing directory metadata.

5.2.1 Edge Formation and Reformation. We measured the resilience of R-Drive's Directory Service based on how fast Directory Service becomes operational after a failure event takes place. Directory Service becomes inoperational when the replica quorum breaks for not having enough replica devices. EdgeKeeper always tries to maintain a maximum number of replica devices in the ensemble. In our experiments

The first set of experiments was conducted outdoors in the Fort Collins, CO deployment which used the VirtualNetCom Featherlite drone. The results are depicted in Figure 17a. In this experiment, the number of replicas is configured to 3. The first two bars represent age formation time, i.e., how long it takes to run a new EdgeKeeper ensemble. In the first bar,



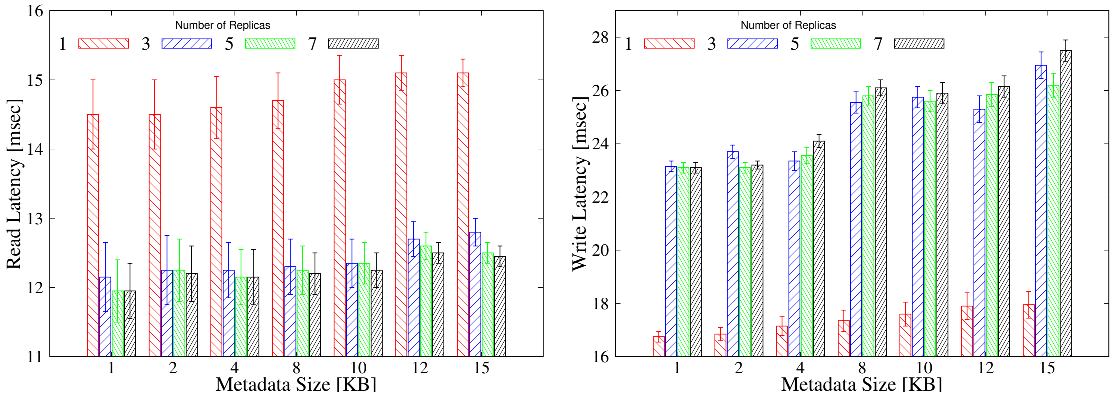
(a)

(b)

Fig. 17. Edge formation and reformation latency for variable EdgeKeeper replica settings. Here M, R, C, and D denote master, replica, client, and new device, respectively. For each x-axis tick, the equation describes the event. The term inside parentheses on the left side of the equation represents an initial condition and the remaining terms represent the changes that have been introduced. The right side represents the new network state. For example, $(1M + 1R) + 1D- > (1M + 2R)$ denotes that, in a network containing 1 master and 1 replica one new device joins and the new device start acting as a replica. We conducted experiments in two different settings: (a) Outdoor experiments conducted in Fort Collins, CO (b) controlled indoor experiments

only two devices join the network, and an EdgeKeeper ensemble is formed as soon as a quorum consisting of two devices is reached. In the case of the second bar, three devices joined together and formed an ensemble, which took longer than when two nodes only joined. The third bar represents the scenario where two replica devices are present and another device joins the network. In this case, the device initially acts as a client device. When the EdgeKeeper master detects the new device in the topology, it requests the new device to serve as a replica and restarts the already-established replica ensemble. That way, the newly formed ensemble has three replicas. We can observe that when forming a new EdgeKeeper ensemble, it takes around 20sec which is the time for a master to detect a stable node in the edge topology (topology ping interval of 10sec) and then request the new nodes to form the replica ensemble. When an EdgeKeepr ensemble is already working with $> r/2$ replicas (r is the required number of replicas), adding a new device to the network takes around 65 s to add the new device as a new replica and restart the whole replica ensemble.

To further investigate the performance, we conducted table-top experiments in a controlled lab room environment. As shown in Figure 17b, we performed experiments with EdgeKeeper replica configurations of 3, 5, and 7 to measure the average latency of edge reformation delay by introducing changes in a stable ensemble. It could be clearly observed that forming a new edge takes longer when we configure the edge ensemble with more replicas. When a new device is added to a network where the required number of replicas are already present (e.g. for 3 replicas if we add two new devices as clients), the clients immediately start their operation. However, when we add a device to an edge network where the edge ensemble is working with $r/2$ replicas, the new device would be added to the replica configuration. Restarting replicas takes significant time, such as in the 60sec in the case of the five-replica configuration. This latency contains the time period when a new device is present in the network topology and the master needs to decide whether the link is stable enough for the new device to be added as a replica. Overall, we observe that the maximum latency for restarting of replica configuration is maintained within the 70sec after a new device is added to an edge network. We note that this latency could be reduced in two ways: 1) by reducing the ping interval in the topology monitoring service; and 2) by using a lower threshold for replica down time. However, changing these thresholds needs further consideration of the amount of bandwidth consumed and the number of replicas. These two parameters should be optimized for specific network conditions (e.g., drone mobility vs. human mobility).



(a)

(b)

Fig. 18. Metadata read (a) and write (b) latencies as a function of metadata size, for link availability 1.0

It should also be noted that our current design of EdgeKeeper considers the master to be the central node of an edge network (e.g., WiFi group leader or HPC attached to deployable manpack). The master node fails when this central node fails, leading to the failure of the wireless backbone. In the future, we plan to investigate a robust master placement strategy where a master node could be automatically elected by a group of nodes. There are many leader election algorithms, such as selecting the node with the highest ID. We need to consider link quality and the duration of a node's connection for the master selection.

5.2.2 Directory Service Latency. Metadata read/write performances can vary significantly for different replica configurations and metadata sizes. Figure 18a and 18b show the average metadata read and write latencies, respectively for variable metadata sizes and EdgeKeeper replica configuration. Each result represents the average latency of 1,000 reads or writes operations. Figure 18a shows that for each metadata size group, as the number of replicas increases more than 1, metadata retrieval latency drops. This is because the read operation does not require many consensus message exchanges and more servers can perform better load balancing, resulting in overall lower retrieval latency. Variable metadata sizes have minimal effect on retrieval latency. As the range of metadata size is tiny, usually within 1 to 15KB, the average cost to fetch most metadata is almost the same. Figure 18b also shows that, as the number of replicas increases more than 1, write latency increases significantly. This is because having more replicas brings the additional message exchange to check for consensus among the replicas before the data is committed. For both read and write, adding more servers does provide additional fault tolerance but does not significantly minimize latency. Lastly, we notice that for some smaller metadata sizes, the read and write latency is higher compared to larger metadata sizes. This is due to the fact that EdgeKeeper ensemble is re-established before performing the read/write operation, adding additional time to read/write operations.

5.2.3 EdgeKeeper Overhead. To assess how lightweight EdgeKeeper is (i.e., its overhead), we employed the Android Studio profiler, which measures the resource consumption of running EdgeKeeper on mobile devices. Figure 19 depicts the memory and energy consumptions of EdgeKeeper. The results show that when running in master mode, EdgeKeeper consumes significantly more memory and network traffic than when running in replica mode. However, the energy consumption is similar for master and slave modes. In slave mode, the memory consumption by the EdgeKeeper remains intact even when the number of nodes in the cluster increases. In all cases, the memory consumption is below 40MB, which is negligible compared to 4GB internal memory of the mobile phone used. EdgeKeeper also consumes very

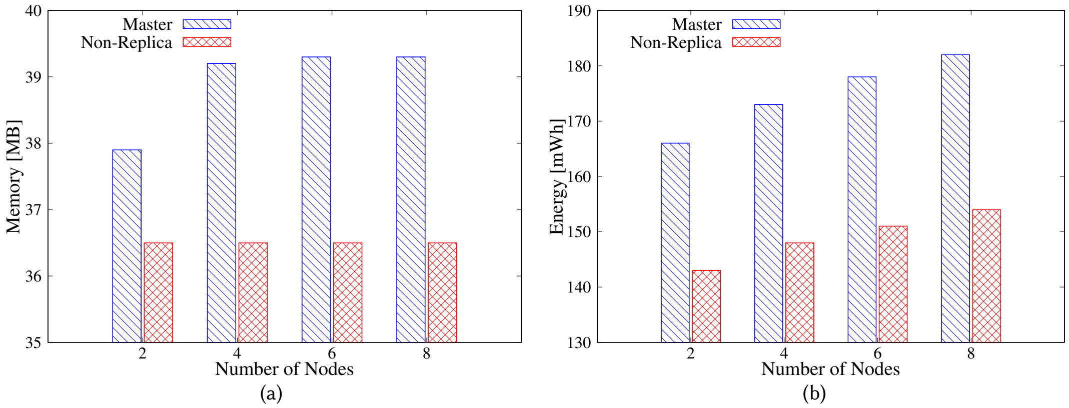


Fig. 19. EdgeKeeper overhead on an Essential PH1 mobile device, as a function of the number of nodes in the edge and the type of EdgeKeeper role: a) memory consumption; and b) energy consumption

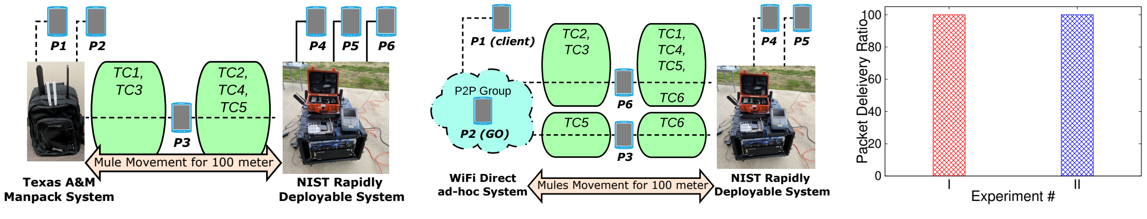
low energy (0.17Watt per hour) compared to a battery capacity of 3050mAh. These results show that EdgeKeeper is lightweight and suitable for MEC mobile devices.

5.3 Communication Resiliency with RSock

These performance evaluation results were obtained from the Disaster City, College Station TX - Wide Areas Search and Rescue deployments, using the Texas A&M Manpack and the NIST Rapidly Deployable systems along with six Android devices (labeled by P_1 to P_6), RSock initially used *Replication factor*, $rf = 1$ (defined in Section 3.2); and R-Share set its QoS parameter $TTL = 3$ hours, i.e., when it executes the registration API call (presented in Section 3.2.4).

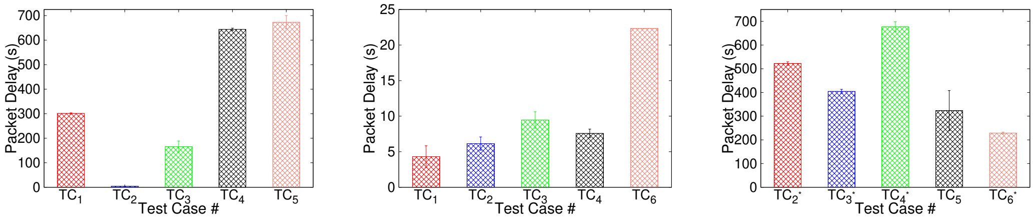
Experiment-I aims to demonstrate the interoperability of EdgeKeeper, RSock, and R-Share when they are deployed into two systems as well as the delay-tolerant routing capability (i.e., resiliency) of RSock. The two systems were physically separated by ~100m (i.e., Texas A&M Manpack was indoor and the NIST Rapidly Deployable was outdoor). We used P_4, P_5, and P_6 that were simultaneously connected via WiFi and LTE to the NIST Rapidly Deployable system and P_1, P_2, and P_3 that was connected via WiFi to Texas A&M Manpack system. This experiment has five test cases, TC_1-TC_5, as shown in Figure 20a. In TC_1, while P_3 is connected to Texas A&M Manpack system, P_1 sent a short text message to P_4, P_5, and P_6. Later, after 4min, P_3 moved to the NIST Rapidly Deployable system and stayed connected to it for ~100sec. In TC_2 (while P_3 was still connected to the NIST Rapidly Deployable system), P_4, P_5, and P_6 sent a short text message to P_1, P_2, and P_3, respectively. In TC_3, P_3 moved to the Texas A&M Manpack system and staying connected to it for 120sec. Meanwhile, P_1 sent a short text message and a 10KB file to P_4, P_5, and P_6, respectively. In TC_4 and TC_5, and after 10min, P_3 moved to the NIST Rapidly Deployable system.

The PDD for TC_1 (i.e., for the short text message via the mule) is shown in Figure 21a. Once the mule, P_3, becomes physically close to any system, it discovers its WiFi or LTE networks and automatically re-associates with it (i.e., P_3 pre-stored the WiFi passphrases and the LTE configurations of both systems). In TC_2, while P_3 is connected to the NIST Rapidly Deployable system, it instantly received the short text message that is sent from P_4, P_5, and P_6. Afterwards, P_3 carried and delivered (i.e., after 100sec) P_4, P_5, and P_6's short text messages to $P1$ and $P2$ as shown in TC_3 in Figure 21a. TC_4 and TC_5 in Figure 21a show the PDD of the short text messages and the 10KB files, respectively, that were sent from P_1 in Texas A&M Manpack system and later, after 600sec, carried and delivered to P_4, P_5, and P_6 smartphones via the mule, P_3. The PDR of all test cases of this experiment is 100%, as shown in Figure 20c.



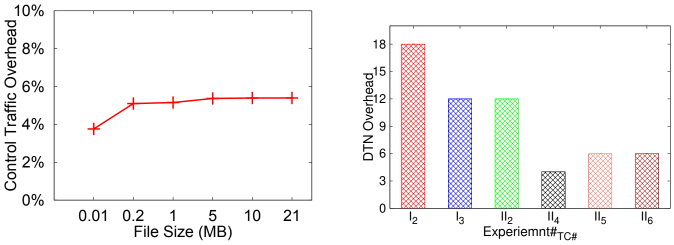
(a) (b) (c)

Fig. 20. A schematic diagram of Texas A&M Manpack system, NIST Rapidly Deployable system, and WiFi Direct ad-hoc system. Mules connectivity during: a) TC_1 - TC_5 of Experiment-I. b) TC_1 - TC_6 of Experiment-II. Solid lines represent simultaneous LTE and WiFi links. Dashed lines represent WiFi or WiFi Direct links. c) Packet Delivery Ratio of Experiment-I and Experiment-II.



(a) (b) (c)

Fig. 21. Packet Delivery Delay for all test cases for: a) Experiment-I. b & c) Experiment-II.



(a) (b)

Fig. 22. a) Control traffic overhead for all files. b) Overhead for DTN test cases of Experiment-I and Experiment-II.

Experiment-II aims to demonstrate the delay-tolerant routing capabilities (i.e., resiliency) of RSock when it runs between a *WiFi Direct ad-hoc system* and the NIST Rapidly Deployable system, as shown in Figure 20b. The WiFi Direct ad-hoc system was formed indoors by P_1 and P_2 phones. P_2 used WiFi Direct to establish a P2P group by performing the Group Owner role, and P_1 was preforming the client role by connecting via WiFi Direct to P_2. Meanwhile, P_4, P_5, and P_6 phones were connected via WiFi to the NIST Rapidly Deployable system, which was outdoor; the physical distance between the two systems was ~100m. This experiment had six test cases that are shown in Figure 20b. In TC_1, P_4 sent a 10KB file to both P_5 and P_6. In TC_2, P_4 sent a 50KB file to P_1, P_2, P_5, and P_6 phones. Later, after ~7min, the user of P_6 left the NIST Rapidly Deployable system and moved towards the WiFi Direct ad-hoc system. In TC_3, P_1 sent a 10KB file to P_2, P_4, P_5, and P_6 phones. Later, after ~6min, TC_4 was conducted while the user of P_6 moved back to the NIST Rapidly Deployable system. Afterwards, after P_6 established its connection with the NIST Rapidly Deployable system, P_4 sent a 200KB file to P_1, P_2, P_5, and P_6 phones. Next, P_3 joined as a mule and its user moved from the WiFi Direct ad-hoc system towards the NIST Rapidly Deployable system, and then back to the WiFi Direct ad-hoc system within ~10min. In TC_5, P_1 sent a 1MB file to P_4, P_5, and P_6. Next, after ~4min, P_3 user moved back to the NIST Rapidly Deployable system. In TC_6, P_6 switched it connection with the NIST Rapidly Deployable system from WiFi to LTE and sent a 1 MB file to P_1, P_2, and P_3. Lastly, after ~5min, P_3 disconnected from the NIST Rapidly Deployable system and connected to the WiFi Direct ad-hoc system.

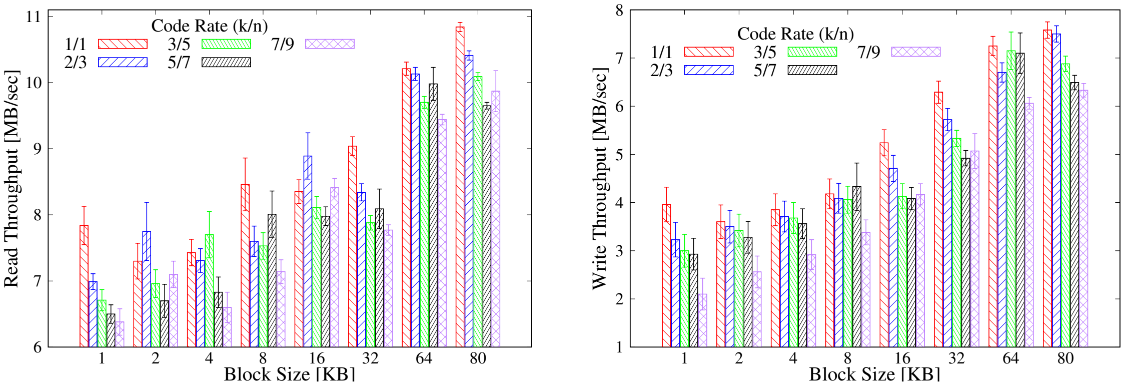
TC_1 and TC_2 in Figure 21b present the PDDs for the 10KB and 50KB files that were sent from P_4 to both P_5 and P_6 in the NIST Rapidly Deployable system. TC_{2^*} in Figure 21c shows the PDD of the 50KB files that were sent from P_4 in the NIST Rapidly Deployable system and then carried and delivered after 420sec to P_1 and P_2 in the WiFi Direct ad-hoc system. Once the mule, P_6, is connected to the WiFi Direct ad-hoc system during TC_3, it received along with P_2 a 10KB file with the PDD that is presented in TC_3 in Figure 21b from P_1. Later, after ~6min, $P6$ carried and delivered the 10KB file to P_4 and P_5 in the NIST Rapidly Deployable system, as shown in TC_{3^*} in Figure 21c. The PDD of ~5sec of the 200KB file that is sent from P_4 and delivered to P_5 and P_6 is presented in TC_4 in Figure 21b. The 200KB file is then carried via the new mule, P_3, to P_1 and P_2 in the WiFi Direct ad-hoc system, as presented in TC_{4^*} in Figure 21c. TC_{5^*} in Figure 21c shows the PDD of three 1MB files, which demonstrates RSock's capability to deliver relatively large files in delay-tolerant scenarios via the mule. TC_{6^*} in Figure 21c shows the PDD when P_6 connected to the NIST Rapidly Deployable system and sent a 1MB file via LTE to P_1 and P_2 in the WiFi Direct ad-hoc system; this file is also carried and delivered via P_3. The PDR of all test cases of this experiment is also 100%, as shown in Figure 20c.

RSock overhead. In order to evaluate the overhead of RSock, we consider the *control traffic overhead* = 1 - $(payload\ size)/(data\ packets\ size + control\ packets\ size)$. Whereas, each *data packet* consists of a header and a payload (i.e., application data), *control packets* are the imposed overhead of RSock. Similarly, we propose the *DTN overhead* = $(\#copies - \#delivered)/\#delivered$. Whereas, $\#copies$ represents the total number of transmissions (i.e., all copies of a message), $\#delivered$ represents the total number of delivered messages (i.e., only the first copy that reaches the destination is counted and the rest are discarded).

As shown in Figure 22a, RSock has a control traffic overhead that ranges from 2.8% up to 6% for 0.01-21MB files. This overhead is due to the additional header that RSock's communication API imposes to the packets' payloads to indicate the QoS parameters, to handle packets fragmentation, etc. Similarly, the overheads of DTN test cases of Experiment-I and Experiment-II are presented in Figure 22b. The DTN overhead for Experiment-I is declining from TC_2 to TC_3 after RSock in the Texas A&M Manpack's smartphones discovers the path via the mule, P_3, to the NIST Rapidly Deployable system. The same decline in the overhead also occurs from TC_2 to TC_4 and from TC_5 to TC_6 for Experiment-II with P_6 and P_3 mules, respectively, as shown in Figure 22b.

5.4 R-Drive Data Storage Throughput

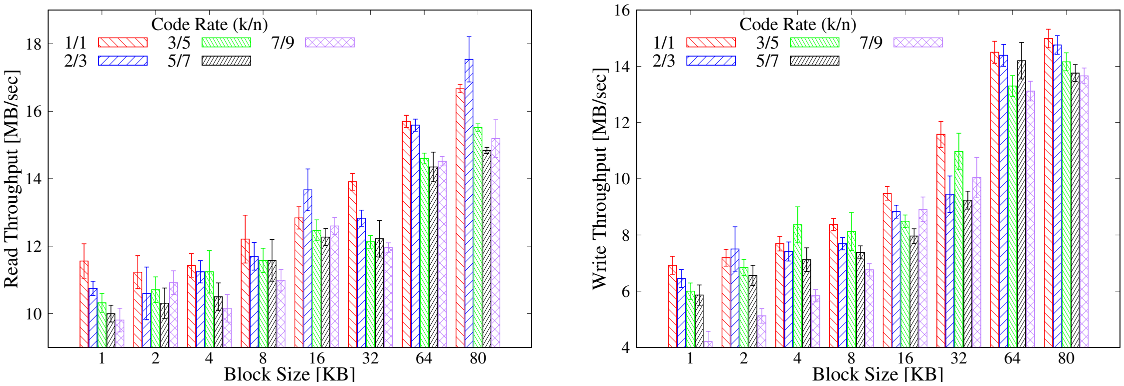
These performance evaluation results were obtained from the Texas A&M manpack and 9 Android devices (with LTE and WiFi connectivity), in an indoor lab environment. Figures 23 and 24 show the average data read and write throughput for variable code rates, block sizes, and link availability. Each phone stored and retrieved 3GB of data simultaneously, comprising file sizes ranging between 10 to 200MB. We calculated throughput by dividing the data size by the time it took for distribution or retrieval. The experiment was conducted in a purely connected network (link availability 1.0), as well as a loosely connected network (link availability 0.5). As figures suggest, read/write throughput is higher in a purely connected network compared to a loosely connected network. Also, increasing block size increases throughput for both read and write. This is due to the fact that a higher block size ensures a lower block count, resulting in a 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 due to lower code rates usually coming with higher n and k values, resulting in more fragments to be distributed or retrieved, respectively. Lastly, throughput for some larger



(a) Read

(b) Write

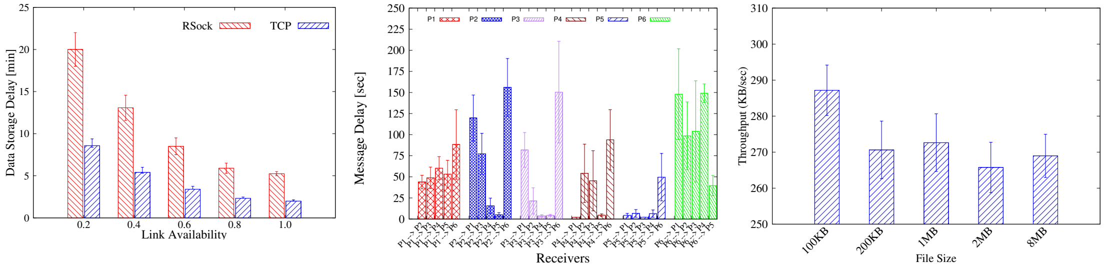
Fig. 23. Data read and write throughput as a function of block size, for 0.5 link availability



(a) Read

(b) Write

Fig. 24. Data read and write throughput as a function of block size, for 1.0 link availability



(a)

(b)

(c)

Fig. 25. (a) R-Drive data storage delay on highly sparse network connectivity compared to TCP; b) R-Share Fort Collins experiment with Drone; c) R-Share throughput for tabletop experiment for small and large file sizes.

block size were higher compared to smaller block sizes. This is due to the fact that RSock communication module makes routing decision which sometimes selects an available path that may take longer time for data delivery.

We evaluated the delay of data storage when using RSock and we compared it with a pure TCP implementation. Using 10 total devices, each device shared a file of 10MB with every other phone. Hence each device stored a total of 90MB of data in R-Drive, and all devices stored a total of 900MB of data in R-Drive. We controlled the WiFi link availability as (0.2,

Table 3. Number of allocated and de-allocated objects for different numbers of file creation and retrieval

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 4. R-Drive energy consumption for different devices

Device	**Runtime**	**Consumed**			**Dist-NG**
	h:min	**%**	**mAh**	**Wh**	**Wh**
Samsung S8	3:30	12.5	377.4	1.5	3.5
Goole Pixel 2	3:05	11.9	323.5	1.2	3.2
Essential PH1	3:15	12.6	381.8	1.5	3.8

0.4, 0.6, 0.8, 1.0) of devices using a synthetic application that periodically turned WiFi links on/off based on previously set probabilities. For example, if the link failure probability is 0.8, the device will be randomly connected to WiFi for 20% of the experiment time. Figures 25a show the performance evaluation results. As shown, RSock data-sharing delay is quite significant 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-based data-sharing delay reduces significantly.

5.5 R-Share Data Throughput and Delay

These performance evaluation results were obtained from the Fort Collins, CO deployment which used the VirtualNet-Com Fetherliste system and Android mobile devices. R-Share's message delivery delay between pairs of mobile devices is shown in Figure 25b. The variance in successful message delay for different devices is fairly high as the network connectivity was highly sparse due to drone movement along the path. Figure 25c also shows R-Share throughput in a table-top experiment using two phones. We sent approximately 100 files of different file sizes 100KB, 200KB, 500KB, 1MB, 2MB, and 8MB, and collected the time for successful reception. Throughput was calculated by dividing the total amount of data by the total time.

5.6 Framework Overhead

In this section, we present our performance evaluation results that show the overhead of the various components for DistressNet-NG resilient data storage and sharing, from the perspective of memory overhead, computation overhead, and energy consumption overhead.

5.6.1 Memory. We traced the real-time memory footprint for R-Drive using Android Profiler during the file storage and retrieval processes. The reason for it was to identify memory management problems in the R-Drive Android application, e.g., repeated memory allocation and deallocation steps. R-Drive is a write-intensive storage system, so it is important to observe whether the R-Drive application causes memory bloating issues over an extended runtime. For this experiment, we used a 20MB file and 1MB block size with (k, n) values of (10,20). Table 3 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 the number of file creations/retrieval increases but remains relatively small.

5.6.2 Processing. We measured processing time for components responsible for encryption key generation (Shamir), data encryption (AES), and data erasure coding (Reed-Solomon), as shown in Table 5. We conducted experiments for variable block sizes such as 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, and 80MB and computed the percentage of average read and write delay for each component.

We also measured the average execution time for Algorithm 1 on a Samsung S8 Android device by running the algorithm over 1000 iterations for network sizes of 30, 20, and 10 devices. The average algorithm execution times were 101.6, 15.3, and 0.5msec, respectively, indicating the relatively low execution time overhead.

Table 5. Processing overhead as a percentage of total delay

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

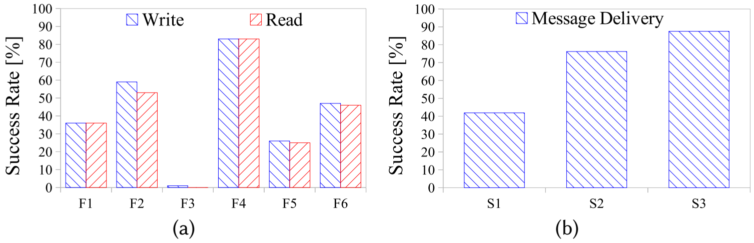


Fig. 26. DistressNet-NG performance under unstable network conditions: a) RDrive data storage write/read success rate; and b) RShare data delivery success rate

5.6.3 Energy. For this experiment, we started with 100% energy in each phone and ran DistressNet-NG application suit until the phones turned off due to battery exhaustion. We used Battery Historian [4] to collect Android battery usage data from Android devices after each experiment run. In each device, we ran EdgeKeeper, RSock, and R-Drive, along with a client application that continuously performed random file creation and retrieval in the R-Drive system. From the results, shown in Table 4 we infer that, if similar devices are used in the 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 Real-world experiments with Varying link quality

Figure 26 shows the results from the UAV-based deployments conducted in Fort Collins where we ran experiments to assess the effect of dynamic network conditions on DistressNet-NG's performance. In six different test flights (F1-F3) we measured R-Drive's read/write performance in the cluster where the success rates are below 100% ranging from close to 4% (almost disconnected network) to 83% (random disconnections). RShare also showed data delivery rates between 40% and 90% across three different test scenarios (S1-S6) of the deployment. This indicates challenges in edge formation during UAV-based deployment where the network condition is truly dynamic. Success rates varied, but a reasonable success ratio suggests that our applications are stable even in hostile network environments.

5.8 Comparison with State-of-the-Art

To the best of our knowledge, DistressNet-NG is the only end-to-end edge system that supports resilient data storage and sharing designed particularly for disconnection-prone mobile edge environments. Below we present a qualitative comparison of data storage between DistressNet-NG system with the state-of-the-art MDFS [14, 15].

- MDFS reliability estimated analytically, not from experimental data, is presented in Figure 8. MDFS shows a proportional relation between reliability and code rate, without considering other variables such as network condition or node availability. In this manuscript, we distinguish network, directory service, and data storage resilience by separate components, namely RSock, EdgeKeeper and R-Drive, and evaluate their individual contributions to reliability, using table-top and deployment experiments.

- Although MDFS attempts to choose optimal k and n values for file storage, it does not provide any guidance on selecting n candidate nodes, when considering available storage capacity or health status such as remaining battery time; we address these factors in R-Drive during node selection process.
- MDFS topology discovery is a flooding based protocol where every node floods the network with beacon message every 30sec to announce their presence, whereas, in DistressNet-NG, topology/network discovery is handled via EdgeKeeper, taking advantage of both network discovery, as well as a consensus protocol, ensuring that it does not congest the network in edge environment.
- MDFS does not maintain a distributed directory service, hence each device's view of the available files can be different, whereas EdgeKeeper maintains a distributed directory service for all node with a consistent view of the available files.

6 CONCLUSIONS

The use of MEC in CPS has been increasing in popularity, particularly in the fields of Disaster Response and Tactical applications. These applications generate a substantial amount of mission-critical and personal data that require resilient and secure storage and sharing. The presented framework is designed, implemented, and evaluated to show that it ensures resiliency in storing critical data and fast reorganization of the CPS. The framework ensures resilient communication among the nodes forming the CPS by deciding an optimal degree for replicating data that is communicated over lossy links, thereby ensuring reliable communication. R-Drive employs an adaptive erasure-coded and encrypted resilient data storage and sharing mechanism while adaptively choosing erasure-coding parameters to ensure the highest data availability with a minimal storage cost. R-Share is a secure and resilient data-sharing application for peer-to-peer communication over the opportunistic networking module RSock. The proposed framework has been successfully implemented on Android devices and integrated with existing MEC applications, demonstrating their effectiveness in enabling resilient data storage and sharing for CPS applications. The performance evaluation results show that the framework offers a reliable and efficient solution for data storage and sharing providing a high level of data availability while reducing storage costs, as well as a seamless data-sharing experience with negligible delay. For future work, we aim to investigate how to transfer fragments from one vulnerable device to a safe one over an opportunistic network before device failure takes place and data becomes lost/unavailable. Moreover, we aim to make EdgeKeeper network topology management more robust by incorporating a network service discovery protocol JmmDNS, so that multiple edge networks can advertise themselves over the same network.

REFERENCES

[1] IBR-DTN - A Modular and Lightweight Implementation of the Bundle Protocol. https://github.com/ibrdtn/ibrdtn, 2023.

[2] Alenazi, M. J. F., Cheng, Y., Zhang, D., and Sterbenz, J. P. G. Epidemic Routing Protocol Implementation in Ns-3. In *Proceedings of the 2015 Workshop on Ns-3* (2015), p. 83–90.

[3] Altaweel, A., Yang, C., Stoleru, R., Bhunia, S., Sagor, M., Maurice, M., and Blalock, R. RSock: A resilient routing protocol for mobile fog/edge networks. *Ad Hoc Networks 134* (2022).

[4] Android. Battery Historian. https://developer.android.com/topic/performance/power/setup-battery-historian, 07 2021.

[5] Apache. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/r3.4.6/zookeeperProgrammers.html, 6 2022.

[6] Apache. Storm. https://storm.apache.org/, 2023.

[7] ArcGIS. Survey123. https://survey123.arcgis.com/, 2023.

[8] Baicells. Baicells Nova 227 eNB. https://na.baicells.com/product/Details?id=c7b62a86-c748-4b71-aeb4-3f01bed0b026, 2023.

[9] Beach, B. Backblaze Reed-Solomon Erasure Coding Source Code. https://www.backblaze.com/blog/reed-solomon/, 6 2015.

[10] Berlin, F. Implementation of Optimized Link State Routing protocols for Mobile Ad-Hoc Networks. https://github.com/OLSR/olsrd, 2023.

[11] Bhunia, S., Stoleru, R., Haroon, A., Sagor, M., Altaweel, A., Chao, M., Maurice, M., and Blalock, R. EdgeKeeper: Resilient and Lightweight Coordination for Mobile Edge Clouds. In *IEEE International Conference on Mobile Ad-Hoc and Smart Systems (MASS)* (2022).

[12] Bulut, E., Geyik, S. C., and Szymanski, B. K. Utilizing correlated node mobility for efficient dtn routing. *Pervasive Mob. Comput. 13* (2014), 150–163.

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

[14] Chen, C.-A., Stoleru, R., and Xie, G. G. Energy-efficient load-balanced heterogeneous mobile cloud. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), pp. 1–9.

[15] Chen, C.-A., Won, M., Stoleru, R., and Xie, G. G. Energy-Efficient Fault-Tolerant Data Storage and Processing in Mobile Cloud. *IEEE Trans. Cloud Computing 3*, 1 (2015), 28–41.

[16] Chenji, H., Zhang, W., Stoleru, R., and Arnett, C. Distressnet: A disaster response system providing constant availability cloud-like services. *Ad Hoc Networks 11*, 8 (2013), 2440–2460.

[17] Clausen, T. H., Dearlove, C., Jacquet, P., and Herberg, U. The optimized link state routing protocol version 2. ietf rfc 7181, 2014.

[18] Dropbox. Dropbox. https://www.dropbox.com/?landing=dbv2, 2023.

[19] Dwyer, D., and Bharghavan, V. A Mobility-Aware File System for Partially Connected Operation. *ACM SIGOPS Operating Systems Review* (1997).

[20] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 29–43.

[21] Google. Google Backup and Sync. https://support.google.com/drive/answer/2374987, 2023.

[22] Google. Google Drive. https://www.google.com/drive/, 2023.

[23] Google. Google Files. https://www.google.com/drive/, 2023.

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

[25] Hui, P., Crowcroft, J., and Yoneki, E. Bubble rap: Social-based forwarding in delay-tolerant networks. *IEEE transactions on mobile computing 10*, 11 (2010), 1576–1589.

[26] Jabbar, W. A., Saad, W. K., and Ismail, M. Meqsa-olsrv2: A multicriteria-based hybrid multipath protocol for energy-efficient and qos-aware data routing in manet-wsn convergence scenarios of iot. *IEEE Access 6* (2018), 76546–76572.

[27] Kacem, I., Sait, B., Mekhilef, S., and Sabeur, N. A new routing approach for mobile ad hoc systems based on fuzzy petri nets and ant system. *IEEE Access 6* (2018), 65705–65720.

[28] Laboratory for Embedded and Networked Smart Systems (LENSS) GitHub. EdgeKeeper, RSock, R-Drive and R-Share source code. https://github.com/LENSS/EdgeKeeper, https://github.com/LENSS/RSock, https://github.com/LENSS/MDFS, https://github.com/LENSS/R-Share, 2023.

[29] Li, Z., and Shen, H. A QoS-oriented distributed routing protocol for hybrid wireless networks. *IEEE Transactions on Mobile Computing 13*, 3 (2014), 693–708.

[30] Lindgren, A., Doria, A., Davies, E., and Grasic, S. Probabilistic Routing Protocol for Intermittently Connected Networks. IRTF RFC 6693, 2012.

[31] Ltd, S. T. C. SHAREit. https://shareit.en.softonic.com, 2023.

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

[33] Microsoft. Onedrive. https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage, 2023.

[34] Otto, G. DHS sees wearables as the future for first responders. https://www.fedscoop.com/dhs-wearables-first-responders/, 2014.

[35] Paiker, N. R., Shan, J., Borcea, C., Gehani, N., Curtmola, R., and Ding, X. Design and Implementation of an Overlay File System for Cloud-Assisted Mobile Apps. *IEEE Transactions on Cloud Computing 8*, 1 (2017), 97–111.

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

[37] Rahman, A., Hassanain, E., and Hossain, M. S. Towards a Secure Mobile Edge Computing Framework for Hajj. *IEEE Access 5* (2017).

[38] Reed, I. S., and Solomon, G. Polynomial Codes Over Certain Finite Fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[39] Sagor, M., Stoleru, R., Haroon, A., Bhunia, S., Chao, M., Altaweel, A., Maurice, M., and Blalock, R. R-drive: Resilient data storage and sharing for mobile edge clouds. In *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)* (2022), pp. 171–179.

[40] Schildt, S., Morgenroth, J., Pöttner, W.-B., and Wolf, L. IBR-DTN: A Lightweight, Modular and Highly Portable Bundle Protocol Implementation. *ECEASST 37* (01 2011).

[41] Scotece, D., Paiker, N. R., Foschini, L., Bellavista, P., Ding, X., and Borcea, C. MEFS: Mobile edge file system for edge-assisted mobile apps. In *2019 IEEE 20th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)* (2019), IEEE.

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

[43] Sharma, A., Tie, X., Uppal, H., Venkataramani, A., Westbrook, D., and Yadav, A. A Global Name Service for a Highly Mobile Internetwork. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 247–258.

[44] Shu, Y., Dong, M., Ota, K., Wu, J., and Liao, S. Binary Reed-Solomon Coding Based Distributed Storage Scheme in Information-Centric Fog Networks. In *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)* (2018), IEEE, pp. 1–5.

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

[46] Xia, M., Saxena, M., Blaum, M., and Pease, D. A. A Tale of Two Erasure Codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[47] Yen, J. Y. Finding the K Shortest Loopless Paths in a Network. *Management Science 17*, 11 (1971), 712–716.

[48] Zhang, M., Bai, Y., Yuan, S., Tian, N., and Wang, J. Design and Implementation of File Multi-Cloud Storage System Based on Android. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)* (2020).

[49] Zhu, R., Niu, D., and Li, Z. Online Code Rate Adaptation in Cloud Storage Systems with Multiple Erasure Codes. In *28th Biennial Symposium on Communications (BSC 2016)* (2016).