EdgeKeeper: Resilient and Lightweight Coordination for Mobile Edge Clouds

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

***Abstract*—Mobile Edge Clouds (MEC) have been gaining significant interest from first responders and tactical teams, primarily because they can employ handheld mobile devices to form a computing cluster (for high-computation tasks such as face/scene recognition, and virtual assistance) when connectivity to the cloud is limited or not possible. High user mobility in disaster response and tactical environments make MEC challenging, as wireless links observe substantial fluctuations. Typical cloud-based coordination (e.g., ZooKeeper-based service discovery and coordination, device naming, security) cannot typically work in these environments. Driven by the need for a resilient and lightweight coordination service, in this paper, we design and implement EdgeKeeper to provide cloud-like coordination for MEC systems. EdgeKeeper provides naming, network management, application coordination, and security to distributed edge computing applications. It maintains an edge cluster among devices and intelligently stores its data on a group of replicas to guard against node failure and disconnections. We provide a full-system implementation of EdgeKeeper for Android and Linux platforms. We have integrated EdgeKeeper with existing MEC applications and performed extensive real-world performance evaluations in wide-area search and rescue operations conducted by first responders. Our evaluation shows EdgeKeeper to be lightweight, resilient and suitable for MEC.**

I. Introduction

Over the past decade, advancements in handheld devices coupled with data rate enhancements in radio access technologies have led to an exponential increase in the number of mobile applications [1], [2]. New applications that generate large amounts of multimedia data (e.g., videos, images, and audio), are gaining significant popularity among disaster response and tactical/military teams. Example use case is automatic human/scene identification/recognition from video captured through body-mounted cameras or voice assistance [3]. However, these applications require significant resources for data processing. Traditionally, processing-intensive big data generated by mobile applications are offloaded to remote cloud servers for processing. In the absence of connectivity to the cloud, such as in disaster response and tactical environments, enabling data processing on mobile devices at the edge emerges as a necessity [4], [5].

Mobile Edge Cloud (MEC) enables a paradigm shift in data processing, where jobs are offloaded to the nearest devices instead of sending them to remote cloud servers [6]–[9]. Instead of treating mobile phones as thin clients, higher computing power allows us to view them as thick clients or effectively thin servers. The MEC scenario, shown in Figure 1, becomes more prominent when a group of mobile nodes loses their connection to the Internet, and thus to the cloud. In a first response scenario a team of first responders is equipped with on-helmet cameras, mobile devices (e.g., “M-1” through “M-5” in Figure 1), and a manpack (i.e., “HPC-1” node in Figure 1). The manpack and mobile devices form a mobile edge that can be leveraged for sharing computation resources for victim face detection, face recognition or voice assistance. In the absence of network infrastructure, the manpack provides LTE and WiFi connectivity. Similar to the Apache Hadoop [10] ecosystem, we have developed the DistressNet-NG [11] ecosystem, which is particularly targeting edge networks formed by handheld devices and deployable manpacks carried by first responders.

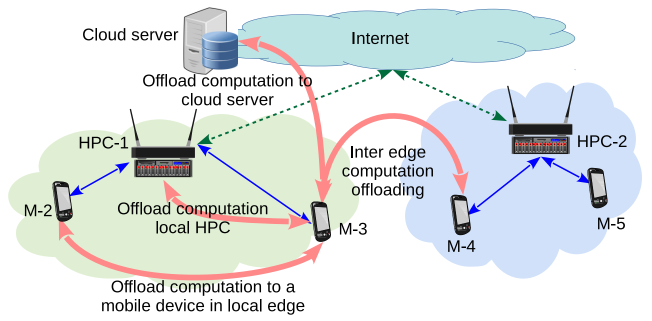

Cloud server
Internet
Offload computation to cloud server
HPC-1
HPC-2
Inter edge computation offloading
M-2
Offload computation local HPC
M-3
M-4
M-5
Offload computation to a mobile device in local edge

Fig. 1: A MEC scenario where two edges are formed. Computation offloading can occur intra-edge and inter-edge.

Edge computing applications, such as mobile stream processing (e.g., MStorm [5]), require a naming and coordination service to schedule the execution of tasks. Other edge applications such as R-Drive [12] require resilient storage for critical metadata. In the distributed cloud computing framework, Apache ZooKeeper is widely used for service coordination. However, it requires statically assigned coordination servers/replicas and fails to provide service as soon as the majority of the replicas get disconnected. Thus, ZooKeeper in its current state is not suitable for distributed edge computing. To fill this gap, we have developed EdgeKeeper, a resilient, distributed coordination service for mobile edge clouds. It is implemented as an application that runs in the background on all edge devices and provides resilient coordination for other applications, e.g., device naming, application coordination, edge status monitoring, and authentication. In particular, we

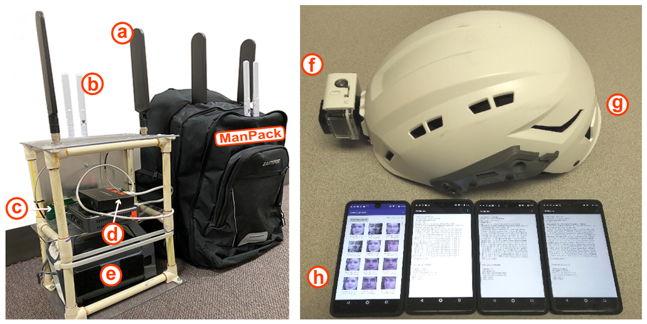


Fig. 2: DistressNet-NG hardware components: a) LTE antenna, b) WiFi AP, c) LTE eNB, d) Intel NUC that runs LTE EPC and HPC, e) Battery, f) Body camera, g) Helmet of first responder, h) Handheld Android phones.

make the following contributions:

- We present the design of EdgeKeeper, a distributed coordination service for mobile edge clouds. EdgeKeeper is designed to ensure the resilience of coordination in edge networks while hiding the complexity of edge coordination from applications.
- EdgeKeeper provides a comprehensive application programming interface (API) for client applications which includes device naming, application coordination, metadata storage, authentication for nodes and users, and edge status monitoring.
- We provide an open-source implementation of EdgeKeeper for both Linux and Android platforms. We have integrated an EdgeKeeper client library with applications such as MStorm, R-Drive, and RSock.
- We present results from real-world deployments with first responders, which show that EdgeKeeper is lightweight and suitable for MEC scenarios.

The rest of this paper is structured as follows. Section II provides the background and motivation of the current work. The design and implementation of EdgeKeeper are presented in Sections III and IV, respectively. In Section V we evaluate the performance of EdgeKeeper. Finally, Section VI concludes our paper.

II. Background and Motivation

In this section, we first present background material on DistressNet-NG, an edge computing ecosystem designed for disaster response teams. Then, we motivate the need for resilient application coordination (similar to what is needed in the cloud). In the traditional cloud computing paradigm, a resource-constrained mobile device offloads computation tasks to a remote cloud server, but it fails to perform the required computation when connectivity to the cloud is lost. In DistressNet-NG, multiple mobile nodes form an edge network (as in Figure 1) where devices can offload tasks to nearby mobile devices as well as the cloud server once it is available.

A. DistressNet-NG Hardware Architecture

A group of first responders carries out its search and rescue mission in a disaster response scenario, assisted by handheld devices, on-body cameras, and other sensors. As the cellular wireless infrastructure is usually unavailable, these teams carry a deployable wireless communication system, which typically consists of the following: 1) mobile devices equipped with LTE and WiFi wireless capabilities, 2) LTE eNodeB (i.e., an LTE access point), 3) WiFi access point(s), 4) High-performance Computing (HPC) device(s). The DistressNet-NG hardware is shown in Figure 2. A lightweight manpack comprises of a Baicells eNodeB and a Ubiquiti WiFi access point that provide LTE and WiFi connectivity to mobile devices. LTE EPC functionalities are managed by NextEPC [13], an open-source LTE core running on an Intel NUC.

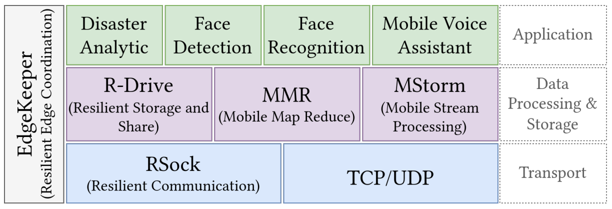


Fig. 3: DistressNet-NG edge computing software ecosystem

B. DistressNet-NG Software architecture

The Apache big data ecosystem has become synonymous with big data processing in the cloud. However, it is not designed to run on mobile devices and performs poorly if servers are mobile. Hadoop, in the Apache ecosystem, splits a submitted job into several small tasks and executes these tasks in parallel, on multiple servers, reducing the delay associated with a sequential execution. The Hadoop Distributed File System (HDFS) replicates the data according to its demand and reliability requirements. Apache also offers MapReduce - a parallel processing framework, and Storm - a distributed processing framework for big data processing in the cloud. To provide coordination to all components of the Apache ecosystem the coordination service ZooKeeper [14] is employed.

To enable these services in MEC, we have developed similar components for mobile devices. Figure 3 provides an overview of the DistressNet-NG software ecosystem. We have developed a big data processing system Mobile MapReduce (MMR) and a real-time stream processing system Mobile Storm (MStorm) which run on mobile devices and provide the same functionality as MapReduce and Storm [5]. To tackle device mobility and frequent disconnections, we have developed Resilient Sockets (RSock) [15], which abstracts data delivery for applications in wireless networks with diverse connectivity. We have also developed R-Drive [12] which stores large amounts of data on mobile devices in a resilient manner.

C. Motivation for Edge Coordination

All DistressNet-NG applications need reliable and resilient coordination. Unfortunately, conventional coordination services such as ZooKeeper fail to operate in mobile edge environments because of frequent node and link failures. If the ZooKeeper ensemble (in which server IPs are statically assigned) loses the majority of its server nodes, the entire ensemble fails to work. When some nodes leave the edge network, new nodes should be dynamically chosen to participate in consensus and reconfigure the edge network. This allows MEC applications to continue their operations. During our development of MEC applications, we have identified the need for the following services:

1) Device Naming

Most applications, such as MMR, rely on the Domain Naming Service (DNS) to abstract the physical IP address for communication with a target device. The conventional hierarchical DNS-based naming service fails to handle intermittent network disconnections and high node mobility.

2) End User and Device Authentication and Authorization

A distributed computing framework requires devices or end-users to be authenticated and authorized before receiving services. Cloud-based services (e.g., Kerberos) fail to work when disconnected from the authorization entity. MEC needs a framework where devices can be authenticated and authorized by an edge network autonomously in both connected and disconnected scenarios.

3) Node and Service Discovery and Coordination

In addition to the naming service, distributed applications also require a coordination service, which provides service provider discovery, data synchronization, group configuration, leader election, status monitoring, critical section synchronization, queuing, etc. The service discovery should provide a global view of the available servers for a service, and when possible, provide a subset of nearby servers.

4) Resilient Metadata Storage

Some applications such as a Distributed File System (DFS) must store file metadata on resilient storage. In Apache HDFS, the file metadata (information about where file fragments are stored) is stored on a single Master device - the Name Node. In the edge network, this metadata must be stored over multiple devices, guarding against device failure and disconnections.

5) Edge Status Monitoring

MEC client applications require knowledge about the edge network status (e.g., wireless link qualities between peer devices, device battery status, and device processing load) in order to make intelligent decisions for computation or data offloading. Typically these applications assess the edge network individually, resulting in congestion. To reduce this overhead, there is a need for a single service in an edge network to provide a comprehensive view of the edge to client applications. Network middleboxes such as firewalls [16] used in most standard deployable networks disrupt the conventional ad hoc link maintenance and routing protocols. Thus, there is a need for an edge status monitoring service that can work with these networks.

III. EdgeKeeper Design and API

In this section we present the design of EdgeKeeper, a resilient and lightweight coordination service for MEC, and how it addresses the needs of MEC applications that were identified in the previous section. EdgeKeeper runs as a background process on all edge devices. Instead of running one EdgeKeeper on a central device and storing all data at a single node, data is replicated over multiple devices to tackle node and link failures. The devices that store the data with consistency are called EdgeKeeper replicas and provide EdgeKeeper functionality to the slaves (non-replica EdgeKeeper devices, or clients). The terms non-replica EdgeKeeper devices, slaves and clients will be used inter-changeably. The roles of replica and slave are chosen dynamically, depending on the network status.

A. Device Naming

EdgeKeeper provides resilient device naming for edge networks. When connected to the Internet, it provides a coherent name resolution service at a global scale. EdgeKeeper employs the Global Naming Service (GNS) [17], which utilizes multiple name servers to handle high name resolution rates across the globe. Each name record is associated with a primary key that is a globally unique identifier (GUID). However, if a GNS server gets disconnected from the federated group of GNS servers, it fails to provide services. To address this, EdgeKeeper also employs a local cache mechanism to store the name records at the edge. The name record updates are committed to the local cache and lazily updated to the GNS server whenever connection to the Internet is restored. The cache is maintained by the EdgeKeeper-replicas that run consensus amongst themselves for consistency.

EdgeKeeper also provides the conventional DNS name-to-IP translation. As EdgeKeeper-master runs the DHCP for the LTE and WiFi network, in the typical cases, the DNS translator also resides there. Upon receiving a DNS query, the translator tries to resolve the name using two methods: 1) checking with the local GUID record in the cluster for IP translation and 2) forwarding the query to one of the GNS servers. The DNS server will return the result from the first completed query.

B. Authentication and Authorization

EdgeKeeper uses X509 certificates for authentication [18], thus each device maintains a public-private key pair. A GUID is a **self-certifying identifier** as it is derived by a one-way hash function (known universally) from a user's public-key. A bilateral nonce-based challenge-response is used to authenticate if a node claims to be the GUID owner. In our proposed architecture, a certifying authority (CA) at each organization creates client certificates and signs them using the CA's private key. The CA's public certificate (.pem) is stored at the TrustStore of the federated GNS servers. When a new node tries to create a GNS account, it provides the signed client certificate. Since the GNS server already contains the public certificate of the CA, it can verify the client. Multiple CAs can be imported to the trust store of GNS servers and thus, multiple entities can have their own CA and provide them to GNS.

The client credentials (public-private key pair and a signed certificate from a CA) are stored in a .p12 file which contains: 1) a public certificate containing user credentials (identity/name, organization, etc.), the public key and the digital signature from a CA; 2) the public key and the certificate of the CA; and 3) the private key corresponding to the public key of the user. The .p12 file is password protected making the private key secure from any unauthorized access.

C. Node and Service Discovery and Coordination

EdgeKeeper uses GUID records for service discovery. If a device offers a service and wants it to be discovered by other nodes in the network, the service name and the role (e.g, mstorm:master) are part of the GUID record. Each GUID record contains an associative array of key-value pairs. Any node that wishes to find a list of nodes offering a particular service will query EdgeKeeper to retrieve a list of GUIDs, which contains the key-value pair as `service: role`.

D. Resilient MetaData Storage

EdgeKeeper also provides resilient metadata storage to client applications. To eliminate the problem of a single point of failure, it uses a ZooKeeper-like consensus. In our design, all replica nodes participate in the consensus. When the cluster cannot reach consensus due to a majority of server failures, new nodes are chosen dynamically to be replicas. Maintaining consensus over wireless links is costly as it involves a large number of message exchanges between replicas. An intelligent decision should be made to choose the number of replicas.

E. Edge Status Monitoring

At the edge, applications need information about the network topology and devices' statuses. EdgeKeeper provides applications this information, through the following services:

1) Network Topology Discovery

EdgeKeeper runs a topology discovery service where each device periodically pings other devices in the network, thus determining the device-to-device link quality. EdgeKeeper maintains a network graph which contains available nodes and the links among the nodes. Each node maintains the link qualities to its immediate neighbors when there are multiple wireless links possible between two devices (e.g., WiFi and LTE), EdgeKeeper maintains information about all links. Periodically each node calculates the optimal distance from itself to all destinations and broadcasts this distance vector to its neighbors. Thereby, the whole network is seen as a two-hop network from all nodes. Each device periodically sends a UDP packet containing its distance vector to individual neighbors using unicast IP addresses. Measuring the bandwidth over the wireless link is difficult because it requires periodically probing a link (thus impacting all other traffic). For link quality, EdgeKeeper measures the round-trip-time (RTT) and the expected number of transmissions required for a packet to be successfully transmitted and acknowledged (ETx). For each link, EdgeKeeper stores an average value of the measured RTTs using an exponential moving average. The ETx of a link between nodes A and B is calculated as follows: $ETx_{AB} = \frac{1}{(1-PDR_{A\rightarrow B})(1-PDR_{B\rightarrow A})}$, where $PDR_{A\rightarrow B}$ is the packet loss the rate from A to B and $PDR_{B\rightarrow A}$ is the packet loss the rate from B to A.

A node inside a deployable edge network can also reach a node residing in the cloud as the cloud nodes have public IPs. However, cloud nodes can not initiate connections to nodes sitting behind middleboxes (NATs, firewalls, etc.). However, if a TCP connection is initiated from an edge network to the cloud server, the cloud server can reply to the edge network through the established session. The EdgeKeeper-master assesses the link quality of the link between itself and the cloud server using periodic pings through TCP sessions. All other nodes in the edge network add this link to their topology graph.

2) Device and Application Health monitoring

As edge applications require the status of peer devices (e.g., number of functioning processors, available memory, remaining battery, available storage, etc.), EdgeKeeper makes health monitoring data available to all nodes in the edge network. The EdgeKeeper running on each device periodically measures the device status and reports it to the local EdgeKeeper-master. Any edge application can obtain a device's status through its resident EdgeKeeper, which queries it from the EdgeKeeper-master. Applications can also report application-specific statuses, such as queue length and processing latency.

IV. EdgeKeeper Implementation

In this section we present the implementation of EdgeKeeper in detail, starting from how an edge network is formed, how EdgeKeeper handles node departures/failures, to edge partitioning and merging. As EdgeKeeper runs on all edge devices, it is implemented as a daemon process in Linux and as a background service in Android. Client applications such as MStorm, RSock, MMR, and R-Drive interact with the EdgeKeeper service running on the local device. The client applications use a Java client library that implements the EdgeKeeper API (i.e., the communication with the EdgeKeeper process through JSON-based RPC over a local TCP socket).

A. Node Discovery with EdgeKeeper

To form an EdgeKeeper cluster, multiple nodes need to discover each other. The node and service discovery was presented in Section III-C. Discovering neighbors in a LAN is a well-studied topic (e.g. ZeroConf LLMNR [19], Multicast DNS (mDNS) [20], DNS-SD [21], etc.). Existing service discovery protocols like Avahi [22], Bonjour [23], and NSD (Android) employ DNS-SD (DNS Service Discovery) or mDNS (multicast DNS) for service discovery and registration. The implementation of DNS-SD and mDNS require support for multicast/broadcast, capability that is not available over LTE wireless links. To tackle this issue, we have developed a lightweight solution through which a new node can find other nodes in the network.

We designate a node as a gateway and call it EdgeKeeper-master to which all new nodes send a first message indicating that they widh to join it. After joining a wireless network using WiFi or LTE, a new node obtains its IP address from a DHCP server. The DHCP reply contains the master's IP in the DNS field (the EdgeKeeper-master runs the DNS server). Using a special hostname (`master.distressnet.org`) the new node then finds the IP address of the master as a first step. The second step in this process is to start the Network Topology Discovery. The new node sends a topology ping message to the master. The ping message contains the new node's GUID, IPs, and a sequence number. Upon receiving the ping message, the master replies to the ping with its GUID and IPs. The master also adds this device to its network topology graph.

For the new node's EdgeKeeper to access the GUID records maintained by the EdgeKeeper replicas, it: a) sends a request to the EdgeKeeper-master for the edge status; and b) it receives the current status of the replicas with their IP addresses. If the status confirms that the minimum number of replicas are present, the new node connects to the replicas. If the status indicates that the quorum (minimum number of replicas) is not met, the new node waits and fetches the status repeatedly. Also, if the master decides to use the node as one of the replicas, it sends a request to join as one of the replicas.

B. *EdgeKeeper Replica Selection - Edge (Re)Formation*

The EdgeKeeper-master uses a parameter r read from a configuration file for the number of replicas. For an edge to be formed there must be at least $r/2+1$ nodes. If r is 1, the master chooses itself as the only replica. For multiple replicas, the master periodically checks the network topology for the presence of other nodes, as presented in the previous section. From the topology, it selects its one-hop neighbors to be replicas. Since the replicas run consensus, they must be reachable by each other without any middle boxes. For a replication factor of r, the master always tries to select r nodes for replicas. If there are less than r nodes available, then it tries to assign the maximum possible suitable nodes to serve as replicas. If less than $r/2$ nodes are selected as replicas, the edge status is updated to the *Looking* state. If more than $r/2$ nodes are serving as replicas, the state is updated to *Formed*.

C. *EdgeKeeper Resiliency to Replica Failure*

EdgeKeeper is resilient to nodes leaving the edge network (either by network or device failure). If the departing node is serving as one of the replicas, a new node needs to be selected to serve as a replica. The EdgeKeeper-master periodically checks the network topology graph for any disconnected replicas, and assigns a new node to serve in its place. The master also informs all the replicas about the change. Note that, if the majority of replicas leave an edge network together, then the quorum cannot be achieved and an edge cannot be formed.

D. *EdgeKeeper in Ad-Hoc Networks - WiFi-Direct*

EdgeKeeper is designed to work in both infrastructure-mode (LTE and WiFi) and Ad-Hoc mode. If a group of nodes gets disconnected from the HPC node (WiFi/LTE), then those nodes can form an Ad-Hoc edge network. A user initiates the WiFi-Direct on one of the nodes as the group owner. EdgeKeeper on this device will operate as the EdgeKeeper-master. Once this is configured, other nodes can join the edge network as described in Section IV-A.

(a) (b) (c)

Fig. 4: Real-world deployments: a) Gypsum, CO, with a rapidly deployable as shown; b-c) Disaster City, TX, with the manpack shown

E. *Edge Partitioning and Merging*

When two edge networks are within communication range of each other (either wireless or wired), information can be exchanged between them. Although they remain separate entities/edges, they "merge" from applications' perspectives. The EdgeKeeper-masters in the two edges share their GUID data and metadata information with each other. Upon receiving this information from a neighboring EdgeKeeper-master, the local EdgeKeeper-master pushes this data to its local replicas. Thus, applications running on the two edges discover each other through service discovery.

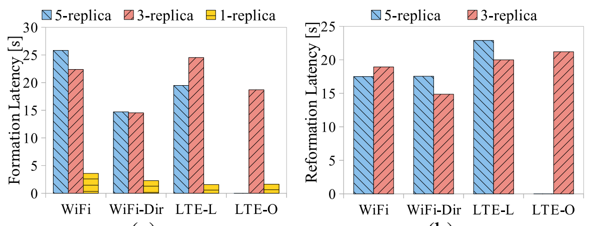
F. *EdgeKeeper Integration with the Cloud*

Client applications such as MStorm can utilize high-performance servers available in the cloud. EdgeKeeper runs on the cloud servers as a separate cluster. An EdgeKeeper-master present in one edge network monitors connectivity to the cloud. If the connectivity to the cloud is present, the EdgeKeeper-master initiates a TCP connection to the EdgeKeeper in the cloud and it monitors the connection quality through periodic ICMP messages. Client applications rely on this cloud-connectivity quality to determine whether or not to offload computation to the cloud.

V. Performance Evaluation

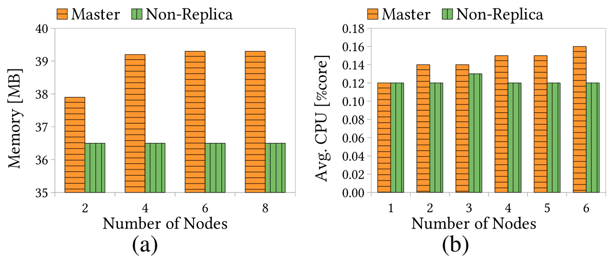
To investigate the resiliency and overhead of EdgeKeeper and its suitability for MEC, we employed the following performance evaluation metrics: edge formation latency, edge reformation latency (when a replica node joins or leave an edge), EdgeKeeper overhead (CPU and memory), and API performance for two applications, namely R-Drive and MStorm. The parameters that we varied were the type of wireless edge connectivity (WiFi, WiFi-Direct, LTE-Lab, and LTE-Outdoor), the number of EdgeKeeper replicas (used for resiliency), read/write latency to EdgeKeeper metadata, and API request latency from applications.

Due to the large number of parameters and performance metrics, we evaluated EdgeKeeper in both a laboratory environment and through three real-world deployments using various types of MEC hardware. Images from our deployment areas and hardware are presented in Figure 4. Our *first deployment* was carried out in 2018 in an open mountainous region near Gypsum, CO (Figure 4a), where the carried devices formed an edge cluster through LTE network. These experiments helped us understand the impact of physical distance on EdgeKeeper API latency. The *second deployment*, depicted in Figure 4b was done in 2020, as part of a disaster response



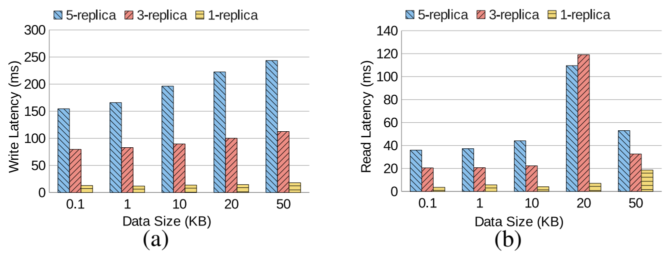
(a) (b)

Fig. 5: Edge formation latency, when: a) at least half of required replicas are present; and b) a new node joins a partially formed edge and it is selected as a replica. The 5-replica edge formation and reformation experiments were not conducted for the LTE-Outdoor experiments, due to time constraints - duration of the flight and set of other experiments needed)



(a) (b)

Fig. 6: 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; b) average CPU utilization



(a) (b)

Fig. 7: EdgeKeeper metadata write and read latency (in a lab environment), as a function of the data size and number of replicas, when: a) a replica node performs a write; b) a non-replica node performs a write; c) a replica node performs a read

exercise (i.e., wide area search and rescue) in Disaster City, TX. The disaster response team consisted of four responders, with one of them carrying a manpack shown in Figure 4c, while the others were equipped with helmet-mounted cameras. All responders carried Android phones that were connected to the manpack using both LTE and WiFi. The MEC application deployed was a Face Recognition app employing MStorm. The *third deployment* [24] was carried out in 2021 in the Christman Airfield in Fort Collins, CO, where the wireless connectivity in the MEC was provided by a drone with a payload of an LTE-in-a-box by Featherlite [25]. From this experiment, we were able to assess the feasibility of EdgeKeeper for highly dynamic environments employing only LTE.

The performance evaluation of EdgeKeeper in the aforementioned environments/deployments is presented in the sections that follow.

A. Edge Formation Latency

In this set of experiments we evaluated our design decisions and implementations mentioned in Sections IV-A and IV-B. We measured the edge formation latency as a function of wireless connectivity (WiFi, WiFi-Direct, LTE-Lab, and LTE-Outdoor) and the degree of resiliency, i.e., the number of EdgeKeeper replicas (r as mentioned before). Figure 5a plots the results of these experiments. As shown, in all scenarios for a 1-replica configuration, EdgeKeeper forms an edge within 2-3s. While this edge formation latency is rather short, it offers the least degree of fault tolerance. When EdgeKeeper is configured to use multiple replicas, the edge formation latency becomes significant. In these cases, the EdgeKeeper-master needs to discover other nodes in the network and to select suitable replicas. As the service discovery employs ping messages sent at 10s intervals, the edge formation latency observed was between 15-25s. In the WiFi and LTE network scenarios, nodes discover the EdgeKeeper-master through DNS resolution which exhibits a higher delay than WiFi-Direct networks. In a WiFi Direct network, nodes treat the group owner as the EdgeKeeper-master and directly join the edge, without going through DNS. In most cases, we can see that an edge cluster forms within 25s, even when EdgeKeeper is deployed in a highly dynamic environment, such as LTE-Outdoor (i.e., third deployment, when the MEC was formed through LTE on a drone).

We have also evaluated edge reformation latency, i.e., how long it takes to add one new node as a replica to an already established EdgeKeeper cluster where there is a shortage of replica nodes (i.e., there are between $r/1+1$ and r replicas only). For the case $r = 3$, EdgeKeeper forms a MEC and serves clients as long as there are 2-replicas. When another node joins, the master includes it in the replica pool aiming for the configured r=3 replicas. Figure 5b shows the edge reformation result after a node joins an existing edge and EdgeKeeper reforms the edge. In most cases, a successful edge reformation took no more than 25s on average.

B. EdgeKeeper Overhead

To assess how lightweight EdgeKeeper is (i.e., its overhead), we employed an Android Studio profiler which measures the resource consumption of running EdgeKeeper on mobile devices. Figure 6 depicts the memory consumption and processor utilization of EdgeKeeper. The results show that when running in master mode, EdgeKeeper consumes significantly more memory than when running in replica mode. However, the processor usage 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. Processor usage is within 0.16%. These results show that EdgeKeeper is lightweight and suitable for MEC mobile devices.

C. EdgeKeeper API - R-Drive Performance

In this set of experiments we investigated the suitability of EdgeKeeper for a MEC application, R-Drive. R-Drive requires resilient metadata storage, which EdgeKeeper provides as

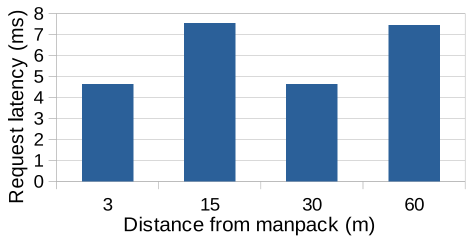


Fig. 8: EdgeKeeper service discovery API latency for 1-replica EdgeKeeper cluster in an outdoor deployment

presented in Section III-D. In a lab environment (using WiFi), we measured EdgeKeeper metadata write and read latencies, when requests are made directly to an EdgeKeeper replica node or a non-replica node, as a function of metadata size and the degree of fault-tolerance (i.e., parameter r). The results are shown in Figure 7. We observe that the metadata write latency is significantly higher when the request is made by a non-replica node, as the request needs to be sent over the network to replicas and thus, it involves network traffic. The results also show that the read latency is lower than the write latency as writing data requires a consensus to be achieved among replica nodes. The results also show that the latency increases with an increase in the number of replicas. This is expected, as the consensus protocol for a large number of replicas requires a higher number of message exchanges.

D. EdgeKeeper API - MStorm Performance

In this set of experiments we investigated the impact of wireless link quality on EdgeKeeper's API performance for a MEC application that employs EdgeKeeper for Service Discovery, MStorm. As mentioned, MStorm is a real-time stream processing framework for distributed edge computing. MStorm employs EdgeKeeper for discovering devices when establishing an MStorm edge computing cluster. Results obtained from the first deployment (i.e., Figure 4a) are shown in Figure 8. Remarkably, the EdgeKeeper API invocation delays for Service Discovery are within 8ms. This is due to the fact the request is served from a local EdgeKeeper cache (i.e., the node and service discovery are completed well before MStorm requests this information from EdgeKeeper). The observed slight variance in the request latency can be explained by the varying processing load of the device at that time.

VI. Conclusions

In this paper, we present EdgeKeeper, a resilient and lightweight coordination service for mobile edge clouds. We designed EdgeKeeper to provide services including device naming, application discovery, service coordination, metadata storage, and edge status monitoring to MEC client applications that may be disconnected from the Internet. We have implemented EdgeKeeper on both Android and Linux platforms and evaluated its performance in both lab and real-world deployments. The performance results show that EdgeKeeper fulfills all requirements for edge coordination in MEC. EdgeKeeper is lightweight and resilient to node and link failures and it is able to reconfigure the MEC seamlessly with short latencies. Future work for EdgeKeeper includes new capabilities such as edge resource orchestration.

References

[1] H. Chenji, *A Fog Computing Infrastructure for Disaster Response*. PhD thesis, Texas A&M University, 2014.

[2] M. R. Rahimi, J. Ren, C. H. Liu, A. V. Vasilakos, and N. Venkatasubramanian, "Mobile Cloud Computing: A survey, state of art and future directions," *Mobile Networks and Applications*, vol. 19, no. 2, 2014.

[3] K. Usbeck *et al.*, "Improving situation awareness with the Android Team Awareness Kit (ATAK)," in *Proceedings of Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security, Defense, and Law Enforcement XIV*, International Society for Optics and Photonics, 2015.

[4] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 2795–2808, 2016.

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

[6] J. George, C. Chen, R. Stoleru, G. G. Xie, T. Sookoor, and D. Bruno, "Hadoop MapReduce for tactical clouds," in *3rd IEEE International Conference on Cloud Networking (CloudNet)*, 2014.

[7] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[8] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: architecture and service modes," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 18–24, 2015.

[9] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: from concept to practice and beyond," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80–88, 2015.

[10] Apache, "Hadoop." [last accessed July 20, 2022].

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

[12] M. Sagor, R. Stoleru, A. Haroon, S. Bhunia, M. Chao, A. Altaweel, M. Maurice, and R. Blalock, "R-Drive: Resilient data storage and sharing for mobile edge clouds," in *19th IEEE International Conference on Mobile AdHoc and Smart Systems (MASS)*, 2022.

[13] NextEPC Inc., "NextEPC." [last accessed July 20, 2022].

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, 2010.

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

[16] T. Lukovszki, M. Rost, and S. Schmid, "It's a match! near-optimal and incremental middlebox deployment," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 30–36, 2016.

[17] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, "A global name service for a highly mobile internetwork," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 247–258, ACM, 2014.

[18] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," 2002.

[19] B. Aboba, D. Thaler, and L. Esibov, "Link-local multicast name resolution (LLMNR)," tech. rep., RFC 4795 (Informational), Internet Engineering Task Force, 2007.

[20] K. Sundaresan, C. Donley, C. Grundemann, and V. Sarawat, "mDNS-DNS architecture," Oct. 25 2016. US Patent 9,479,422.

[21] S. Cheshire and M. Krochmal, "DNS-based service discovery," tech. rep., RFC 6763, February, 2013.

[22] L. Poettering and T. Lloyd, "Avahi." [accessed Jul 20, 2022].

[23] Apple Inc., "Bonjour." [accessed Jul 20, 2022].

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

[25] VirtualNetCom, "FeatherLite™ - Perfect Solution for Drone Deployment." [last accessed July 20, 2022].