Reliable and Efficient Mobile Edge Computing in Highly Dynamic and Volatile Environments

Minh Le*, Zheng Song†, Young-Woo Kwon*, and Eli Tilevich†
*Dept. of Computer Science, Utah State University
Email: minh.le@aggiemail.usu.edu, young.kwon@usu.edu
†Software Innovations Lab., Virginia Tech
Email: {sonyyt,tilevich}@cs.vt.edu

***Abstract*—By processing sensory data in the vicinity of its generation, edge computing reduces latency, improves responsiveness, and saves network bandwidth in data-intensive applications. However, existing edge computing solutions operate under the assumption that the edge infrastructure will comprise a set of pre-deployed, custom-configured computing devices, connected by a reliable local network. Although edge computing has great potential to provision the necessary computational resources in highly dynamic and volatile environments, including disaster recovery scenes and wilderness expeditions, extant distributed system architectures in this domain are not resilient against partial failure, caused by network disconnections. In this paper, we present a novel edge computing system architecture that delivers failure-resistant and efficient applications by dynamically adapting to handle failures; if the edge server becomes unreachable, device clusters start executing the assigned tasks by communicating P2P, until the edge server becomes reachable again. Our experimental results with the reference implementation show high responsiveness and resilience in the face of partial failure. These results indicate that the presented solution can integrate the individual capacities of mobile devices into powerful edge clouds, providing efficient and reliable services for end-users in highly dynamic and volatile environments.**

I. Introduction

Edge computing exploits data locality by processing massive amounts of sensory data collected by IoT devices "at the edge of the network." IoT devices, mobile devices, and edge servers process the locally collected data and transmit only the processed results to the cloud. In addition, edge servers function as gateways that coordinate the at-the-edge computation by assigning tasks to the available connected devices for execution. Traditional edge computing setups operate under the assumption of having a stable network connection, both between the edge server and the cloud, as well as between the local devices and the edge server.

In this paper, we consider edge computing environments that are *highly dynamic* and *volatile*. These environments are characterized by intermittent network connectivity, device mobility, and the presence of partial failure. In other words, the network connections both within the edge cloud and to the Internet are unstable or even non-existent. Users carrying the mobile devices involved can move at will, thus potentially affecting their devices' reachability to the edge cloud. Any of the computing devices involved, including the edge servers, can crash or become unreachable at any time.

The technical solutions presented herein enable reliable and efficient mobile edge computing in highly dynamic and volatile environments, such as those exemplified by disaster recovery scenes, battlefields, or expeditions to the wilderness.

A. Motivating Example

As a concrete example of the problem domain, consider Figure 1. A team of first responders reports to the location of a recent disaster. Each responder is supported by a collection of personal devices, both mobile and wearable. These devices are heterogeneous, in the sense that they differ in terms of their hardware resources, operating systems, and platform versions. A recovery vehicle hosts an edge server that also provides a WiFi access point (AP). Assume that the edge server's processing power is vastly superior to those of the personal devices, the WiFi AP covers the entire disaster recovery area, and the Internet connection's cellular signal is intermittent or non-existent.

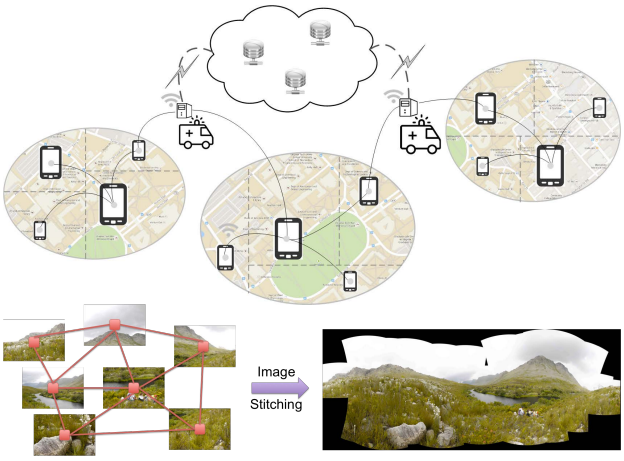

Figure 1: Motivating example: Image Stitching Service: (top) shows the overall service execution scenario; (bottom) shows the input images from different responders, the data-graph, and the stitched panorama.

Enabling the mobile devices and the edge server to cooperate as an edge cloud can greatly assist the responders in their mission. First, the responders must assess the situation on the ground and come up with a recovery strategy. To that end, they

978-1-5386-2859-1/17/$31.00 ©2017 IEEE

need to be able to efficiently map the entire recovery area. This task is called Image Stitching, a computer vision operation that glues adjacent pictures of an area together into a single panoramic image. The pictures are taken by the geographically dispersed responders using their respective devices, while the stitched panorama provides a detailed yet holistic view of the recovery area for the responders to facilitate their immediate recovery tasks (e.g., "What's around the corner from me?").

B. Technical Challenges

To realize the vision of enabling the heterogeneous mobile devices and the edge server to collaborate as a coordinated edge cloud, one must address two key technical challenges—device mobility and partial failure.

The responders need to move at will to attend to the recovery task at hand. This requirement entails that the devices forming the edge cloud may move to locations not covered by the limited WiFi AP, thus causing partial failure in the device-edge server distributed execution. Therefore, another technical challenge is to provide not only an efficient and reliable edge cloud architecture, but also resistant to partial failure caused by device mobility. The device-edge server distributed execution may operate in a peer-to-peer pattern, if the connection with the edge server is broken. The available network is likely to exhibit a high degree of volatility, with fluctuating bandwidth, latency, and packet loss rates.

For example, to stitch individual images into a single panoramic view, the individual devices that have captured their pictures need to send them to the edge server that has the computational and storage capacities to execute the stitching algorithm. However, the Image Retrieval service may operate in a peer-to-peer pattern, if the requested portion of the stitched map happens to be located on some nearby devices, or if the connection with the server is broken. The available network is likely to exhibit a high degree of volatility, with fluctuating bandwidth, latency, and packet loss rates.

This paper describes a solution that provides reliable and efficient mobile edge computing in highly dynamic and volatile environments. Our solution comprises a novel service architecture that includes the service infrastructure, a trusted portable store of vetted mobile edge services, each of which constitutes a self-contained executable module to be downloaded to mobile devices and invoked at runtime. In addition, the coordination of mobile services and the edge server is orchestrated by our edge server architecture, in a context-adaptive, failure-resistant fashion.

This paper makes the following contributions:

- **Mobile Edge Service Infrastructure**—a novel system component for deploying microservices on demand to heterogeneous mobile devices at the edge.
- **Adaptive Edge Service Architecture**—a novel distributed system architecture that dynamically reconfigures itself to provide resilience to partial failure.
- **Empirical Evaluation**—We rigorously evaluate the effectiveness and performance/energy efficiency of our reference implementation on a series of micro and macro benchmarks and applications.

The rest of this paper is organized as follows. Section II describes the proposed system architecture in detail. Section III presents our evaluations results. Section IV introduces the related works, and Section V presents future work directions and concludes the paper.

II. Technical Approach

In this section, we present our novel mobile edge service architecture, which enables reliability and efficiency in the face of the following technical challenge: the possibility of the mobile devices involved moving outside of their reachability range or failing for other reasons (i.e., partial failure).

To solve the aforementioned technical challenge, the architecture organizes mobile devices and edge servers into a hierarchical structure. All computing nodes, including the edge server and the set of available mobile devices, are connected using peer-to-peer communication interfaces, the main and the backup ones. When partial failure renders the edge server inaccessible, the backup interface makes it possible for the mobile devices to communicate with each other directly, with a cluster of mobile devices providing the edge services.

A. System Architecture

Here we provide an overview of our system architecture; the specific technical details are covered in the subsequent subsections, to which we provide forward references.

Our solution comprises a *service infrastructure for reliable and efficient mobile edge computing*. To understand how one can develop distributed mobile applications using this architecture, let us revisit the motivating example above. In that example, computing vision operations will be implemented as mobile microservice. Each service is a self-encapsulated unit of functionality managed by a service infrastructure. Each service includes a set of execution constraints that define the type of an edge computing device that can execute it in a given environment.

As the responders move around over time, the network topology of the mobile devices they are carrying is continuously changing. The technical challenge here is to hide the required underlying re-configurations of the mobile networks in response to the deployment of services on the continuously fluctuating—both in size and location—collection of mobile devices. We address these problems by providing *middleware support for dynamic and volatile environments*, whose adaptive facilities dynamically restructure the patterns of distributed communication in response to partial failure. Section II-C discusses the general design of the middleware system, while Section II-C3 discusses the details of handling partial failure.

B. Mobile Edge Services

Our service infrastructure features of application markets and mobile service repositories. Following the application market model enables mobile devices to automatically install and execute the required mobile edge services, while following

the service repositories model enables mobile application developers to implement the required functionalities as service invocations.

Since the platform on which a mobile edge service will be executed is unknown until the runtime, service developers are expected to provide several equivalent versions for each service to support execution on all major platforms. An important design assumption is that of mobile devices possessing limited resources, with some of the limitations making it impossible for a given device to execute a given service.

C. Adaptive Edge Service for Reliability and Efficiency

Our middleware provides efficient communication support for mobile microservices, coordinating their executions between heterogeneous edge computing participants, such as the edge server and mobile devices. In addition to the communication support, due to the dynamic and volatile nature of wireless networks, the middleware provides a novel failure handling mechanism, activated in cases of service execution failures or network disconnections.

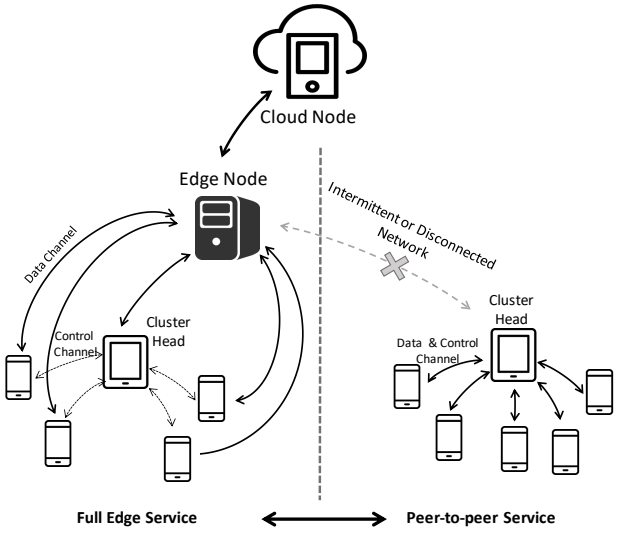

Figure 2: Edge cloud architecture.

Figure 2 gives an overview of the system architecture. The primary service execution model is client-server executing microservices at the edge server, as long as it is reachable. However, when the network or edge server becomes unavailable, mobile services are executed by means of nearby mobile devices in a peer-to-peer model. Then, as soon as the network connection between the edge server and mobile devices is restored, the service execution model is switched back to the client-server model.

1) Service Execution: Although our system architecture consists of two communication models: the client-server model and the peer-to-peer model, we use an adaptive system architecture on top of the *topic-based* publish/subscribe middleware [12] with three main constituent components: *Broker*, *Worker* and *Client*.

Algorithm 1 Service request algorithm.

```
1:  function SELECTDEVICES(client)
2:      workerList ← getAvailableWorkers()        ▷ Include worker on edge
3:      if (workerList.edgeAvailable()) then      ▷ Edge server is used
4:          client.offloadToEdge()
5:      else                                      ▷ P2P is used
6:          broker ← client.offloadToP2P()
7:          sortedDRLs[] ← sort(workerList.DRLs)  ▷ Descending order
8:          maxCombine ← 0
9:          availWorkerList[] ← null
10:         for all (DRL_i ∈ sortedDRLs) do       ▷ Find max (i × DRL_i)
11:             if (maxCombine < (i × DRL_i)) then
12:                 availWorkerList ←add workerList[i]
13:                 maxCombine ← (i × DRL_i)
14:             end if
15:         end for
16:         if (maxCombine < DRL_client) then     ▷ Prefer local execution
17:             broker.runAtLocal(client)
18:         else                  ▷ Job is distributed to the selected workers
19:             broker.sendToWorkers(availWorkerList)
20:         end if
21:     end if
22: end function
```

The *broker* hosted on either an edge server or mobile devices plays a critical role in executing mobile services, including (1) receiving service execution requests from clients; (2) looking up and downloading execution packages from a service repository; (3) selecting workers based on their resource availability and capacity; (4) delivering service execution packages to the selected workers; (5) gathering execution results from the workers and returning them back to the clients.

The *worker* located on an edge server and all mobile devices reports its resource availability and capacity to the broker that assigns tasks by sending service packages to workers and then execution results are sent back to the broker.

The *client* requests mobile service execution to the broker and waits for the result from the broker. If the broker cannot handle incoming new service requests due to the limited number of available workers, the client immediately cancels the service execution request and executes it locally.

When the edge server is available and the communication model is the client-server model, the edge server functions both as the broker and the worker. When the edge server is not available, the mobile device which is the cluster head works as the broker, and the other mobile devices in the cluster are taken as available workers. For a client device, it ignores the differences in the network communication model, and only needs to coordinate with the available broker. If partial failure happens (e.g., no available workers), the broker returns "execution error" to the client device, and the client device executes the service locally.

2) Optimal Device Selection: The result of executing a service remotely is often dominated by the hardware configuration of the service execution device, as well as its network conditions such as bandwidth/delay characteristics [10]. Thus, our approach finds an appropriate number of devices for the

current service execution environment to provide the best execution results with respect to performance. In the following discussion, we describe our service request algorithm in detail.

Our service request algorithm determines the best service execution model between an edge server and peers, and finds the most favorable devices based on the service execution capacity of each device. For the service execution capacity, we define the Device Responsiveness Level (DRL) metric, which is expressed as follows: $DRL = \sum_{i=\overline{1,M}} C_i \times C_{R_i}$, where C_i is the CPU speed of core i ($C_i = CPU_i$); C_{R_i} is the remaining percentage of the CPU core; and M is the number of cores. To find the most favorable mobile devices in a P2P network, a `Broker` selects N devices by comparing their DRL values with the client's DRL, divides a job into N equal pieces[1], and sends them to the corresponding workers through the following steps:

- Assume that we have D devices. First, we sort the available devices based on their processing power (DRL) in the decreasing order. Here, we have a list of devices $\mathcal{D} = \{d = 1, 2, ...D\}$, with their DRL $\{DRL_d \mid \forall d \in \mathcal{D}\}$, $DRL_{d1} \geq DRL_{d2}, \forall d_1 < d_2 \in \mathcal{D}$.
- For each $d \in \mathcal{D}$, we calculate the $d \times DRL_d$, and select $N = \max(d \times DRL_d), \forall d \in \mathcal{D}$. Then, select all devices from the sorted list that has higher DRL values than d, and the selected device set is: $\mathcal{N}^* = \{1, 2, ..., N\}$. The basic intuition behind this arrangement is, considering that the job will be divided into N equal pieces, the overall execution time can be estimated as the longest execution time of executing one piece on each worker, or say, the execution time on a worker device with the lowest DRL of all selected workers. If we use I to represent the job, and I/N to represent the piece on a worker, the overall execution time can be represented as $t = \frac{I}{N \min(DRL_d)} (\forall d \in \mathcal{N})$. As we sort the devices based on their DRL in decreasing order, $t = \frac{I}{N \times DRL_N}$, for the Nth device has the lowest DRL on all selected devices. Therefore, to minimize the overall execution time, is to first find a device d to maximize $d \times DRL_d$, and select all the devices whose DRL is higher or equal to d.
- We further consider the DRL of the client. If $DRL_{client} \geq N \times DRL_N$, which means that executing the job on the client is faster than executing the job on the peers, we choose to execute the job on the client; Otherwise, if $DRL_{client} < N \times DRL_N$, we choose to distribute the job equally into N pieces, on the selected workers $\mathcal{N}^*$.

3) Handling Network and Service Failures: Clients maintain two communication channels with an edge server and a cluster head. By exchanging a heartbeat message between an edge server and clients, each client can check the availability of the edge server and network status to both the edge server and nearby devices. If a client does not receive an acknowledgement from the edge server due to the network disconnection, it informs nearby clients of the network failure through a cluster head. Then, clients immediately switch their service execution model to the P2P mode and continue the failed service execution through a peer-to-peer network.

Algorithm 2 Handling network and service failures.

```
1:  function CHECKHEARTBEAT()
2:      openNewHBConn()                          ▷ open new Heart-beat connection
3:      while (Thread.isInterrupted()) do
4:          try
5:              wait(HEARTBEAT_PULSE)  ▷ HEARTBEAT_PULSE = 3s
6:              sendHeartbeatToEdge()
7:              resp ← waitForResponse(TIMEOUT)      ▷ TIMEOUT = 2s
8:              if (resp != null) then
9:                  notifyOKEvent()
10:             else                              ▷ If unable to receive response
11:                 notifyFailedEvent()
12:             end if
13:         catch (NetworkException)
14:             try
15:                 wait(REESTABLISH)                ▷ REESTABLISH = 3s
16:                 closeCurrentHBConn()
17:                 openNewHBConn()          ▷ Reopen heartbeat connection
18:                 notifyRestoredEvent()      ▷ Network has been restored
19:             catch (Exception)
20:                                              ▷ When attempt failed again
21:                                                ▷ Silently start a new loop
22:             end try
23:         end try
24:     end while
25:     closeCurrentHBConn()          ▷ Close current Heart-beat connection
26: end function
```

Algorithm 2 explains our failure handling strategy. Any clients maintain one heartbeat communication to the edge server and they periodically send heartbeat requests to the server and wait for the response. If the response is not received within a timeout, a `failedEvent` will be dispatched to the holder to notify of the network failure. Once the client receives `failedEvent`, it immediately switches its service execution model to the P2P mode and continues the failed service execution. By exchanging heartbeat messages with a the cluster head, the client can also detect a service failure in the P2P mode and execute the failed service locally. In the meantime, the client keeps attempting re-connection to the edge server by sending a heartbeat request. If the client receives a heartbeat response from the edge server, it dispatches `restoredEvent` to the holder and restores the system to the normal state[2].

III. Evaluation

We evaluate the effectiveness of our approach through a micro benchmark and realistic case studies. Specifically, we conduct two test cases to ascertain how efficient and reliable our system would be in highly dynamic and volatile mobile edge computing environments[3]. The testbed for experiments has been built up with various Android devices featuring WiFi Direct and one edge server.

[1]To make the data partitioning problem simple, we adopted the Hadoop's idea that splits data into multiple chunks of the same size.

[2]For the details, please see our demo at: http://youtu.be/7dd1EQFb_vk

[3]Our system implementation and evaluations can be found at: https://github.com/minhld/Pub-Sub-Middleware

Device	CPU	RAM	Battery	OS
Moto G4	Octa 1.5GHz	2GB	3000mAh	6.0.1
G4	Quad 1.5GHz	3GB	3000mAh	5.1
Asus ZF2	Quad 1.7GHz	3GB	2400mAh	5.1
BLU R1	Quad 1.3GHz	1GB	2500mAh	6.0
S3	Quad 1.4GHz	1GB	2100mAh	4.4
Dell PC	i7 3.6GHz	8GB	N/A	Win10

Table I: Devices used in the experiments.

A. System overhead

In this experiment, we evaluate the system's overhead by measuring the execution time of each main component of our system architecture, which include the client, the broker, and the worker, both on the edge server-based and the peer-to-peer-based networks. We (1) pre-install an empty service, which sleeps for 15 seconds on the edge server and P2P workers and (2) place mobile devices in a nearby area for fast, stable WiFi Direct communication. We timestamp the start and end times of each component executing its job and then aggregate all the times, excluding the service execution time (i.e., 15 seconds). Figure 3 shows the total overhead time, which can be disregarded.

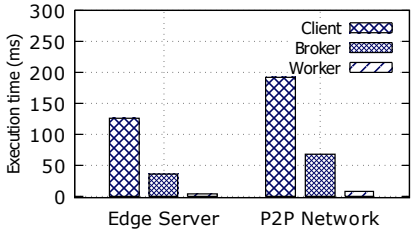

Figure 3: Aggregate overheads in two different networks.

B. Case Study

To evaluate our system's implementation in realistic scenarios, we conduct two case studies:

- **Image Processing Service:** splits an image into several parts, and each device then blurs one image part and sends the results back to the client, which merges all these partial results into a single image.
- **Internet Sharing Service:** provides Internet connectivity to nearby devices that lack a cellular data plan. The service loads a web page in the device's mobile browser by engaging surrounding Internet-connected devices.
- **Word Counting Service:** splits an e-book into several text parts, and each device then find most frequent words and sends the results back to the client, which merges all these partial results and shows the top 50 most frequent words.

1) Image Processing Service: For this experiment, we implemented an image blurring service using the Gaussian convolution. An image is split into N equal parts vertically, and then each part is sent to a worker. Each worker executes the Gaussian convolution definition to blur its image part and sends the result back to be merged into a complete image.

We first requested the service to the edge server, and then disconnect the network between the edge server and a client, resulting in switching the service execution model to the P2P mode. We measured the total execution time of the client that elapsed between the initiation of the service request and the arrival of all results. Figure 4 shows that the edge server-based and P2P-based service execution. Although the edge server-based service execution outperforms the performance of P2P-based service executions, as the number of mobile devices increases, the performance of the P2P-based service model is also significantly improved.

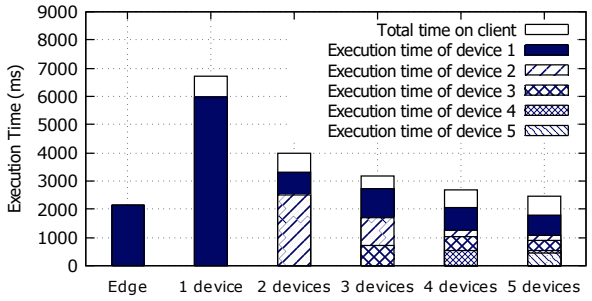

Figure 4: Image processing: performance comparison of the edge server- and P2P-based execution model.

Then, to evaluate our peer selection approach, we compared the estimated execution time with the actual time taken by the P2P collaboration in five scenarios, which engage between 1 and 5 devices. Figure 5 (left) shows the estimated and actual lines having the same trend and close values. We found that this trend approximation also occurs when starting the service execution from different devices in the P2P network. Figure 5 (right) describes the similar trends on 3 devices with different resource capacities, when requesting the same service from different mobile device in the P2P network.

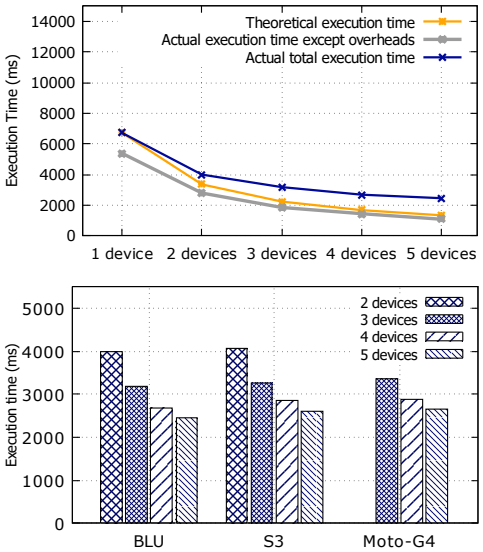

Figure 5: Image processing: comparison of the estimated execution time and the actual execution time (top); performance comparison of different clients in P2P networks (bottom).

2) Word Counting Service: This service will be used mostly in the section III-B4. Although a word-counting service would not represent a typical example targeted by our solution, we use this canonical computation-intensive scenario to compare the respective performance and energy efficiency of the edge server and the P2P networks under multiple configurations. Figure 6 shows the similar trends as in the image processing service above, indicating that our scheduling algorithm allocates the optimal number of devices to maximize the performance and minimize the energy consumption.

Figure 6 shows the similar trends as in the image processing service above, indicating that our scheduling algorithm allocates the optimal number of devices to maximize the performance and minimize the energy consumption.

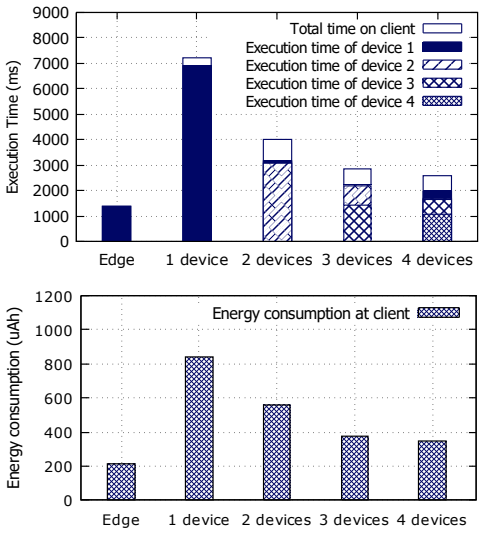

Figure 6: Word-counting: execution time (top) and energy consumption (bottom) measured at the client when executing the service at the edge server and peers.

Figure 7 evaluates the accuracy of Algorithm 1 by comparing its output with the actual measured performance numbers, summarized in Figure 6. The graph shows that the two lines follow the same trend, indicating that the algorithm approximates the actual performance with an acceptable level of accuracy.

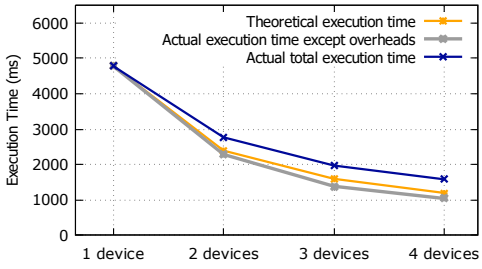

Figure 7: Word-counting: comparison of the estimated execution time and actual execution time on two different devices.

3) Internet Sharing Service: Next, we study the Internet sharing service, an example of sharing the nearby devices' computing resources, including networks, files, sensors (e.g., GPS, motion, environmental etc.). We design the Internet sharing service to distribute requests including *URL*, *n* – the number of devices in the cluster, and *index* – the index of a device. Each device (including the client) downloads the HTML text contents of a web-page from the *URL*, collects its resource URLs (images, audio, videos etc.) and downloads a batch of URLs according to its *index* position of *n* devices.

In this experiment, we assume that only one device, lacking Internet access, sends the same content-downloading request to an edge server or a cluster head. The first two bars in Figure 8 show the total execution time measured at different devices when connecting to the edge server, while the next five bars show the total execution time on each mobile device when connecting to different cluster heads. To show how dissimilar resource capacities lead to different performance characteristics, we configured our testbed to sequentially select different devices as the cluster head for each experiment. Unlike the first two bars, which are almost the same, the cluster heads have dissimilar resource capacities, with their respective performance levels fluctuating in the wider range between 37 and 42 seconds, being obviously slower than the edge server.

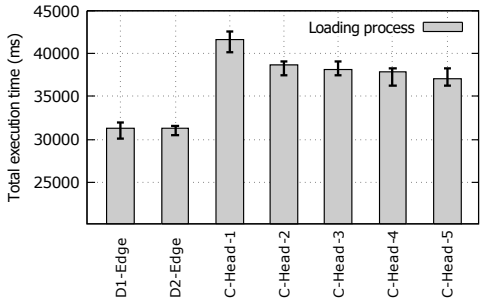

Figure 8: Internet sharing performance: the edge server- vs. P2P-based service execution with different cluster heads.

4) Failure handling: As discussed in Section II-C3, our failure handling mechanism can switch the remote execution back and forth between the edge server and the P2P network in response to the edge server becoming unavailable and available again, as well as the network getting disconnected and reconnected. We evaluate the efficiency and effectiveness of our failure handling mechanism by implementing a word counting service, a well-known use case of multi-processing setups, such as Apache Hadoop.

The word counting service searches for 50 most frequently used words from a textbook. First, a client loads the entire text from a book and attaches it to the request message. Then, a broker finds the number of available workers, divides the text into the same number of parts, and sends them to the workers.

Switching to a P2P network: First, we enable the edge server, so that the client can offload the word-count service normally there within 4.3 seconds (Figure 9). Then we repeat the test but shut the edge server down at the 3rd second; then the client detects the system failure at around the 5th second (the timeout for failure detection is 3 seconds) and

immediately initiates a new offloading session to the P2P network. The P2P collaboration returns the result within 23 seconds, and the total process takes around 29 seconds. From this moment, if we offload the service again, the offloading will be dispatched directly to the P2P and completed within around 23 seconds.

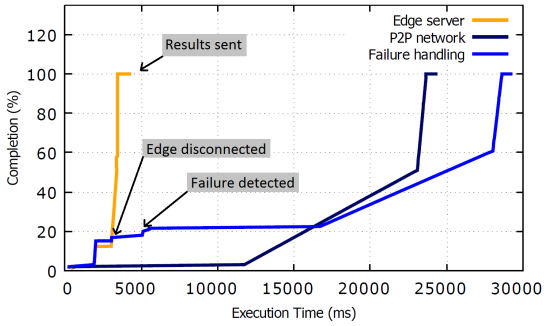

Figure 9: Execution time of the *word-count* service measured in 3 cases: (1) Edge server, (2) P2P network with 1 worker and (3) Failure handling from the Edge to P2P.

Edge server restoration: In this experiment, we observe and examine how our system behaves in consecutive scenarios: (1) run normally on the edge server, (2) detect failure, (3) switch to P2P, and (4) restore back to the edge when the network reappears. To this end, we use the same test-bed as in the above experiments, thus enabling the client to offload a number of service requests; then we measure the total time of each request at the client. In this test, the client delivers three different services: empty, word-count, and image processing (the image processing service is discussed in Section III-B1).

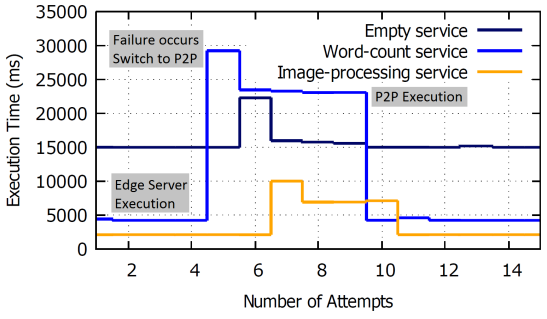

Figure 10: System performance when requesting multiple services (the edge server fails between the 4^{th} and 5^{th} attempts).

Figure 10 depicts the overall performance of our system in each type of service requests with a number of continuous attempts. Particularly, in the case of word-count service, the first 4 requests are resolved by the edge server within 4.3 seconds. During the 5th request, the edge server is shut down, with the client immediately detecting this event and switching to the P2P execution mode, completing the request in 29 seconds. After that, the upcoming requests from the 6th to 9th requests are resolved by the P2P configuration in 23 seconds on average. When the edge server comes back, the client reestablishes the connection and pushes the 10th and subsequent requests to the edge, thus reverting to the original normal execution mode.

C. Discussion

Our approach offers a number of advantages and has several limitations that we discuss in turn next.

1) Advantages: Mobile edge services serve as building blocks, enabling application developers to design mobile applications that take advantage of the complimenting strengths of nearby devices from the resource provisioning perspective. In other words, developers can orchestrate the execution of an application's functionalities on the devices whose resources are the most suitable and abundant for given execution tasks.

This software architecture eliminates the need to accept executable code from nearby devices by introducing a trusted third-party intermediary. The introduction of the intermediary component increases the trustworthiness of the distributed execution model. However, for this intermediary component to become widely applicable, multiple stakeholders in the technology will need to come to an agreement, which may be hard to accomplish.

Once a service is downloaded to a mobile device, the Internet connection is no longer required, making it possible to execute mobile services in environments with limited or intermittent wide-area networks. In fact, this architecture can increase the utilization of nearby mobile devices in such execution environments.

Finally, the ability to address resource scarcity makes this architecture potentially suitable as a solution for orchestrating the execution of IoT setups, in which each participating device is known to posses specialized unique functionality (e.g., sensing, media capture, etc.) while lacking general hardware or software resources.

2) Limitations: The trustworthiness of mobile edge services hinges entirely on the reputation of the trusted intermediary component, thus restricting this distributed execution model to environments that provide such trusted components. In addition, the high dynamicity of the mobile context increases the risk that no suitable nearby device may end up being available for executing a service with a specified set of requirements. To defend against these risks, mobile developers need to provide back-up options for executing services, either with relaxed QoS requirements or using the local resources.

Since one cannot predict what platforms will be run by the available nearby devices, service developers have no choice but to provide multiple versions of their services, equipped to run on all major platforms. This design feature increases the developer workload, even though JavaScript execution may provide a reasonable cross-platform solution.

IV. Related Work

The work presented here is related to other complementary efforts that improve the reliability and efficiency of mobile distributed execution, including frameworks, peer-to-peer networking, code migration, and computation offloading.

Alljoyn[4] is an open source framework that hides the complexity of network communications for application programmers. By providing interoperability between multiple platforms without any transport layers, Alljoyn makes the integration and initiation of network communication easy and straightforward. Before the Wi-Fi Direct, many efforts have focused on optimizing peer-to-peer network based on existing short-range/wireless communication technologies available on mobile devices including Bluetooth, Wireless IEEE802.11 and cellular communication link [5], [11].

In the category of wireless-based P2P communication, before the Wi-Fi Direct technology, several efforts utilized other wireless communications to establish P2P networks, such as media sharing system sin urban transport using Bluetooth [9], resource sharing using cellular networks [16], and radio resource sharing over ultra-wideband [3]. Built on top of Wi-Fi P2P, Rio [1] leverages I/O system devices to capture and share contents and resources between the existing applications running on different devices without any modification. Some of their applications are multi-system photography and gaming, singular SIM card for multi-devices, music and video sharing.

Another related work direction is code migration to update existing, legacy systems [4]. Similar to our approach, code migration mechanisms are mainly used to run code in different memory spaces (e.g., running C++ code on multi-core systems [2], running JavaScript code on a server [14], object-level migration for distributed systems [15], thread-level migration through middleware [13]). These code migration approaches have influenced the design of offloading mechanisms in the mobile computing area.

Finally, our work shares objectives and techniques with execution offloading [6], [7], well-known mobile application optimizations that execute the resource-intensive functionality at cloud-based servers to avoid draining the mobile device's battery. In this paper, we generalize these offloading mechanisms using two different distributed execution models—client/server and peer-to-peer communication models.

V. Future Work and Conclusion

As a future work, we plan to explore additional diverse failure handling mechanisms by exposing them as reusable components, activated in response to the underlying system behaving abnormally, based on our prior work on hardening remote services with network volatility resilience [8].

In this paper, we present a service middleware architecture and reference implementation that execute services reliably and efficiently on available devices, both mobile and stationary, accessed via self-adaptive communication channels. Our solution centers around the characteristics of highly dynamic and volatile environments, in which the network connection between the devices forming the edge cloud is intermittent. The presented solution automatically detects and handles partial failure of the network, by switching between client-server and peer-to-peer mobile edge execution modes. Our experimental evaluation shows that our solution enables resilient and efficient mobile edge execution over unreliable networks, typical of highly dynamic and volatile environments, heretofore not supported by edge clouds.

[4]https://allseenalliance.org/framework

VI. Acknowledgment

This research is supported in part by the National Science Foundation through the Grant CCF-1649583.

References

[1] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: a system solution for sharing i/o between mobile systems. In *Proceedings of the 12^{th} Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[2] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload – automating code migration to heterogeneous multicore systems. In *Proceedings of the 5^{th} International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[3] F. Cuomo, C. Martello, A. Baiocchi, and F. Capriotti. Radio resource sharing for ad hoc networking with uwb. *IEEE J.Sel. A. Commun.*, 20(9):1722–1732, Sept. 2006.

[4] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing incremental code migration with XML. In *Proceedings of the 22^{nd} International Conference on Software Engineering*, 2000.

[5] H. C. Frank H. P. Fitzek. *Mobile Peer-to-peer (P2P): A Tutorial Guide*. Wiley, 2009.

[6] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10^{th} USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[7] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32^{nd} International Conference on Distributed Computing Systems (ICDCS '12)*, June 2012.

[8] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *ACM/IFIP/USENIX Middleware 2009*.

[9] L. McNamara, C. Mascolo, and L. Capra. Media sharing based on colocation prediction in urban transport. In *Proceedings of the 14^{th} ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2008.

[10] T. Nabi, P. Mittal, P. Azimi, D. Dig, and E. Tilevich. Assessing the benefits of computational offloading in mobile-cloud applications. In *Proceedings of the 3^{rd} International Workshop on Mobile Development Lifecycle*, 2015.

[11] R. M. P. Bellavista, A. Corradi and C. Stefanelli. Context-aware middleware for resource management in the wireless internet. In *IEEE Transactions on Software Engineering*, pages 1086–1099. IEEE, 2003.

[12] M. Rostanski, K. Grochla, and A. Seman. Evaluation of fairness in message broker system using clustered architecture and mirrored queues. In T. Czachrski, E. Gelenbe, and R. Lent, editors, *Information Sciences and Systems*, pages 407–417. Springer International Publishing, 2014.

[13] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the 2^{nd} International Symposium on Agent Systems and Applications and 4^{th} International Symposium on Mobile Agents*, 2000.

[14] T.-L. Tseng, S.-H. Hung, and C.-H. Tu. Migratom.Js: A JavaScript migration framework for distributed Web computing and mobile devices. In *Proceedings of the 30^{th} Annual ACM Symposium on Applied Computing*, 2015.

[15] M. Yoshida and K. Sakamoto. Code migration control in large scale loosely coupled distributed systems. In *Procs. of the 4th International Conf. on Mobile Technology, Applications, and Systems*, 2007.

[16] C. H. Yu, K. Doppler, C. B. Ribeiro, and O. Tirkkonen. Resource sharing optimization for device-to-device communication underlaying cellular networks. *IEEE Transactions on Wireless Communications*, 10(8):2752–2763, 2011.