An Ensemble Approach to Fault Tolerance in Mobile Cloud Computing Offloading

MSc Research Project
MSc in Cloud Computing

Erel Ozturk
Student ID: 21245312

School of Computing
National College of Ireland

Supervisor: Vikas Sahni

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Erel Ozturk
Student ID:	21245312
Programme:	MSc in Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Vikas Sahni
Submission Due Date:	14/08/2023
Project Title:	An Ensemble Approach to Fault Tolerance in Mobile Cloud Computing Offloading
Word Count:	7343
Page Count:	20

Signature:	Erel Ozturk
Date:	9th August 2023

An Ensemble Approach to Fault Tolerance in Mobile Cloud Computing Offloading

Erel Ozturk
21245312

Abstract

With the increasing complexity of mobile applications, offloading compute-heavy tasks to cloud servers has become a popular solution. However, due to the mobility of mobile devices and server failures, offloading is an error-prone process. Fault tolerance techniques, such as checkpointing and replication, are used to address this issue. A combination of various techniques helps consolidate the performance by combining their strengths.

This research looked at an ensemble technique of checkpointing and replication to ensure fault tolerance during mobile cloud computing offloading (MCCO). Conducting a quantitative analysis of the findings, such as network data usage and memory utilization, the results indicate that the ensemble method performs better than checkpointing and replication as standalone fault tolerance techniques.

1 Introduction

Over the past years, mobile applications have dramatically increased the level of complexity of the technology they incorporate. As mobile devices improve the technology they use, mobile applications adapt in parallel McLean et al. (2018); Noor et al. (2018). Even though the most recent mobile devices have improved a lot compared to the past, they still have some limitations such as battery life, storage, or computation power. Therefore, when there is a computation-heavy task to be executed, the mobile device can face some issues regarding the previously mentioned limitations. Mobile Cloud Computing (MCC) is a paradigm that has emerged to address these kind of limitations Almaiah and Al-Khasawneh (2020); Shamshirband et al. (2020).

When there is a computation-heavy task to be executed in a mobile application, instead of consuming the mobile device's resources, the task is sent to the cloud server. The cloud server executes the task and returns the results to the mobile device. This procedure is called task offloading. The main reason behind MCCO is to make use of cloud resources since a cloud server's resources, such as computation power and storage, are much wider than a mobile device's resources. This process also benefits mobile devices in terms of computation capability, battery consumption, network connectivity, etc Fernando et al. (2013).

On the other hand, MCCO faces some hurdles such as network stability and server inaccessibility. Since the whole procedure is based on data exchange between the cloud server and the mobile device through the network, it is necessary to have a stable connection between the client and the server to prevent package loss Rahmani et al. (2021).

However, as mobile devices are designed to be constantly on the move, the stability of the connection cannot be guaranteed all the time. Moreover, random server failures can occur due to a wide variety of reasons. Thus, MCCO is an error-prone process in terms of connection stability.

The success of the whole procedure can be crucial since the entire application may be relying on this computation-heavy task. Therefore, it's an important necessity to enhance MCC infrastructures with the most suitable fault tolerance method to sustain availability. Many fault tolerance methods can be applied to an MCC infrastructure such as checkpointing, replication, autonomous recovery, task migration, etc. In this work, checkpointing and replication methods will be discussed as these two techniques are more focused on the high availability of the system.

As mentioned earlier, MCC infrastructures should be strengthened with fault tolerance techniques to sustain availability during the task offloading process. The explained two fault tolerance techniques, checkpointing, and replication, are two commonly suitable fault tolerance techniques for MCC infrastructures. However, it is crucial to know which one to apply under which circumstances and the trade-offs between these two methods.

Research Question. The above research problem motivates the following research question: How does the ensemble method improve performance compared to checkpointing and replication within the context of ensuring fault tolerance in MCCO?

This paper proposes an ensemble technique for ensuring fault tolerance in MCCO and provides a comparative work between checkpointing, replication, and ensemble technique. As mentioned earlier, a big portion of errors that occur during MCCO is due to intermittent network connectivity of mobile devices and random server inaccessibility. That is why, the study focuses on faults that are caused by intermittent network connectivity of mobile devices and server failures. The study focuses on checkpointing and replication during the experiments. The proposed technique is emulated in a real-world MCC infrastructure using virtual mobile devices and remotely deployed servers.

The rest of the document is organized as follows: In Section 2, relevant background information and a discussion of the current state of the art is provided for fault tolerance in MCCO. Section 3 provides an overall look into the following methodologies for producing data and evaluating them. Section 4 discusses the design and architecture of the underlying system. Section 5 elaborates on the implementation details. Section 6 gives a detailed analysis of the results and findings of the conducted experiments. Section 7 provides a brief conclusion and recites the findings and discusses the possible future work of the conducted study.

2 Background and Related Work

This section provides a high level of background information about the terminology discussed throughout the paper as well as a literature survey related to the topic. Checkpointing and replication are briefly explained below:

2.1 Checkpointing

Checkpointing is one of the most commonly used fault tolerance methods. It refers to taking a snapshot of the current state of the system periodically so that if a failure occurs, it will be possible to roll back to the most recent snapshot of the system instead of starting from the beginning. The frequency of snapshot-taking depends on the requirements and error expectancy of the system. The snapshots are saved in a persistent storage unit or in memory, again depending on the system requirements, and retrieved when necessary. Although it is also possible to use checkpoints at the kernel and user levels, the paper discusses the use of checkpointing at the application level of the system as application-level checkpointing provides more granularity and allows for faster and more efficient recovery in case of a failure.

The main goal of checkpointing is to reduce the amount of task repetition when a failure occurs. Figure 1 represents an execution task, which is divided into 10 subtasks. Let's assume the snapshot period is once per task finishes (illustrated as cpn where n is a number between 1 to 10). If there is a failure on subtask 7, the execution can continue from the state where the 6th task was finished, being cp6, by rolling back to the latest checkpoint.

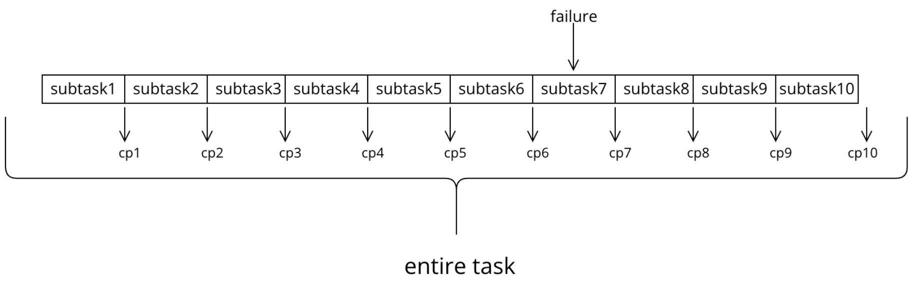

Figure 1: Checkpointing-applied Task Execution

There are 3 checkpointing algorithms based on the placement of the checkpoints; uncoordinated checkpointing, coordinated checkpointing, and communication induced checkpointing Sowjanya Lakshmi et al. (2022).

- **Uncoordinated Checkpointing:** In the uncoordinated checkpointing method, each process takes its snapshot independent from the others. This situation can cause the checkpoints to be inconsistent which might lead to a domino effect when a failure occurs, and the system tries to roll back to the latest consistent state.

- **Coordinated Checkpointing:** In the coordinated checkpointing method, unlike uncoordinated checkpointing, there is a globally consistent state that's agreed upon by all other processes. When a process takes a snapshot, it coordinates with others to determine the consistent global state so that whenever a failure occurs, the system will only be rolled back to that global state.

- **Communication Induced Checkpointing:** In communication-induced checkpointing (CIC), the main goal is to optimize the number of checkpoints taken to reduce the required amount of data. In CIC, the snapshots are taken according to the messages to be transferred rather than the processes. This might reduce some overhead; however, it might also lead the system to lose some important checkpoints Sowjanya Lakshmi et al. (2022).

2.2 Replication

Replication is another widely used fault tolerance technique among distributed systems. In the replication technique, the main goal is to increase reliability by creating multiple replicas of data or resources across different nodes. Therefore, whenever a node failure occurs, the task execution can continue from another node. Thus, the system won't lose access to compute resources or data in the case of an error.

There are many types of replication such as data replication, service replication, state machine replication, database replication, etc. Among all, data replication and service replication are the most common replication types.

- **Data Replication:** In data replication, data inside a node is constantly synchronized with other nodes that hold a copy of this portion of data. Whenever a node receives a "write" request, the current version of the data in this node is updated, and then the update is reflected upon all the nodes that hold a copy of the old version of the data. Thus, all copies of the data among all other nodes are up to date.

- **Service Replication:** In service replication, the application or service itself is replicated to other nodes. In the case of a node failure, the requests will be addressed to another node with the same version of the application at hand so that clients will still be able to receive responses to their requests.

2.3 Offloading

Before diving deep into the application of fault tolerance techniques, it's important to understand ensuring fault tolerance is a crucial aspect of MCCO. Akherfi et al. (2018) researched the current offloading frameworks and techniques as well as the critical issues related to them. The researchers listed fault tolerance as one of the general issues and challenges related to MCCO due to the freedom of movement of mobile users. In a similar work, Saeik et al. (2021) researched combining edge and/or cloud to promote task offloading with an emphasis on mathematical aspects and artificial intelligence to overcome some challenges related to it. The researchers stated ensuring fault tolerance is an open challenge. They think fault tolerance should be ensured since the network bandwidth and data exchange rates may vary due to freedom of movement. Chaâri et al. (2022) presented research about task offloading in ground and flying robots. They believe one of the most critical challenges related to computation offloading in mobile robots is the dependency on the network. They stated that a malfunction can happen either due to a temporary connectivity issue of the robots or a problem related to the remote server. They inferred that a sort of fault tolerance functionality must be in place to recover from these kinds of malfunctions. The work done by Zalat et al. (2022) is an interesting contribution to the field of mobile computing. The authors proposed a novel strategy for deciding the most optimal execution location for each device's offloading task. They were able to benefit power consumption and execution time by combining Niching Genetic Algorithm (NGA) and Markov Decision Process (MDP) to decide on execution location. However, the authors identified fault tolerance as an area requiring further research.

It is also worth mentioning an important paradigm in MCCO, which is task scheduling. Task scheduling refers to the decision-making of which tasks should be executed locally and which tasks should be offloaded to the cloud. Ali and Iqbal (2022) proposed a novel task scheduling algorithm by classifying tasks based on their priority and accomplished to enhance overhead during the task arrival. ul Hassan et al. (2023) proposed a novel strategy for task scheduling which involves classifying tasks based on their characteristics and providing the tasks with the most optimal and suitable resources in the cloud. The authors focused on completion within the given deadline as well as minimizing mobile application costs. Shahid et al. (2021) proposed a three-layer architecture for offloading in MCC. The first layer is the mobile devices, the second layer is the task scheduler layer, and the third layer is the cloud layer. The authors allocate a separate virtual machine for each offloaded task in the cloud layer.

2.4 Fault Tolerance

Fault tolerance refers to the capability of staying functional regardless of errors. A fault-tolerant system should be able to know how to recover when particular errors occur Kumari and Kaur (2021). Goyal and Saini (2016) proposed a fault-tolerant computational offloading framework with an emphasis on reducing energy consumption. The researchers' methodology involves dividing the task to be offloaded into smaller subtasks. Even though the proposed framework was able to decrease power consumption, the study does not involve a post-offloading algorithm to process data. Sowjanya Lakshmi et al. (2022) presented the Communication Induced Adaptive Checkpointing Fault Tolerance Mechanism (CIAC-FTM) in their research. This is a checkpointing-based mechanism that focuses on reducing the overhead of checkpoints based on the type of fault. Although the authors managed to reduce the overhead of checkpoints, they did not consider enhancing their system with replication. Lakhan and Li (2019) conducted research and introduced a mobility-aware adaptive task offloading algorithm (MATOA). This algorithm makes a task offloading decision at run-time and chooses the most optimal network channels for a seamless offloading experience. Even though the proposed algorithm benefits response time, the study does not consider the impact of different types of tasks. Naeem and Sadik (2014) introduced a middleware that offloads computation-heavy tasks on cloud resources and performs adaptive or dynamic replication. However, the authors did not consider checkpointing in their work.

Manoj Kumar Malik (2022) proposed a task replication-based fault tolerance algorithm that replicates tasks across multiple nodes in a cluster. The authors were able to benefit execution time and resource utilization while ensuring fault tolerance. However, the authors did not consider another fault tolerance technique. Thus, the system is subject to the limitations of replication. Abdulhamid et al. (2016) proposed a dynamic clustering league championship algorithm (DCLCA) which is a fault-tolerance-aware algorithm. The proposed algorithm exposes the currently available resources and reduces the untimely failure of autonomous tasks. The authors stated that they reduced the failure rate. However, the system does not involve a fault recovery mechanism for the remaining failures.

Lakhan et al. (2021) proposed a novel technique that is called Mobility Aware Blockchain-Enabled Offloading Scheme (MABOS). MABOS is an offloading technique that makes use of blockchain technology and considers fault tolerance and security aspects of the system. The authors followed the consensus mechanism, a blockchain-based fault tolerance technique to ensure fault tolerance.

One of the most common methodologies that researchers have preferred to follow is to group mobile devices and apply the most suitable fault tolerance method to each group. In this context, Zhou and Buyya (2017) conducted research and proposed a group-based fault-tolerant mechanism called GFT-mCloud. They group mobile devices based on their processing capacity, mobility, and reliability. The authors then applied different fault tolerance techniques to each group. The proposed mechanism benefits the makespan of the system by 5-10%. However, this work only considers a standalone fault tolerance method for each group. Lee and Gil (2019) did the grouping based on the availability and reliability of the mobile devices and applied either checkpointing or replication to each group. The authors introduce scheduling groups (SG) for each group to effectively utilize all system resources. The proposed method favoured performance by reducing the execution time by 60%. However, the research is limited by the fact that mobile devices' resource information changes dynamically, and thus, groups requiring to adapt in parallel. Yet another research is conducted by Park et al. (2014) that involves group-based fault tolerance. The grouping was done according to the mobile devices' availability and mobility. Similar to the other studies, the authors applied checkpointing or replication to each group based on the group's reliability. To decide which technique to apply to each group, they followed the following methodology:

- **groups with the highest reliability:** Checkpointing
- **groups with middle-level reliability:** A mixture of checkpointing and replication
- **groups with lowest reliability:** Replication

The authors were able to favour the system's overall performance (such as execution time). However, solely relying on replication for the groups with the lowest reliability may not be an optimal choice in terms of reliability. Replication can address only a limited set of errors such as server failures.

Even though grouping the mobile devices based on their similar aspects is an effective technique and a great contribution to MCC, it can be expensive in terms of computation to apply grouping and fault tolerance method decision algorithms as the groups get wider. Moreover, in the mentioned research that involves the device grouping method, each group receives only one fault tolerance technique. Thus, the system is limited to the bottle-necks of the applied standalone technique.

Table 1 gives an overall look into the surveyed works.

3 Methodology

It is important to address the most appropriate methodology to both generate the data to be analysed and the technique to analyse the generated data. This section describes how the experimental environment was set up to generate the metrics to be analysed as well as the methodology followed to analyse these metrics.

3.1 Experimental Setup

To create the necessary experimental setup, a mobile application has been implemented using Flutter framework[1] to execute the mentioned techniques. The mobile application was run on a local emulator provided by the Android Studio[2]. The mobile application, which is the client of this setup, communicates with two servers that are deployed

[1]https://flutter.dev/

[2]https://developer.android.com/studio

Table 1: Comparison of Related Works

Reference	**Methodology**	**Gains**	**Limitations**
Goyal and Saini (2016)	Dividing tasks to be offloaded to subtasks	Improved energy consumption	No post-offloading algorithm
Sowjanya Lakshmi et al. (2022)	CIAC-FTM	Reduced overhead of checkpoints	No use of replication
Lakhan and Li (2019)	MATOA	Improved response time	No consideration of different types of tasks
Manoj Kumar Malik (2022)	Replication	Execution time and resource utilization	No use of other fault tolerance methods
Abdulhamid et al. (2016)	DCLCA	Reduced failure rate	No fault recovery mechanism
Zhou and Buyya (2017)	Group-based fault tolerance	Increased makespan by 5-10%	Standalone fault tolerance mechanism for each group
Lee and Gil (2019)	Group-based fault tolerance	60% reduction in execution time	Dynamic change of resource in mobile devices
Park et al. (2014)	Group-based fault tolerance	Performance improvement	Solely relying on replication on lowest reliability groups

to AWS EC2 instances in different regions, Ireland (eu-west-1) and London (eu-west-2) through a web socket. Server-side implementation of the application is done using Python's FastApi[3] framework.

Using socket communication, the client sends a message to the server as a simulation of task offloading. The server then starts executing a computation-heavy task. The simulated computation-heavy task is a loop that counts from 0 to the given number by the client with a 1-second wait at each loop. According to the scenario that is being simulated, the server applies the corresponding fault tolerance technique.

The reason why Flutter was preferred for mobile application implementation is the wide variety of libraries available in Flutter. Using some of the libraries made it possible to establish a web socket (*socket_io_client* and *web_socket_channel*), recognize if the device has a network connection (*connectivity_plus*), etc. The reason why AWS EC2 instances are selected for the servers is the free access that the National College of Ireland provides. Therefore, no further bills have been incurred during this experimental setup.

It was chosen to set up communication between the client and the servers through a web socket because momentary health checks are required to understand if the client is still connected or not. If this mechanism was implemented with HTTP requests, the cost of network data usage as well as the latency of each request would dramatically increase as HTTP requests are more expensive than web socket communication in terms of network data usage and latency Pimentel and Nickerson (2012).

[3]https://fastapi.tiangolo.com/

3.2 Measurement & Data Analysis

To collect data, Android Profiler, provided by Android Studio, is used during the experiment. Android Profiler provides developers with a real-time device metric monitor. Using this tool, CPU usage, RAM usage, power consumption, and network data usage is captured for both scenarios. Apart from these metrics, the total execution time, which is logged by the servers, is also captured for analysis.

The main reason why the task offloading paradigm was introduced in MCC is to save mobile devices' resources, such as CPU, RAM, energy consumption, etc., as they are limited compared to a cloud server. That's the reason why the metrics that are mentioned in the previous paragraph are being captured for further analysis.

4 Design Specification

In this section, the design specification that underlies the implementation is detailed with a focus on the overall architecture and used techniques. The design is based on a client-server model, with the client being the mobile application and the services that are deployed to EC2 instances being the servers. The communication between the client and the servers is realized through a WebSocket protocol as WebSocket favours the system in terms of efficient data usage and latency. The design employs three different fault tolerance methods: checkpointing, replication, and a combination of checkpointing and replication during computation offloading. The corresponding fault tolerance mechanism is applied based on the scenario that is being executed. It is critical to emphasise that the necessity for a robust system that can manage connection problems and server-side failures without impacting the execution of the computation-intensive activities drove the design choices.

A high-level architecture diagram is illustrated in Fig 2. As can be seen in the diagram, the setup is made up of a mobile client and two servers, one being the main server and one being the backup server. The client communicates with the servers via WebSocket protocol. Once the process is initiated by the client, the server periodically checks if the client is still connected. If the server understands that the client is disconnected, it stops execution and saves its latest state for later usage. If the main server becomes inaccessible due to server-side issues, the client tries to connect to the backup server and complete its execution in the backup server.

Using this setup, it is possible to simulate three scenarios, being checkpointing, replication, and ensemble technique:

4.1 Checkpointing

As mentioned earlier, the compute-heavy function that is being executed is a for-loop which counts from 0 to the given number. When the checkpointing scenario is being simulated, the server takes a checkpoint at the end of each iteration as long as it ensures the client is still connected. It sends the current iteration number to the client at each iteration and expects the client to respond with the same number. If the client does not send a response at a given time interval, the server assumes that the client has lost connection and saves the latest checkpoint in memory. If the client sends the same request after a short period of time, it recognizes that the same client sent the same request. Instead of starting from 0, the server starts from the latest checkpoint. Therefore, the

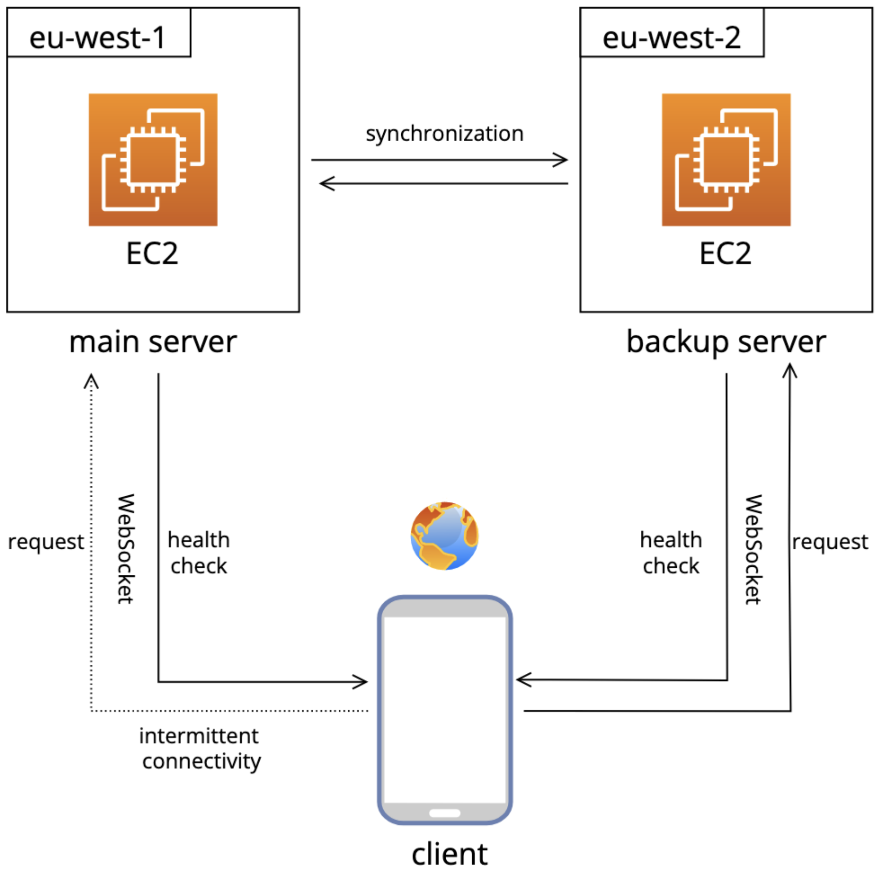

Figure 2: A high-level architecture diagram

server avoids executing from 0 to the latest checkpoint and applies checkpointing as a fault tolerance mechanism. The sequence diagram of the checkpointing scenario is illustrated in Fig 3.

The client's internet disconnectivity is simulated by turning on and off the flight mode of the emulator. Naturally, when the flight mode is turned on, the emulator loses internet connection. When the flight mode is turned off and the internet connection is restored, the client automatically sends the same request to the server again.

4.2 Replication

When the replication scenario is being simulated, the main server intentionally raises an exception to simulate a server-side failure. The client understands that there is an error related to the main server and closes the web socket between itself and the main server. Then, it opens a new web socket connection between itself and the backup server and offloads its task execution to the backup server instead. The data flow diagram of the replication scenario is illustrated in Fig 4.

4.3 Ensemble

This scenario is almost the same as replication with an improvement to it. In the replication scenario, the latest checkpoint before the server failure is not transferred to the

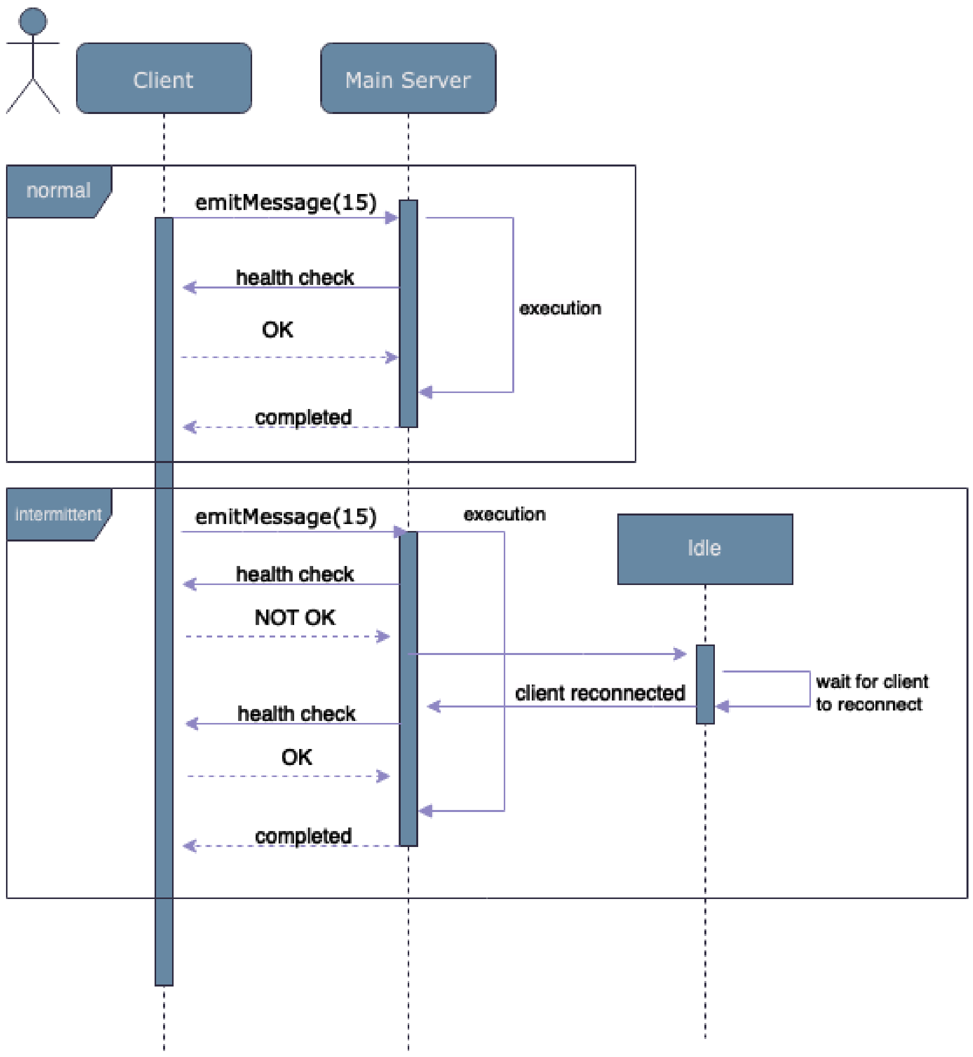

Figure 3: Sequence diagram of checkpointing scenario

backup server. Therefore, the backup server starts execution from the start. Therefore, the portion of execution that is until the failure of the main server is executed twice redundantly.

On the other hand, in the ensemble scenario, the checkpoint is transferred to the backup server before closing the connection. Thus, the backup server can continue from where the main server left off. The sequence diagram of this scenario is given in Fig 5. The only difference between Fig 5 and Fig 4 is the arrow that shows checkpoint synchronization right after the server failure.

4.4 Server Synchronization

The services that fulfill the execution of the computation-heavy task are periodically synchronized between the backup server and the main server to ensure that the backup server can execute the task exactly the same way the main server executes, which can be seen in Fig 2. By doing so, it was possible to transfer every single change that is applied to the main server, removing the redundant step of manually adding the same changes that has been done in the main server to the backup server. This is accomplished by

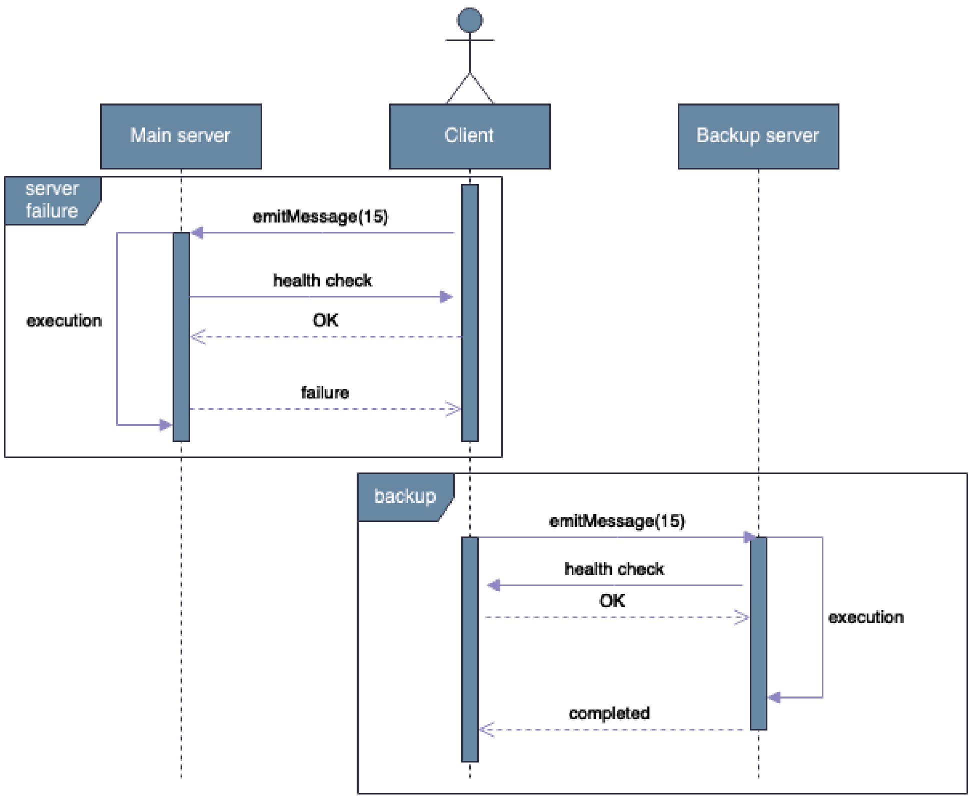

Figure 4: Sequence diagram of replication scenario

running a bash script periodically as a background process in the main server.

Also, specifically in the ensemble method, the data file that contains the latest checkpoint information is synchronised every time a server failure occurs to ensure that the backup server can continue from the latest checkpoint.

The first part mentioned under this subsection can be classified as service replication whereas the second part can be classified as data replication. Therefore, adaptive checkpointing, mentioned in the previous sections, is incorporated into this study.

5 Implementation

This section provides a brief explanation of the implementation of the system from a technical perspective (details are discussed in the configuration manual). Also, information about used tools and technologies as well as the justification of the selections are explained in this section.

The client-side development has been conducted in Android Studio using the Flutter framework with Dart programming language. Android Studio is preferred since it provides developers with Android Virtual Device (AVD) for testing the application and Android Profiler for real-time monitoring of the metrics of the device, such as RAM-CPU usage, network data usage, and power consumption. The Flutter framework is preferred because of the large variety of public libraries available in Flutter. These libraries include but are not limited to keeping track of the mobile device's connectivity status (i.e., connection lost, and connection restored) and establishing web socket connection with the remote servers. Please consult the configuration manual for a detailed list of all the libraries

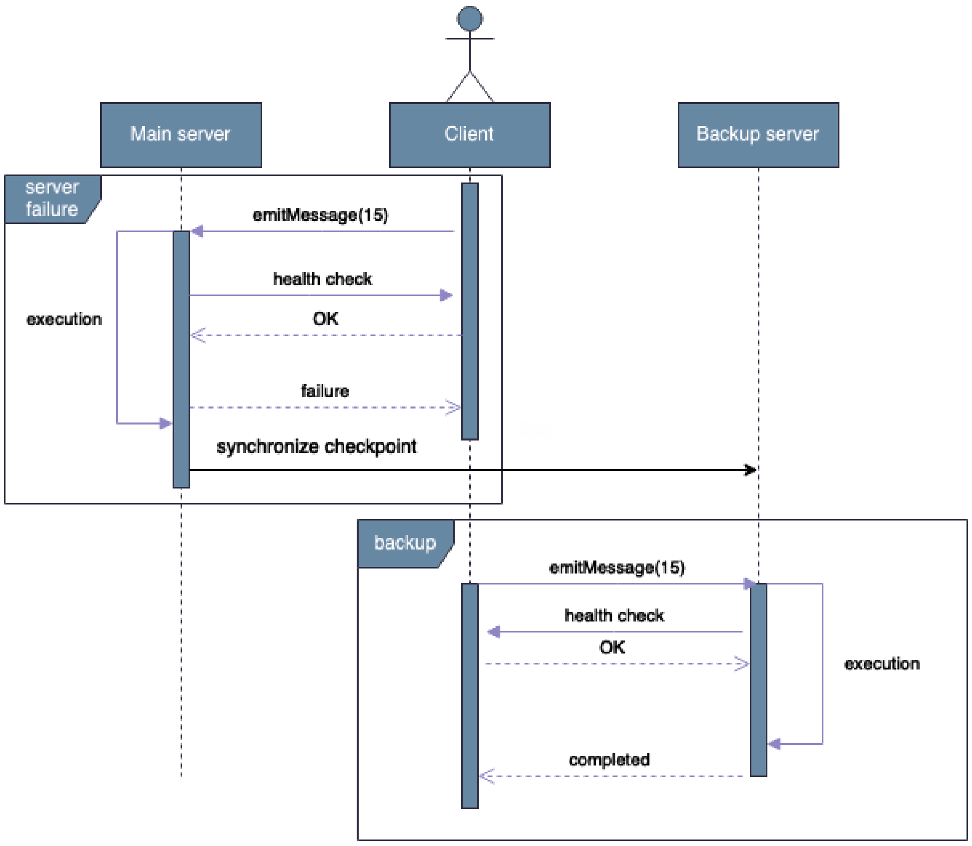

Figure 5: Sequence diagram of ensemble scenario

along with the version numbers.

The server-side development is conducted inside the EC2 instances directly. The servers are implemented with Python's FastAPI framework. Using FastAPI enabled the servers to handle requests asynchronously. For example, at every iteration of the execution, the server sends a health check request to the client and waits for a response from the client before continuing with the next iteration of the execution. This process is heavily dependent on the asynchronous features that FastAPI provides.

The server is listening to the channel that the client opens the connection at. When the client establishes a connection on that channel, the server starts executing the simulated computation-heavy task as described earlier. To understand which scenario is being executed, a constant variable exists with a value of either "checkpoint", "replication", or "combined". The rest of the execution slightly differs with respect to that constant (i.e., if the value of this constant is "replication", the server will intentionally raise an exception at the fifth iteration to simulate server failure).

The EC2 instances are configured to listen for custom TCP connection on (*0.0.0.0*) by adding an inbound rule. By doing that, it was possible to run the servers locally and connect to servers from the client by simply using the public IP address of the servers.

The servers are periodically synchronized to have the same services that handle requests. This process is done by running a bash script as a *nohup* task. *nohup* (short for "no hang-up") is a Linux command that orders the operating system to constantly execute a task even after the terminal session is closed by the developer. These tasks are usually referred to as "*nohup* tasks". Using the *nohup* command, the operating system is commanded to run the bash script constantly with given periods.

The mentioned bash script that ensures synchronization between the main server and backup server follows the following algorithm:

- Log the date of the beginning of data synchronization.
- SSH into the backup server using the pem key.
- Synchronize the source code using the rsync Linux command, which is used to transfer files between two remote hosts, with a couple of exceptions such as the virtual environment files of Python.
- Log the date of the completion of data synchronization.

6 Evaluation

This section outlays the results of the experiments that were detailed in the previous sections. As mentioned earlier, the experiments were run separately for three scenarios. For each scenario, four metrics are captured and visualized with graphs, which are network data usage in kilobytes, CPU usage in percentage, RAM usage in megabytes, and power consumption with a scale from light to heavy. Each graph and result are explained and analysed in detail within the context of this research. Each experiment is run multiple times to ensure that the outputs are reliable. Therefore, the discussed numbers are an average of the outputs of conducted experiments. The results of all three experiments' network data consumption, CPU utilization, and memory utilization graphs are given in Fig 6, Fig 7, and Fig 8 respectively.

CPU utilization and memory utilization graphs are based on the maximum reached value during the experiments. The rest of the values are not included as these values don't make any difference to the outcomes since they maintain a constant value. These constant values are idle value for the emulator, meaning they are not related to the experiments.

6.1 Experiment / Checkpointing scenario

Checkpointing scenario is simulated as mentioned in the previous sections. The entire process was completed in around 22.4 seconds. In Fig 6, the blue line shows the network data usage of the mobile application in kilobytes over time. In total, approximately 4 kilobytes of network data is spent for sending data to the server. It's worth noting that this value is calculated for the data that is transferred from the client to the server, not the other way around. Because the data that is being transferred from the server to the client will be the same in all cases and thus won't affect the results of this research.

As mentioned earlier, this experiment involves an intentional client connectivity issue. The figure clearly shows that this disconnectivity occurred between the 6th and 8th seconds as the data usage went to 0 kilobytes within this interval. The rest of the data flow, which is around 0.2 kilobytes, is the health checks between the server and the client.

As can be seen here, there are two peaks in the graphs, one is at the 1st second, and one is at the 11th second. These peaks occur when the client establishes an initial connection with the server. The first one is naturally the beginning of the experiment. The second one is when the client restores the network connection and reconnects to

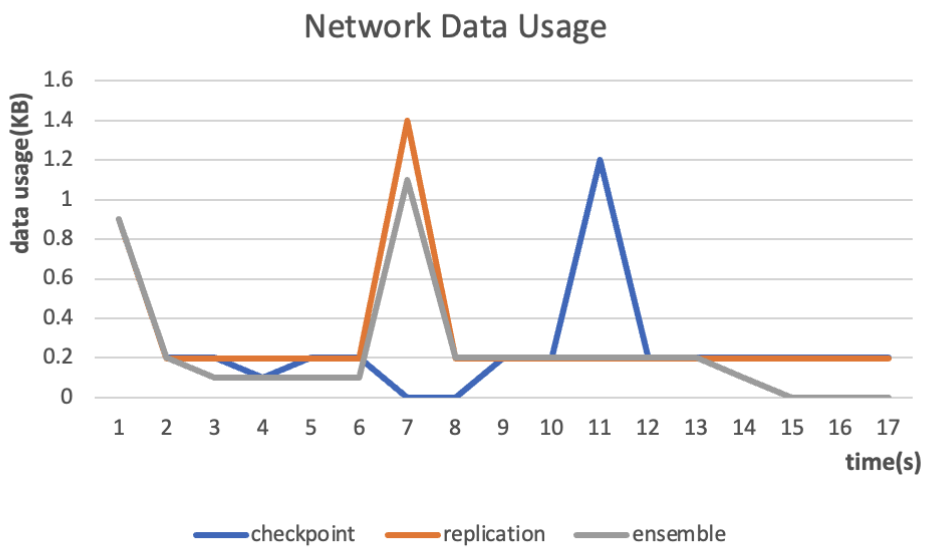

Figure 6: Network data usage graph

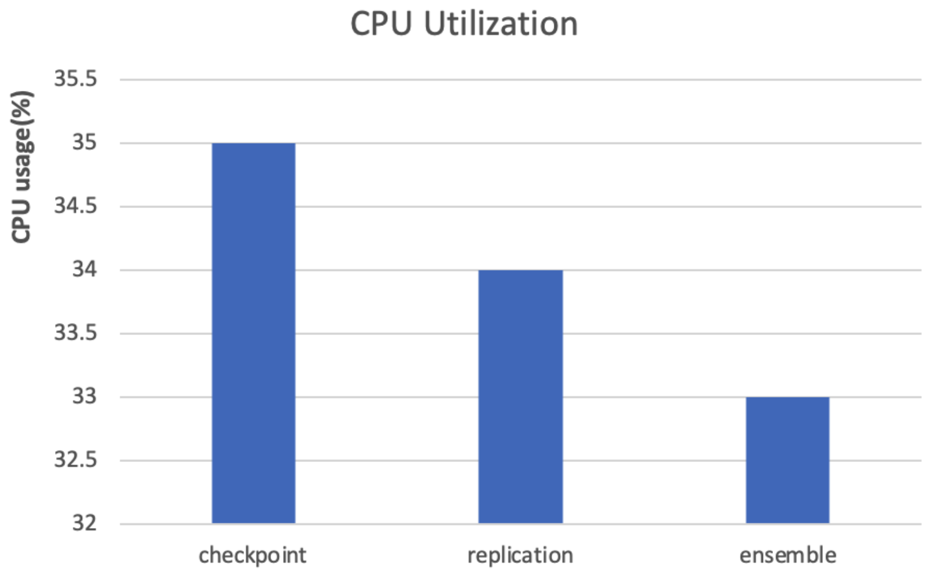

Figure 7: CPU utilization graph

the server. From the values that are visible in the graph, it can be said that within the context of network data usage, the most expensive action is the initial server connection, which is approximately 5 times more expensive than the health check actions.

Fig 8, it can be seen that the peak memory utilization for the checkpointing experiment was 265 megabytes. The peak value was recorded during the startup of the application. The memory utilization was indifferent to the rest of the actions performed throughout the checkpointing experiment.

Fig 9 shows the CPU usage of the mobile device in percentage. The peaks that are visible in this graph are distinguished as "app" and "others", with a green colour and grey colour respectively. App's utilization peaks at a value of approximately 35% at

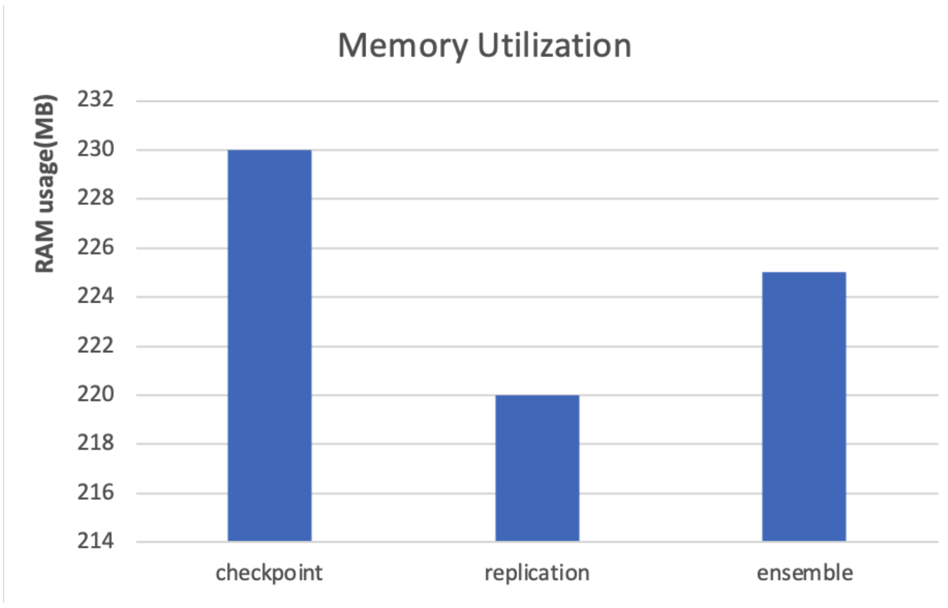

Figure 8: Memory utilization graph

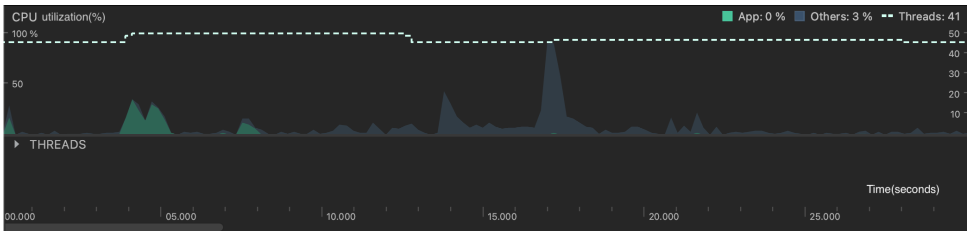

Figure 9: Checkpointing - CPU utilization graph

the beginning of the experiment and continues with a value between 5% and 0% for the rest of the app's life cycle. One aspect worth noting here is the peak of "others" at 13th second and 17th second. The CPU utilization of other tasks peaks to a value of approximately 30% and 95% on the 13th and 17th second respectively. The 13th second is the moment internet disconnectivity is simulated by turning on the flight mode of the emulator and the 17th second is the moment when the emulator restored its internet connection. Therefore, rather than being related to the application itself, this peak is related to the mobile device's internet connectivity. In other words, the peak of other tasks during the intermittent connectivity condition proves that losing and restoring an internet connection is a CPU-intensive task for a mobile device.

6.2 Experiment / Replication scenario

The replication scenario is simulated as mentioned in the previous sections. The end-to-end process took around 23.3 seconds. In Fig 6, the orange line shows the network data usage graph in kilobytes over the time of the replication scenario. In total, approximately 4.4 kilobytes of network data is spent throughout the entire process. It can be seen that at the 7th second, there is a jump in data usage. This is because of the intentional server failure that was simulated in this scenario. Similar to the checkpointing scenario, there are two peaks visible in the graph. The first peak is the initial connection between the client and the server. The second peak is when the client connects to the backup server for the first time. At the 7th second, the client closes its connection with the main server and establishes a new connection with the backup server.

In Fig 8, it is visible that the maximum memory usage was around 220 megabytes. Similar to the checkpointing, this peak occurred during the beginning of the application. The memory utilization was indifferent to the rest of the actions performed throughout the replication experiment. This analysis shows that the memory usage difference between checkpointing and replication is almost negligible.

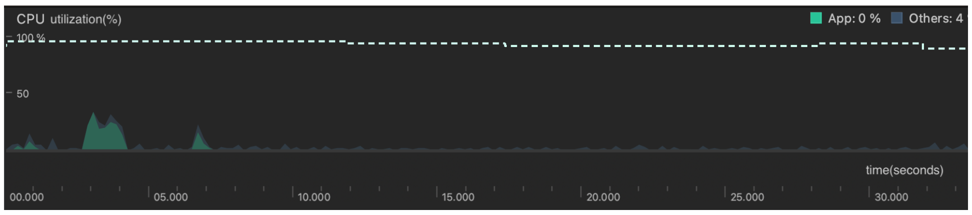

Figure 10: Replication - CPU utilization graph

Fig 10 shows the CPU usage of the mobile device in percentage over time. App's CPU utilization peaks at a value of approximately 34% at the beginning of the experiment and continues with a value between 5% and 0% for the rest of the app's life cycle. Unlike the checkpointing scenario, there is no peak at the other tasks throughout the execution. This proves the earlier hypothesis about the peak of other tasks being related to network connection loss and recovery of the mobile device as there is no intermittent connectivity in the replication scenario.

6.3 Experiment / Ensemble scenario

The ensemble scenario is simulated as described in the previous sections. The whole execution was completed in 20 seconds. In Fig 6, the grey line shows the network data usage graph in kilobytes over the time of the ensemble scenario. Similar to the replication scenario, there is a peak at the 7th second, which shows that the intentional server disconnectivity was simulated at this second. Other than the 7th second, there is another peak at the 1st second which is mutual in all three experiments. This peak occurs when the client tries to establish a connection with the server for the first time.

In Fig 7 and Fig 8, the CPU utilization and memory utilization of the ensemble scenario can be seen respectively. The maximum CPU utilization is 33%, which is not very different than the previous scenarios. The maximum memory utilization is 258 megabytes. During the rest of the performed actions, CPU utilization and memory utilization were indifferent to the application.

6.4 Discussion

Results of all of the scenarios are gathered in Table 2. The best performing method, which is the ensemble method, is highlighted in blue. The power consumption was measured along with the other metrics. The results were indifferent to the experiments.

From the network data point of view, the proposed ensemble technique seems to be cheaper than replication and almost the same as checkpointing. It's quite natural that

Table 2: Experiment results

technique	time	network data	memory	CPU	power
checkpointing	22.4s	4KB	265MB	35%	light
replication	23.3s	4.4KB	270MB	34%	light
ensemble	20s	3.9MB	258MB	33%	light

it's almost the same with checkpointing as these two techniques do almost the same thing, continuing from the latest checkpoint when there is an error. The benefit that the ensemble technique brings is the flexibility of being able to continue execution from another server without losing the latest checkpoint in the case of a server failure.

From a network data point of view, replication seems to be more expensive than checkpointing. This outcome is natural since the execution starts from the beginning when the client connects to the backup server. It can also be inferred that if the cached checkpoints were also synchronized to the backup server and execution could continue from the latest checkpoint before the main server ran into an error, the network data consumption could be reduced since the redundant execution of the iteration before the latest checkpoint would be skipped. This would require involving adaptive replication, being a mixture of service replication and data replication.

Within the context of execution time, the proposed ensemble technique outperforms checkpointing by 2.4 seconds and replication by 3.3 seconds. The ensemble technique favours execution time by neglecting the redundant execution of the tasks from the beginning to the failure occurrence.

For CPU and memory utilization, these two metrics are not very determinant from a performance point of view. The only inference that can be made is from the checkpointing experiment: Restoring network connection after losing it is an expensive task in terms of the mobile device's CPU.

In summary, the key gains of the proposed ensemble technique are in execution time and system reliability. By combining the ability to continue from where the execution was left off of checkpointing and the flexibility to continue execution from another server of replication, the ensemble technique is the best method among all three techniques that have been simulated.

7 Conclusion and Future Work

In conclusion, it is clearly visible that the proposed ensemble method made considerable contributions to mobile devices' performance metrics. The main idea here is to be flexible enough to cover both server failures and client connectivity failures at the same time. By combining the checkpointing method's snapshot-taking and replication method's reliability to connect to another server under a server failure case, the ensemble method outperforms both of them if thought of as standalone. This ensemble technique favoured overall execution time by an average of 2.8 seconds. Moreover, instead of being able to tolerate only one type of error, the ensemble technique can tolerate two types of errors at the same time. Therefore, the overall reliability of the system is increased.

The possible future work may involve strengthening the proposed technique with more fault tolerance techniques. The main idea behind the proposed ensemble technique is to combine the strengths of checkpointing and replication to cover a wider variety of faults.

Thus, following the same idea, more fault tolerance techniques, such as autonomous recovery, task migration, etc., could be incorporated to cover even a wider variety of errors.

References

Abdulhamid, S. M., Latiff, M. S. A., Madni, S. H. H. and Abdullahi, M. (2016). Fault tolerance aware scheduling technique for cloud computing environment using dynamic clustering algorithm, *Neural Computing and Applications* **29**(1): 279–293.
URL: *https://doi.org/10.1007/s00521-016-2448-8*

Akherfi, K., Gerndt, M. and Harroud, H. (2018). Mobile cloud computing for computation offloading: Issues and challenges, *Applied Computing and Informatics* **14**(1): 1–16.
URL: *https://www.sciencedirect.com/science/article/pii/S2210832716300400*

Ali, A. and Iqbal, M. M. (2022). A cost and energy efficient task scheduling technique to offload microservices based applications in mobile cloud computing, *IEEE Access* **10**: 46633–46651.

Almaiah, M. A. and Al-Khasawneh, A. (2020). Investigating the main determinants of mobile cloud computing adoption in university campus, *Education and Information Technologies* **25**(4): 3087–3107.
URL: *https://doi.org/10.1007/s10639-020-10120-8*

Chaâri, R., Cheikhrouhou, O., Koubâa, A., Youssef, H. and Gia, T. N. (2022). Dynamic computation offloading for ground and flying robots: Taxonomy, state of art, and future directions, *Computer Science Review* **45**: 100488.
URL: *https://www.sciencedirect.com/science/article/pii/S1574013722000296*

Fernando, N., Loke, S. W. and Rahayu, W. (2013). Mobile cloud computing: A survey, *Future Generation Computer Systems* **29**(1): 84–106. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X12001318*

Goyal, M. and Saini, P. (2016). A fault-tolerant energy-efficient computational offloading approach with minimal energy and response time in mobile cloud computing, *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pp. 44–49.

Kumari, P. and Kaur, P. (2021). A survey of fault tolerance in cloud computing, *Journal of King Saud University - Computer and Information Sciences* **33**(10): 1159–1176.
URL: *https://www.sciencedirect.com/science/article/pii/S1319157818306438*

Lakhan, A., Ahmad, M., Bilal, M., Jolfaei, A. and Mehmood, R. M. (2021). Mobility aware blockchain enabled offloading and scheduling in vehicular fog cloud computing, *IEEE Transactions on Intelligent Transportation Systems* **22**(7): 4212–4223.

Lakhan, A. and Li, X. (2019). Mobility and fault aware adaptive task offloading in heterogeneous mobile cloud environments, *ICST Transactions on Mobile Communications and Applications* **5**: 159947.

Lee, J. H. and Gil, J. M. (2019). Adaptive fault-tolerant scheduling strategies for mobile cloud computing, *Journal of Supercomputing* **75**: 4472–4488.

Manoj Kumar Malik, Dr. Hitesh Joshi, D. A. S. (2022). Study of scheduling for fault tolerant scientific workflow using task replication in cloud computing/system, *Journal of Pharmaceutical Negative Results* p. 4206–4214.
URL: *https://pnrjournal.com/index.php/home/article/view/9658*

McLean, G., Al-Nabhani, K. and Wilson, A. (2018). Developing a mobile applications customer experience model (mace)- implications for retailers, *Journal of Business Research* **85**: 325–336.
URL: *https://www.sciencedirect.com/science/article/pii/S0148296318300171*

Naeem, T. and Sadik, S. (2014). Optimizing performance and fault tolerance through cloud based adaptive replication for mobile applications, *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pp. 309–314.

Noor, T. H., Zeadally, S., Alfazi, A. and Sheng, Q. Z. (2018). Mobile cloud computing: Challenges and future research directions, *Journal of Network and Computer Applications* **115**: 70–85.
URL: *https://www.sciencedirect.com/science/article/pii/S1084804518301504*

Park, J., Yu, H., Kim, H. and Lee, E. (2014). Dynamic group-based fault tolerance technique for reliable resource management in mobile cloud computing, *Concurrency and Computation: Practice and Experience* **28**(10): 2756–2769.

Pimentel, V. and Nickerson, B. G. (2012). Communicating and displaying real-time data with websocket, *IEEE Internet Computing* **16**(4): 45–53.

Rahmani, A. M., Mohammadi, M., Mohammed, A. H., Karim, S. H. T., Majeed, M. K., Masdari, M. and Hosseinzadeh, M. (2021). Towards data and computation offloading in mobile cloud computing: Taxonomy, overview, and future directions, *Wireless Personal Communications* **119**(1): 147–185.
URL: *https://doi.org/10.1007/s11277-021-08202-y*

Saeik, F., Avgeris, M., Spatharakis, D., Santi, N., Dechouniotis, D., Violos, J., Leivadeas, A., Athanasopoulos, N., Mitton, N. and Papavassiliou, S. (2021). Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions, *Computer Networks* **195**: 108177.
URL: *https://www.sciencedirect.com/science/article/pii/S1389128621002322*

Shahid, M. A., Islam, N., Alam, M. M., Mazliham, M. and Musa, S. (2021). Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment, *Computer Science Review* **40**: 100398.
URL: *https://www.sciencedirect.com/science/article/pii/S1574013721000381*

Shamshirband, S., Fathi, M., Chronopoulos, A. T., Montieri, A., Palumbo, F. and Pescapè, A. (2020). Computational intelligence intrusion detection techniques in mobile cloud computing environments: Review, taxonomy, and open research issues, *Journal of Information Security and Applications* **55**: 102582.
URL: *https://www.sciencedirect.com/science/article/pii/S2214212620307523*

Sowjanya Lakshmi, A., Vani Priya, C. and Gupta, G. (2022). Communication induced checkpointing based fault tolerance mechanism – a review and ciac-ftm framework in iot environment, *2022 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pp. 1–6.

ul Hassan, M., Al-Awady, A. A., Ali, A., Iqbal, M. M., Akram, M., Khan, J. and AbuOdeh, A. A. (2023). An efficient dynamic decision-based task optimization and scheduling approach for microservice-based cost management in mobile cloud computing applications, *Pervasive and Mobile Computing* **92**: 101785.
URL: *https://www.sciencedirect.com/science/article/pii/S1574119223000433*

Zalat, M. S., Darwish, S. M. and Madbouly, M. M. (2022). An adaptive offloading mechanism for mobile cloud computing: A niching genetic algorithm perspective, *IEEE Access* **10**: 76752–76765.

Zhou, B. and Buyya, R. (2017). A group-based fault tolerant mechanism for heterogeneous mobile clouds, *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous 2017, Association for Computing Machinery, New York, NY, USA, p. 373–382.