Fault Tolerant Horizontal Computation Offloading

Droob, Alexander; Morratz, Daniel; Jakobsen, Frederik Langkilde; Carstensen, Jacob; Mathiesen, Magnus; Bohnstedt, Rune; Albano, Michele; Moreschini, Sergio; Taibi, Davide

Published in:
Proceedings - 2023 IEEE International Conference on Edge Computing and Communications, EDGE 2023

DOI (link to publication from Publisher):
10.1109/EDGE60047.2023.00036

Publication date:
2023

Document Version
Accepted author manuscript, peer reviewed version

Fault Tolerant Horizontal Computation Offloading

Alexander Droob, Daniel Morratz, Frederik Langkilde Jakobsen, Jacob Carstensen, Magnus Mathiesen, Rune Bohnstedt, and Michele Albano
Aalborg University, Aalborg, Denmark

Sergio Moreschini
Tampere University
Tampere, Finland

Davide Taibi
University of Oulu, Oulu, Finland
Tampere University, Tampere, Finland

***Abstract*—The broad development and usage of edge devices has highlighted the importance of creating resilient and computationally advanced edge-to-cloud continuum environments. When working with edge devices these desiderata are usually achieved through replication and offloading. This paper reports on the design and implementation of a fault-tolerant service that enables the offloading of jobs from devices with limited computational power. We propose a solution that allows users to upload jobs through a web service, which will be executed on edge nodes within the system. The solution is designed to be fault tolerant and scalable, with no single point of failure as well as the ability to accommodate growth, if the service is expanded. The use of Docker checkpointing on the worker machines ensures that jobs can be resumed in the event of a fault. We provide a mathematical approach to optimize the number of checkpoints that are created along a computation, given that we can forecast the time needed to execute a job. We present experiments that indicate in which scenarios checkpointing benefits job execution. Our experiments shows the benefits of using checkpointing and restore when the completion jobs' time rises compared with the forecast fault rate.**

***Index Terms*—checkpointing, edge nodes, workers, orchestration, replication, totally ordered multicast**

I. Introduction

Many IoT and edge devices deployed in the edge to cloud continuum [1] [2], have limited hardware capabilities such as processing power and memory, and some lack hardware components such as local storage or Graphical Processing Units. To overcome the challenges posed by the lack of computational power and missing hardware, a widespread solution is to outsource computational jobs to other more powerful or more specialized machines. This concept is known as computation offloading [3].

Platforms targeted by the computation offloading can either prioritize cost over reliability (in the case of edge nodes) or be not under the full control of the user in the case of computation power provided by volunteers. One way to add resilience to faults that can interrupt the current job, is to create a checkpoint of the current status of the computation, move it to another device, and use it to continue the computation from the checkpoint in case a fault occurs.

In this work, we aims at developing an efficient and robust offloading solution based on edge nodes, considering possible sources of faults:

1) The worker running on an edge node can suffer a fault
2) The orchestrator that allocates jobs on workers and keeps track of the checkpoints, can suffer faults itself
3) To identify the location (IP address, port, etc) of the components of the architecture, it is either necessary to have a register (e.g.: a Dynamic DNS), or have all architectural components act like clients towards a message broker.

This paper presents a robust-by-design solution[1] and analyzes it in terms of its efficiency in terms of total computation time for the submitted jobs, and energy expenditure.

The rest of this paper is structured as follows: Section II provides background information on the concepts and technologies employed in this work, and discusses related work; Section III-A presents an analysis of the requirements considered for this work; Section III describes the solution we created; Section IV reports on experimental results corroborating our approach; Section V draws conclusions on the topic at hand and proposes future work.

II. Background Information

A. Fault tolerant computation offloading

A computation offloading solution is inherently a distributed system where components interact with each other by passing messages [4].

Computation offloading can either be vertical, for example from a mobile device to the cloud, or horizontal, with the computational job being sent from an edge node to another one. We focus on horizontal computation offloading, which is especially beneficial in cases where high latency can have a critical result in the performance of edge nodes processes [5].

One recurrent issue in computing is making the system fault-tolerant, meaning that the system can keep running as intended in the occasion of partial failure [6].

One of the most popular methods for dealing with fault tolerance is replication, involving creating one or more copies (replicas) of the system and keeping them ready to take over if the original system fails.

Another way to provide fault tolerance is checkpointing and recovery [7]. This involves creating a snapshot of a running application, saving it, and then using the saved snapshot to recover the application and continue it from that point, should a fault occur. With regular checkpoints, it is possible to minimize the time lost in the event of faults.

[1]The code of the solution is released as open source, and it is available at https://github.com/orgs/P7-workrs/repositories

B. Related Work

Multiple works in the last decade have been showing the importance of computation offloading in such an environment.

In [8] Lin et al, provided an overview of the different architectures as well as reviewed related works focused on different key characteristics such as application partitioning, task allocation, and resource management.

Mach et al. in [9] provided a different take on the subject and centered on user-centric use cases in mobile edge computing. Their study examined computation offloading decisions, allocation of computation resources, and mobility management, comparing different works on these subjects.

Mao et al. proposed a dynamic computation offloading posing particular attention in energy harvesting technologies [10], supporting dynamic computation offloading by means of a Lyapunov optimization-based dynamic computation offloading (LODCO) algorithm.

As for horizontal computation offloading, a two-step distributed horizontal architecture for computation offloading was presented in [11]. In this work, horizontal offloading is mostly performed in the fog through directed acyclic task graphs. This results in an optimization of resources at the price of communication latency, which was justified in heavy computationally-required tasks.

Checkpointing for system preservation was adopted by Karhula et al. in [12] where the checkpoint has been used to suspend long-running functions allowing Function as a Service and Serverless applications.

However, compared to previous works, together with taking into account the resource requirements for edge devices, we also add one more level of fault tolerance by performing replication of the orchestrator and the message broker. Moreover, we provide a mathematical definition to compute an optimal checkpointing strategy, and we corroborate our approach by means of experimental evaluation focused on both the execution time of computational jobs and energy consumption.

III. The Proposed Solution

A. Requirements

We identified the following requirements:

[R1] A user must be able to: upload a job to the solution, download results, cancel a job in progress, see current status of a job, see previously completed jobs, see previously canceled jobs, identify themselves by means of a username, to see a graph visualization of orchestrators and workers working on their jobs, see statistics and performance information on their previous jobs

[R2] The application must distinguish between users

[R3] The application must be scalable and fault-tolerant, to the point where there is no single point of failure

[R4] A job must be resumable from a checkpoint

[R5] The service must be secure

Since our proposed solution will be implemented as a prototype only, security is not deemed a priority.

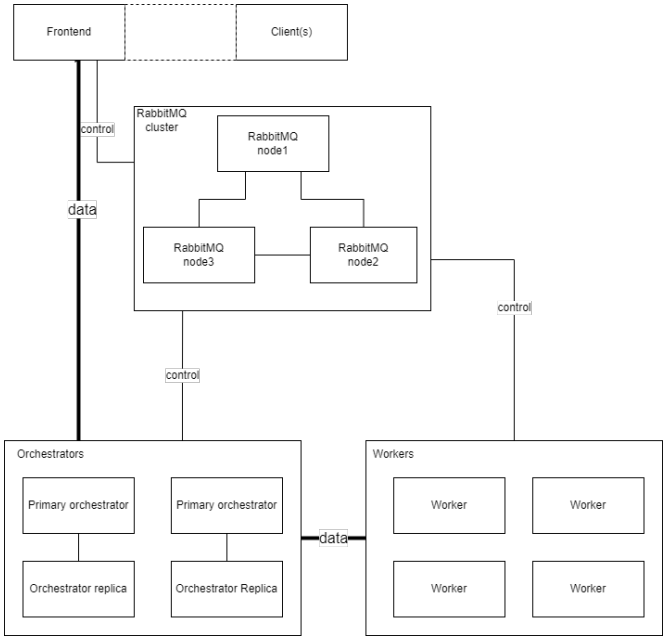


Fig. 1: Overview of proposed architecture

B. System Architecture

The overall solution (Figure 1) was designed with scalability and reliability in mind. The number of orchestrators, clients, and workers can be scaled up, and all architectural components are replicated so there is no single point of failure.

The **client** represents the computer utilized by the final user of the solution. It will either communicate with a server-rendered webserver that will act as frontend, or it will run the frontend in a single page web application.

The **frontend** is stateless and it provides a way for the client to interact with the rest of the solution. The frontend communicates with the orchestrators through the RabbitMQ cluster for control messages, and it can interact directly with the FTP server of an orchestrator for data messages, i.e.: to submit the script for job and to download the job results.

The **orchestrator** is responsible for handling business logic in the solution between the frontend and workers, which connect through RabbitMQ and using the FTP servers exposed by each orchestrator. When requested, the orchestrator will lookup relevant data on the client/worker association in a distributed hash table and respond to the client/worker with their ID in the orchestrator, which orchestrator will serve them will be responsible for their data. This allows the client or worker to initiate a session. When the orchestrator receives a job from a client via FTP the orchestrator creates makes relevant available to an available worker as a job, which will be notified through the RabbitMQ broker and will retrieve related data from the orchestrator's FTP server. When the orchestrator receives a checkpoint via FTP, it will be saved and in the event that a worker suffer from a fault, the job will be resumed from the latest checkpoint. When the orchestrator receives the result of a job from a worker via FTP, it will notify the client who

owns the job via the RabbitMQ broker to provide it with a FTP download link for the result.

The **workers** are responsible for executing the jobs, checkpointing and resuming clients jobs. When a worker receives a job, the worker starts executing it. During execution the worker will periodically create checkpoints and upload them to the orchestrator. The worker will also periodically send a heartbeat to the orchestrator to let it know that it is still operational. When a worker has finished executing a job, the result will be uploaded to the orchestrator and the worker will once again be ready to receive a new job.

The **RabbitMQ cluster** is the message broker. The decision to use RabbitMQ for connecting various devices was made due to its ability to provide Quality-of-Service, which guarantees the delivery of sent messages between the devices. The RabbitMQ broker provides different options for how to communicate through the solution and for this service is decided to use a combination of queues and exchanges in a topic, direct and fanout configuration. By introducing a centralized broker the amount of connections within the solution is kept to only increase linearly when edge nodes are joining the service thus making it scalable but creating other issues such as introducing the possibility of bottleneck and single points of failure. To overcome these issues RabbitMQ can be configured to run as a cluster increasing the throughput and having a failover strategy in case a RabbitMQ node becomes unavailable [13].

C. Replication strategy

Among the desired requirements, an essential *must have* was related to the possibility of resuming a job from a checkpoint. Resuming from a checkpoint results in a solution capable of creating snapshots of a running job periodically. The creation of snapshots would enable the chance of resuming a job from the latest snapshot whenever the edge node executing the job suffers from a fault. We decided to use the Checkpoint/Restore In Userspace (CRIU) [14] of Docker for this aim.

As for the orchestrator, for the sake of ensuring reliability and availability, we decided to have backup orchestrators that can take over for the primary orchestrator if this latter component experiences hardware or software failure. We implemented with passive replication, using a primary replica manager and one or more backup replica managers that can act as the primary in case of a replica fault. With regards to control messages, the primary replica will receive requests from a frontend, relay all requests to the other replica, and acknowledge the requests to the fronted only after the replicas confirm them. With regards to data, i.e. the scripts to be run as jobs and the results from the workers, the primary orchestrators save them into a folder shared with the orchestrator replica, to let the operating system perform the replication.

D. Consistency strategy

To maintain consistency across the primary and the backup orchestrators when they receive novel information such as the presence of a new checkpoint, it is important to make sure the all the orchestrator replicas would reach the same state when targeted by requests. It is therefore necessary to notify the orchestrator on changes of a shared resource, and to enxure that all the changes are applied in the same order. It is therefore important to use a multicasting strategy that provides total ordering, which is a communication procedure where a message is sent to a set of receivers, with all messages being received in the same order.

RabbitMQ provides fanout exchanges which allows for messages targeting the exchange to be received by multiple queues in the order in which the messages were received by the exchange. This means that RabbitMQ fanout provides multicasting with total ordering. Thus RabbitMQ fan out fulfills the solution needs for multicasting.

Here the totally ordered multicast using RabbitMQ as described above is taken advantage of. When an orchestrator wishes to make a change to a shared resource it will notify all known orchestrators and each one will then stop accessing the shared resources and respond that they are ready to receive a change. The orchestrator wishing to make the change will then publish the change to the fan out meaning that all orchestrators, including itself, will receive the change and once all known changes are consumed from an orchestrator queue and has been applied, the orchestrator will release the lock.

An orchestrator can also initiate the locking procedure for a change of the shared resource:

1) A locking request including a Globally Unique Identifier (GUID) is sent to the RabbitMQ multicast.
2) The RabbitMQ broker takes the request and multicasts the request to all orchestrators.
3) When each orchestrator reaches a safe state and it is ready to update the shared resource, it sends an accept locking acknowledgement to RabbitMQ targeting the orchestrator requesting to lock the shared resource.
4) The RabbitMQ broker forwards all the replies to the requesting orchestrator.
5) When the requesting orchestrator has received acknowledgement from all orchestrators, it publishes the change to the RabbitMQ multicast along with the GUID associated with the change.
6) RabbitMQ multicast the change to all orchestrators. When an orchestrator sees the change it validates the change by comparing the included GUID with the GUID from step 1 and performs the change on the locally stored shared resource. It then checks if further changes have been requested, otherwise it unlocks the shared resource and resumes normal operation.

E. RabbitMQ configuration

The solution uses a RabbitMQ broker for communication. RabbitMQ topic exchanges leverage routing keys to direct messages to their appropriate queues, with each component of the architecture consuming messages from its designated queue. The broker contains three exchanges, and all messages targeting the orchestrators are routed through the orchestrator

exchange, messages targeting clients through the client exchange and messages targeting workers through the worker exchange.

a) Client connection flow: (Figure 2a)

First, the client registers for a session by sending a username to the orchestrator exchange with the routing key "clientRegister", and the name of a temporary queue created by the client itself to receive a response.

Then, the orchestrator that consumed the session registration from the client will look up client information using the received username. The orchestrator responds to the client in the temporary response queue including a client ID and the name of the orchestrator that will serve the client. The client saves the received information, discards the temporary queue, creates a client queue and binds it to the client exchange with routingkey "{clientId}", and adds a consumer to this queue that will receive all future messages for the client.

Lastly, the client sends a connection request to the orchestrator that the client was told will serve it by targeting the orchestrator exchange with routing key "{orchestratorname}.clientConnect". Here the client provides its client id. Upon receiving the message the orchestrator will set up a consumer on the queue bound to the orchestrator exchange with routing key "{orchestratorname}.{clientId}", to receive future messages from the client.

b) New job flow: (Figure 2b)

The client sends a message to the orchestrator exchange with the routing key "{orchestratorname}.{clientId}", the header "<type,startNewTask>" and the job name.

The orchestrator then responds acknowledging the newly created job with a link to a folder in the FTP server running on the orchestrator. The client then uploads the script for the job on the orchestrator's FTP server, and then sends a "<type.taskUploadCompleted>" message to the orchestrator.

c) Worker connection flow: (Figure 2c)

The first message from worker to orchestrator provides a worker id rather than a username when registering for a session, and creates a temporary queue to receive a response. The very first time a worker connects it will provide an empty workerId and the orchestrator will create a new id for the new worker. If the worker has connected before it will have already saved the id on its edge node and will provide this when registering. When responding to the session registration, the orchestrator will respond to the temporary queue with the created or provided worker id and the name of the orchestrator that should serve the worker and the worker creates a consumer on the queue bound to the worker exchange with routing key "{workerId}". Lastly, the worker sends a connection request to the orchestrator that it will server it, and the orchestrator creates a consumer on the queue bound to the orchestrator exchange with routing key "{orchestratorname}.{workerId}".

F. Optimal frequency of checkpointing

In case both the faults frequency and the total time to perform a computation can be forecast, it is possible to compute the optimal frequency for the checkpoints to minimize the total execution time of the job. This section will use the following definitions:

Listing 1: Optimal number of checkpoints, given that T and C can be forecast

```
int optimal_checkpoints_numer(mu, T, C):
    bool in_progress = true;
    int best_N = 0;
    double best_time = predict_time(mu, T);
    do {
        N = N + 1;
        double exec_time = N *
            predict_time(mu, T / N) + (N-1) * C;
        if (exec_time > best_time) {
            in_progress = false;
            N --;
        } else {
            best_time = exec_time;
        }
    }
    return(N);
}

double predict_time(double mu, double T) {
    return (Math.Exp(mu * T) - 1) / mu;
}
```

- T = Total time to complete a job
- μ = Probability of fault in the unit time
- $p(t)$ = Probability density for a fault
- Execution time per part, with checkpointing
- n = Time between checkpoints
- $overhead$ = Cost of checkpointing

We model the fault's distribution as a Poissonian:

$$p(t) = \mu e^{-\mu t}$$

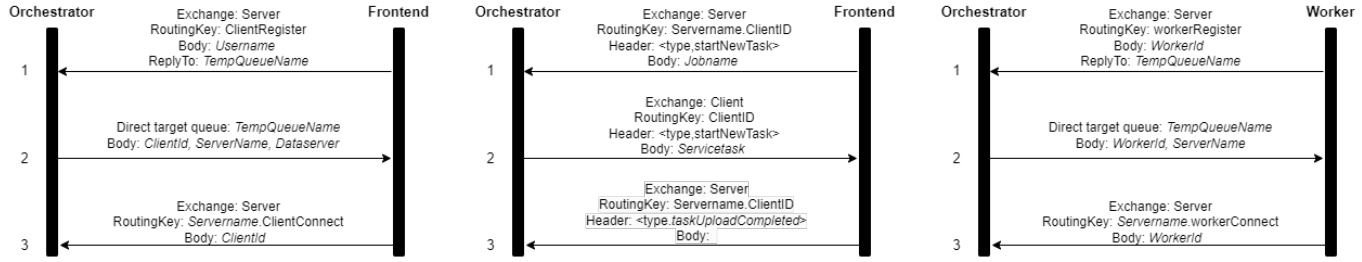
The time to complete a job (see the extended version of the paper [15]), given that faults can occur and they lead to restarting the job, is given by:

$$E_x(\mu, T) = \frac{e^{\mu T} - 1}{\mu}$$

When checkpointing is part of the picture, the process is essentially split into a set of $N \in \{1, ...\}$ processes of length T/N by making use of $N - 1$ checkpoints, at the cost of an overhead of size C for each checkpoint, thus the total execution time becomes:

$$NE_x\left(\mu, \frac{T}{N}\right) + (N-1)\ C \tag{1}$$

Equation 1 is convex (see [15] for its proof), thus it is possible to find the optimal number of checkpoints to be used by means the algorithm reported in Listing 1).



(a) Overview of the client connection flow (b) Overview of the job uploading flow (c) Overview of the worker connection flow

Fig. 2: Examples of communication between frontend, orchestrator and worker

G. Frontend Implementation

IV. Experiments

This section describes the experiments performed over the solution, and their results.

A. Experimental Deployment

The prototype we created runs on a Local Area Network (LAN) containing one personal computer and 8 edge nodes.

The edge nodes are Raspberry Pis 4 with 1 GB Ram and 16 GB of removable SD storage, running Ubuntu Server 20.04.5 LTS (64-bit). The containers are using Docker Engine 20.17.

Six of the edge nodes run one worker each. One more edge node runs the primary orchestrator. The last edge node runs the orchestrator replica and the RabbitMQ broker. Since the experiments focus on corroborating the formulas from subsection III-F and the fault-tolerant orchestrator and workers deployment, we did not set up a clustered RabbitMQ broker, but from its specifics, it appears that it would have not impacted the message bandwidth, nor it would have been part of the trade-off that we are evaluating. The replication of the checkpoint files over the secondary orchestrator is performed by saving the files on a folder shared via the SMB protocol between the orchestrators, thus it happens asynchronously with respect to the rest of the checkpointing functions and it does not impact the performance of the solution.

We have power meters in place to measure the energy spent by each edge node. However, in the experimental results, we will show only the total energy spent by the two orchestrators and two workers, one executing the job and the other one ready to take over if any fault occurs, since the other edge nodes were not involved in the experiments.

A desktop computer (whose energy consumption we did not measure) runs the client and the frontend web server. Initally, the idea was to create a Docker image with the Python interpreter, its libraries and the script related to a job every time a job is submitted. However, it was discovered that building the image in the orchestrator would introduce unnecessary overhead during job startup, since all images were identical except for the Python script to be executed. Thus, we created one base image containing the Python interpreter and some of its most useful libraries, and we pre-installed it on all the edge nodes. The worker would then download the Python script for the particular job to be executed, it would run the base image, and inject into it the Python script. The job startup time got much smaller, since the download time for the Python script is much lower than downloading the full Docker image.

B. Checkpoint time penalty

The first question we aim to answer regards the overhead incurred by periodically creating checkpoints of the jobs and uploading them to the primary orchestrator. To this aim, we ran a series of jobs, each of them having a completion time of 300 seconds, and we set a very low μ, meaning that we expected to have no faults during the job execution.

The first set of jobs were executed without doing any checkpoint, then more jobs were executed performing a checkpoint every 18.75 seconds of job execution (meaning that we stopped the checkpoint timer while performing the checkpoint itself), thus performing a total of 15 checkpoints.

Figure 3 shows the time to complete the job on the x axis, and the energy spent on the y axis. The results hint that the cost of checkpointing is approximately 90 seconds in total, i.e.: each checkpoint causes a delay of 6 seconds. When computing Eq. 1 in the rest of this section, we will consider the cost of checkpointing as 6 seconds.

C. Fault Time penalty

To assess if checkpointing is required given the experiments' parameters, we did not perform any checkpointing with different μ. Figure 4 shows that a low μ is perfectly compatible with not using checkpointing, while a high $\mu = 0.131$ leads to a very long execution time.

To understand how often to perform a checkpoint, and to corroborate the formulas and algorithm from Section III-F, we set a relatively high $\mu = 0.003$. We performed experiments with no checkpoint, with 15 checkpoints, and with 5 checkpoints (one checkpoint every 50 seconds), as suggested by the algorithm in Listing 1. Figure 5 shows that the best results correspond to a checkpoint every $50s$, corroborating our formulas.

V. Conclusion

The scope of this work was to create a robust distributed system for computation offloading. By focusing on creating a solution with high fault tolerance (see the requirements in

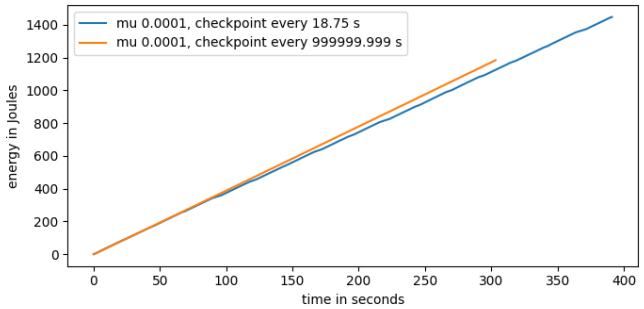


Fig. 3: Execution time $300s$, no faults, comparison between no checkpoints and 15 checkpoints.

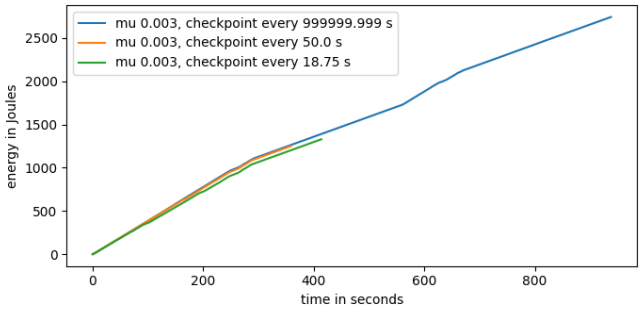


Fig. 5: Execution time $300s$, frequent faults, different checkpointing frequencies.

subsection III-A) every architectural component was designed with the goal of eliminating single points of failure while remaining scalable. For the sake of focusing on the problem at hand, the current prototype had to sacrifice other characteristics such as system security.

In section IV the cost-benefit of checkpointing was investigated and the expected total execution time for jobs of varying lengths both with and without checkpointing with different fault rates were compared. The results confirm experimentally that checkpointing is more useful when the jobs' completion time gets larger with respect to the expected fault rate. A limitation of the study is that in the real world, the fault rate might be hard to forecast and dependent on the environmental conditions the workers experience, and the completion time for a job can be even harder to forecast.

On the positive side, the experiments were performed considering that both workers and orchestrators were run on the same kind of edge device. In a real deployment, the orchestrator would be deployed in a more energy-saving device (e.g.: Raspberry pi), the node devices running workers would be more energy hungry (e.g.: Jetson Xavier), and the ratio worker edge nodes / orchestrator edge nodes would be much higher than in the current work.

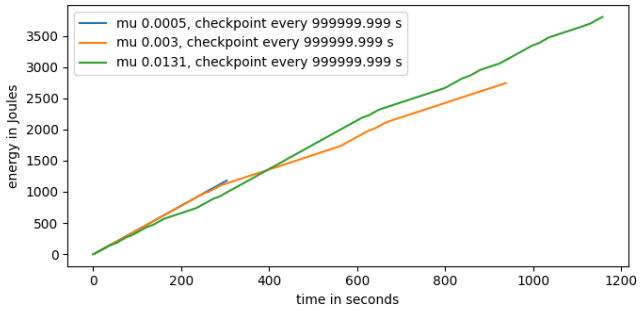


Fig. 4: Execution time $300s$, different faults frequencies, no checkpoints.

ACKNOWLEDGMENTS

This work was partially supported by the S4OS Villum Investigator Grant (37819) from Villum Fonden, by Industry X and 6GSoft projects funded by Business Finland.

REFERENCES

[1] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, and D. Taibi, "Cloud continuum: The definition," *IEEE Access*, vol. 10, pp. 131 876–131 886, 2022.

[2] S. Moreschini, F. Pecorelli, X. Li, S. Naz, M. Albano, D. Hästbacka, and D. Taibi, "Cognitive cloud: The definition," in *Int. Conf. on Distributed Computing and Artificial Intelligence*, 2023.

[3] K. Kumar, J. Liu, Y. Lu, and B. K. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Networks Appl.*, vol. 18, no. 1, pp. 129–140, 2013.

[4] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan, "Toward computation offloading in edge computing: A survey," *IEEE Access*, vol. 7, pp. 131 543–131 558, 2019.

[5] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, 2021.

[6] A. Sari and M. Akkaya, "Fault tolerance mechanisms in distributed systems," *International Journal of Communications, Network and System Sciences*, vol. 8, pp. 471–482, 12 2015.

[7] C.-H. Huang and C.-R. Lee, "Enhancing the availability of docker swarm using checkpoint-and-restore," in *International Symposium on Pervasive Systems, Algorithms and Networks*, 2017.

[8] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, 2019.

[9] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[10] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, 2016.

[11] P. K. Deb, S. Misra, and A. Mukherjee, "Latency-aware horizontal computation offloading for parallel processing in fog-enabled iot," *IEEE Systems Journal*, vol. 16, no. 2, pp. 2537–2544, 2022.

[12] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of iot edge functions," in *Edge Systems, Analytics and Networking*, 2019.

[13] RabbitMQ, "Reliability guide - rabbitmq," 2022. [Online]. Available: https://www.rabbitmq.com/reliability.html#clustering

[14] Docker, "Docker checkpoint," 2022. [Online]. Available: https://docs.docker.com/engine/reference/commandline/checkpoint/

[15] A. Droob, D. Morratz, F. L. Jakobsen, J. Carstensen, M. Mathiesen, R. Bohnstedt, M. Albano, S. Moreschini, and D. Taibi, "Workrs: Fault tolerant horizontal computation offloading," 2023, available online at https://arxiv.org/abs/2305.18219v1.