Workrs: Fault Tolerant Horizontal Computation Offloading

Alexander Droob, Daniel Morratz,
Frederik Langkilde Jakobsen,
Jacob Carstensen, Magnus Mathiesen,
Rune Bohnstedt, and Michele Albano
Department of Computer Science, Aalborg University
Selma Lagerløfs Vej 300, 9220 Aalborg, Denmark

Sergio Moreschini
Tampere University
Tampere, Finland
Email: sergio.moreschini@tuni.fi

Davide Taibi
University of Oulu Oulu. Finland
Tampere University. Tampere, Finland
Email: davide.taibi@oulu.fi

***Abstract*—The broad development and usage of edge devices has highlighted the importance of creating resilient and computationally advanced environments. When working with edge devices these desiderata are usually achieved through replication and offloading. This paper reports on the design and implementation of Workrs, 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. The results achieved are based on a working prototype which shows clear benefits of using checkpointing and restore when the completion jobs' time rises compared with the forecast fault rate. The code of Workrs is released as open source, and it is available at https://github.com/orgs/P7-workrs/repositories. This paper is an extended version of [1].**

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

I. Introduction

In the past decade, more and more devices such as kitchen appliances, cars and even toilets have become part of what is called the Internet of Things (IoT). The Internet Of Things describes a network of physical objects connected to the internet, which makes it possible to collect and exchange data with other systems and devices [2].

In 2022 more than 14 billion devices were connected to the IoT and this number is expected to reach over 27 billion by 2025 [3]. In fact, as time goes by, there is more access to low-cost, low-power sensors making IoT technology more available. Even with this advance in technology, IoT still comes with some challenges. Many IoT devices have limited hardware capabilities such as processing power and memory, and some devices are lacking some components like local storage or Graphical Processing Units, moreover some IoT devices have custom hardware, which makes application development tedious and can limit the availability of ready-to-use software libraries.

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 machine. This concept is known as computation offloading [4], and its benefits comprise improved application performance and energy saving for battery-driven devices.

We consider that the platform targeted by the computation offloading process 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. To add resilience to faults that can interrupt the current job, it can be useful 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.

This work aims at creating an efficient and robust offloading solution based on edge nodes, by taking into account possible sources of faults:

1) The worker running on an edge node can suffer a fault
2) The orchestrator that allocate 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. Both cases represent the introduction of one more architectural component that can suffer a fault

In our proposed solution, called Workrs, we adopt the utilization of a RabbitMQ message broker [5] in a clustered configuration [6] to take care of source of fault 3 from the list just presented. We replicate the orchestrator to protect our architecture against their faults, and use a totally ordered multicast, enabled by the clustered RabbitMQ Broker, to perform passive replication between them, to solve source of fault 2. Each worker periodically creates a checkpoint, and it sends the checkpoint's information (data dump, timestamp, job name, worker name) to an orchestrator to solve source of fault 1.

This paper presents the Workrs solution, which is robust by construction, and it analyzes it in terms of its efficiency. We study its performance in terms of total computation time for the submitted jobs, and energy expenditure. We consider that the baseline is a job run on a single worker without making use of checkpoints. If the worker fails frequently with respect to the length of the job, the total computation time can grow dramatically. Thus, making use of checkpoints can speed up the execution of a job. Regarding energy, consumption is proportional to the total computation time, plus an overhead related to the checkpointing procedure (which is not necessary if the architecture does not consider checkpointing).

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 presents a MoSCoW analysis of the requirements considered for this work; Section IV describes the solution we created; Section V reports on experimental results corroborating our approach; Section VI draws conclusions on the topic at hand and proposes future work.

II. Background Information

This section presents and analyzes some of the fundamental ideas behind computation offloading, and explores the technologies that will be used in this work.

A. Fault tolerant computation offloading

A computation offloading solution is inherently a distributed system where components interact with each other by passing messages [7]. In fact, even though there are benefits to computation offloading, it introduces multiple factors that have to be taken into account such as scalability, fault tolerance [8], Quality of Service, deadlocks, communication capacity [9] and robustness [10]. Moreover, introducing communication between devices over the internet can lead to high latency, if the required internet connection is slow or when transferring large amounts of data. Finally, security is another challenge that requires attention, since information is sent to a service outside of the user's control [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. This work focuses 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 [11].

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 [12]. In particular, fault tolerance comprises the properties of *Reliability*, meaning that the system keeps delivering what it is supposed to even when one or more components break down, and *Availability*, which involves that the system is ready to provide its functionalities even when a fault occurs.

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 [13]. This involves to create a snapshot of a running application, save it and then use the saved snapshot to recover the application and continue it from that point, should a fault occur. If checkpoints are taken regularly, checkpointing and recovery can be used to minimize the time loss in the event of faults.

B. Technologies

The goal of this section is to provide a concise introduction to the technologies and software used in this work.

1) Docker: To standardise the job uploaded from the clients to the workers, we will utilise the Docker framework [14]. This framework allows to deploy a container image that wraps up code and dependencies and isolates the application thus making it possible to run the application reliably as a self-contained unit.

An interesting feature of Docker can be leveraged to create a fault tolerant solution that can create checkpoints of the jobs. In fact, there is an experimental feature of Docker that makes use of an external tool called Checkpoint/Restore In Userspace (CRIU) [15]. It works by freezing a running container and saving the state as a collection of files to disk thus making it possible to restore the container from the point it was frozen at.

2) Storage of checkpoints' information: Based on the requirements that a user should be able to upload jobs, see the status of a job in progress and download the result, the orchestrator needs to be able to manage and store data. We decided to make use of a SQLite database [16], which allows to embed the database itself in the host application. SQLite appears to be faster compared to both a traditional database, and to applications that read and writ files using ad hoc functionalities [16]. We aim at having a SQLite databases on the orchestrators, and having them replicated over the multiple replicas of these architectural components.

3) Message broker: To implement communication between the components of our architecture, we decided to make use of a message broker, since it provides a number of advantages [17]. RabbitMQ is an open-source message broker software that uses a producer/consumer model allowing producers to send messages to consumers. RabbitMQ brokers work by having producers send messages through exchanges that are responsible for routing these messages to various queues [5]. The queues act as pools where messages are stored. RabbitMQ uses acknowledgement-based message retention meaning that messages are retained by the queues and are only removed when the message has been sent to a consumer and the broker has received acknowledgement from the consumer [18]. RabbitMQ brokers can also be set up in a cluster configuration, where a set of RabbitMQ instances share users, exchanges, queues, runtime parameters etc. This configuration ensures failovers in case a RabbitMQ node becomes unavailable [6].

C. Related Work

In a continuum environment characterized by devices with different nature and requirements, computation offloading plays a crucial role in ensuring efficient and effective use of resources [19]. Multiple works in the last decade have been showing the importance of computation offloading in such environment.

In [20] Lin et al, performed a survey analyzing the first evolution of the trend of computation offloading towards edge computing. The work 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 [21] provided a different take on the subject, and centered on user-centric use cases in mobile edge computing. Their study examined computation offloading decision, 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 [22]. The technique developed in the work supported dynamic computation offloading by means of a Lyapunov optimization-based dynamic computation offloading (LODCO) algorithm, which is characterized by low complexity, no prior information required, and takes into account the battery limitations of the devices.

For what concerns specifically horizontal computation offloading, a two-step distributed horizontal architecture for computation offloading was presented in [23]. In the work the 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.

A solution based on checkpointing for system preservation was presented by Karhula et al. in [24]. In this work, the checkpoint has been used to suspend long-running functions allowing Function as a Service and Serverless applications. Specifically, the checkpoint and restore approach has been used together with Docker to preserve the function states of the system enabling long-running functions.

However, compared to all previously described works, together with taking into account the resource requirements for edge devices, we also aim at adding one more level of fault tolerance in the architecture by performing replication of the orchestrator and the message broker as well. Such replication ensures that our whole solution is fault tolerant. Moreover, we provide a mathematical formula to compute an optimal checkpointing strategy, and we corroborate our approach by means of experimental evaluation focused on both jobs execution time of computational jobs and energy consumption.

III. Requirements

Given the problems introduced in Section-I, we define the Workrs solution in terms of its MoSCoW requirements [25]. The list of requirements is as follows:

[R1] A user must be able to upload a job to the solution.
[R2] A user must be able to download the result.
[R3] A user must be able to cancel a current job in progress.
[R4] A user must be able to see the current status of a job.
[R5] A user must be able to see previously completed jobs.
[R6] A user must be able to see previously canceled jobs.
[R7] A user must be able to identify themselves by providing a username
[R8] The application should be able to distinguish between users.
[R9] The application must be fault tolerant to the point where there is no single point of failure.
[R10] The application must be scalable.
[R11] A job must be resumable from a checkpoint
[R12] A user must be able to see a graph visualization of the orchestrators and workers working on their jobs
[R13] A user must be able to see statistics and performance information on their previous jobs
[R14] The service could be secure

Following the MoSCoW analysis, requirements **R1**, **R2**, **R7**, **R8** and **R11** have been identified as *must have*. Requirements **R9** and **R10** are *should have*. Finally, requirements **R3**, **R4**, **R5**, **R6**, **R11** and **R12** are *could have*. Since our proposed solution will be implemented as a prototype only, security is not deemed as a priority. Given that security is a complex topic and can require a lot of effort on the design of a solution, requirement **R14** was put into the won't have category.

IV. The Workrs Solution

This section describes the proposed solution to the issues and requirements defined so far.

A. System Architecture

The architecture of our proposed computation offloading solution is illustrated in Figure 1. The overall solution was designed with scalability and reliability in mind. In fact, the solution is designed so that the amount of orchestrators, clients and workers can be scaled up, and all architectural components are replicated so that there is no single point of failure.

The **client** represents the computer utilized by the final user of the solution. The client will either communicate with a server-rendered webserver that will act as frontend, or it will itself run the frontend in a single page web application [26].

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. Based on the requirements in section III, the frontend exposes both an API (server-rendered web page), and a graphical user interface (single page web application) to provide the following functionalities to a user:

- Identification with a username
- Retrieve information about jobs uploaded by the user
- Upload a new job to the solution

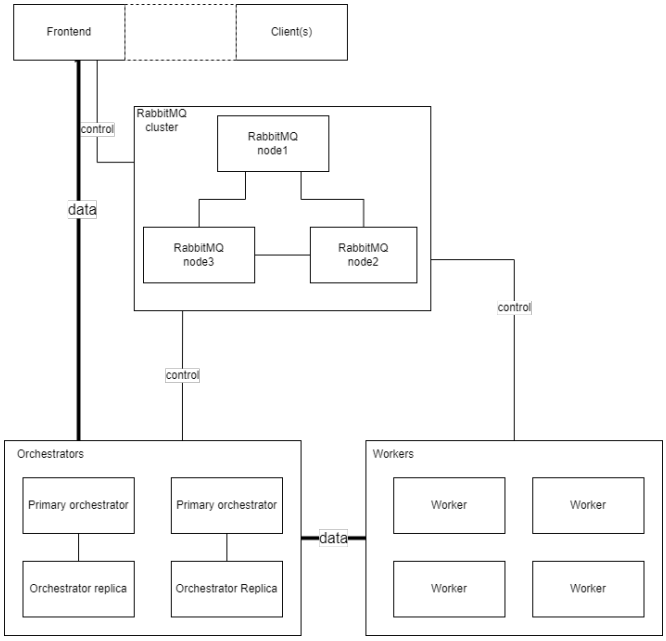


Fig. 1: Overview of proposed architecture

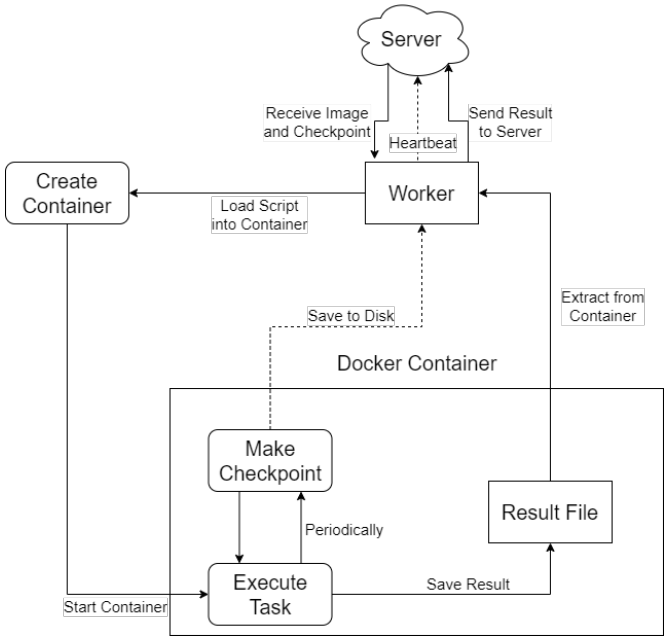


Fig. 2: Flow diagram of a worker

- Download the result of a completed job
- Cancel a job in progress
- Graph visualisation of components in use

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 execution flow of worker can be seen on Figure 2.

The **RabbitMQ cluster** is the message broker of the solution. 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 [27], direct [28] and fanout [27] configuration. By introducing a centralised broker the amount of connections within the solution is kept to only increase linearly when edge node 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 [6].

B. 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) [15] of Docker to this aim.

With regards to the orchestrator, for 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 decided to implement this by means of passive replication, which is implemented by having a primary replica manager and one or more backup replica managers that can act as the primary

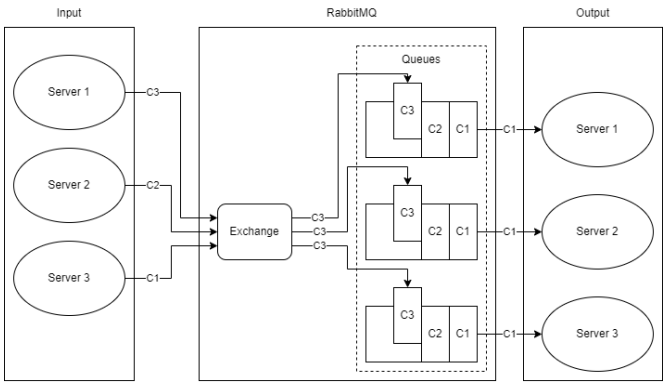


Fig. 3: Example of multicast communication using RabbitMQ fanout exchanges

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 saves them into a folder shared with the orchestrator replica, to let the operating system perform the replication.

C. 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. An example of the chosen multicast strategy is illustrated in Figure 3 where three orchestrators push messages C1, C2 and C3 at the input side. The three messages arrive at the RabbitMQ exchange in the following order C1, C2 and C3 which is now replicated to all queues and are consumed by each orchestrator on the output side in the order the messages arrived to the exchange.

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

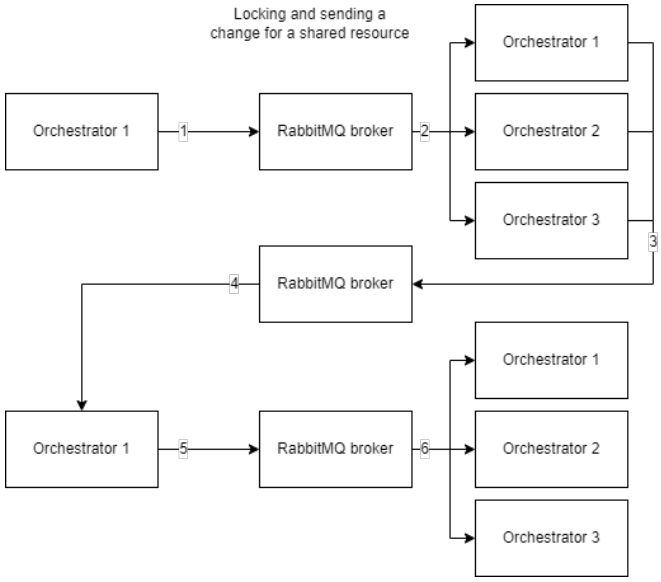


Fig. 4: Illustration of a change procedure of a shared resource

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.

Figure 4 shows how an orchestrator initiates 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.

D. RabbitMQ configuration

The solution uses a RabbitMQ broker for communication, whose configuration is reported on Figure 5. 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.

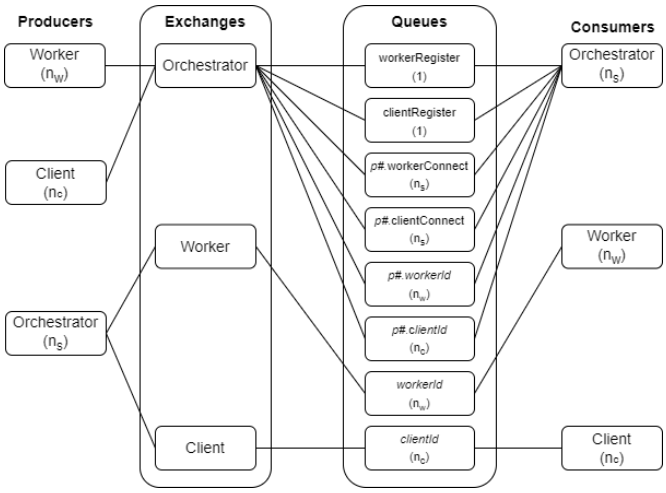


Fig. 5: Overview of exchanges, queues and consumers where n_s is the number of orchestrators in the solution, n_w is the number of workers, $p\#$ is the orchestrator number and n_c is the number of clients

a) Client connection flow: Figure 6a shows the communication flow of the client that connects to the solution through its frontend. First the client registers for a session by sending a username to the orchestrator exchange with routing key "clientRegister". Along with the message, the client sends the name of a temporary queue that was created by the client to receive a response.

After this the orchestrator that happened to consume the session registration from the client will look up client information using the received username. The orchestrator responds to the client by directly targeting the temporary response queue. The response contains 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 6a shows the communication flow when a client communicates with an orchestrator requesting to create a new job. 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, comprising a link to a folder in the FTP server running on the selected 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: The communication flow of a worker connecting to the service can be seen on Figure 6c.

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}".

E. 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:

- 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:

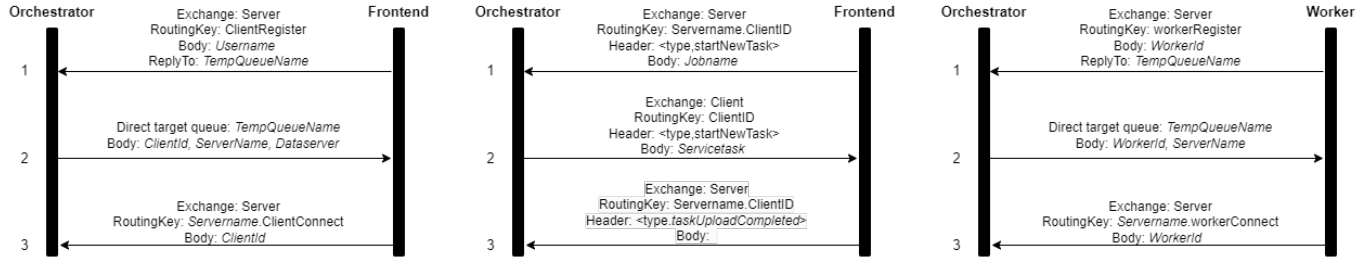
$$p(t) = \mu e^{-\mu t}$$

The time to complete a job (see Appendix A), 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:

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



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

Fig. 6: Examples of communication between frontend, orchestrator and worker

Equation 1 is convex (see Appendix A), thus it is possible to find the optimal number of checkpoints to be used by means the algorithm reported in Listing 1).

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;
}
```

F. Frontend Implementation

We chose to implement the **frontend** in the .NET framework Blazor. Blazor allows for server-side rendering, making it possible to send events from the browser to the orchestrators through it [29]. Additionally, Blazor is component-based, written in C# and it can create interactive user interfaces. Moreover, the C# libraries to interact with a RabbitMQ server proved to work "as they were", while other libraries proved to have issues such as incompatibilities with clustered RabbitMQ brokers.

V. 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. A logical representation of this setup is illustrated on Figure 7, and the physical Raspberry pi cluster we built is shown in Figure 8.

The edge nodes are Raspberry Pis 4 with 1 GB Ram and 16 GB of removable SD storage for the main drive. The operating system flashed to the Raspberry Pi is Ubuntu Server 20.04.5 LTS (64-bit). The containers are run 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 IV-E 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 [6]. The replication of the checkpoint files over the secondary orchestrator is performed by saving the files on a folder shared via the SMB protocol [30] 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

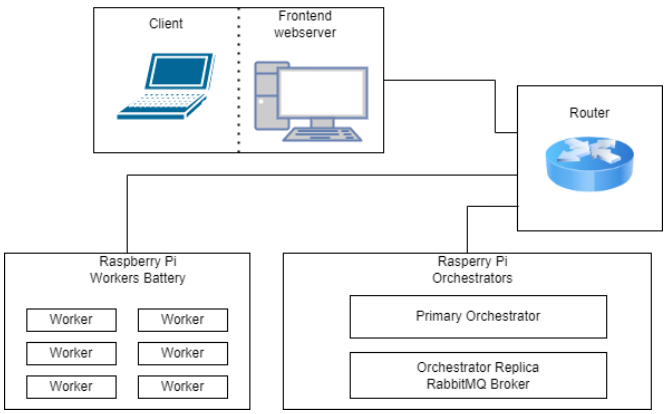


Fig. 7: Logical representation of the experimental setup

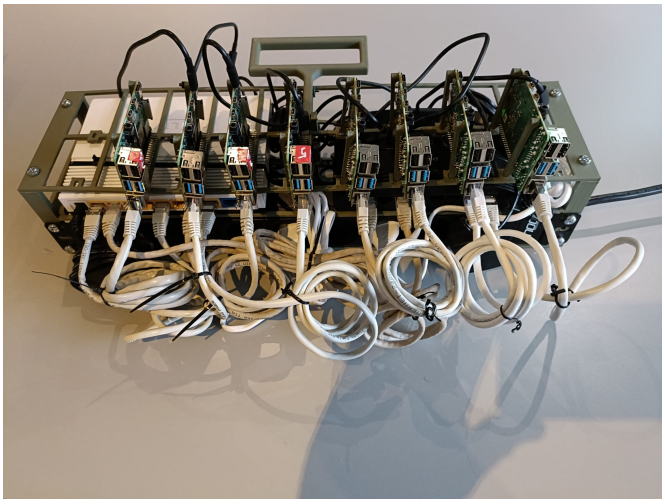


Fig. 8: Raspberry pi cluster used in the experiments

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 9 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.

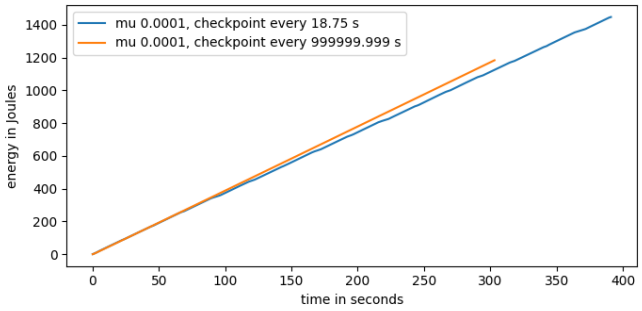


Fig. 9: Execution time $300s$, no faults, comparison between no checkpoints and 15 checkpoints.

C. Fault Time penalty

To assess if checkpointing is required given the experiments' parameters, we did not perform any checkpointing with different μ. Figure 10 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 IV-E, 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 11 shows that the best results correspond to a checkpoint every $50s$, corroborating our formulas.

VI. Conclusion and Future Work

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

In section V the cost-benefit of checkpointing was investigated and the expected total execution time for jobs of varying

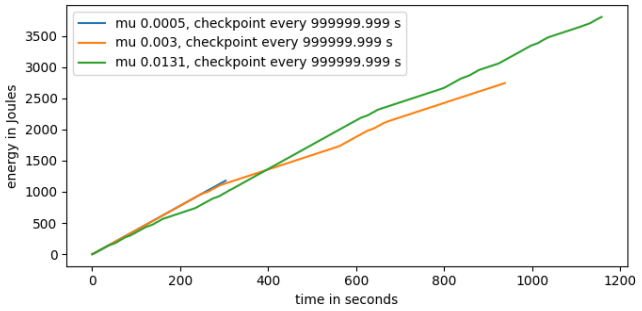


Fig. 10: Execution time $300s$, different faults frequencies, no checkpoints.

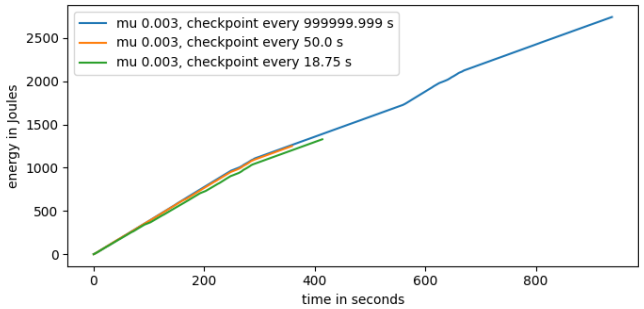


Fig. 11: Execution time $300s$, frequent faults, different checkpointing frequencies.

length 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.

A. Future Work

Future work was planned to improve the solution on different aspects.

a) API: As this solution is aimed towards users with a software background, it could be beneficial to provide an API interface, making it possible to gain programmable access to the solution.

b) Frontend as a One Page App: We are considering to create a Javascript One Page App that will integrate the frontend within the client, instead of a server-rendered web page, to interact with the RabbitMQ broker, thus allowing the client to be independent from a webserver.

c) User interface: The user interface is still a prototype, and its future implementation will comprise an interactive admin tool for administrators, and a user interface that provides useful settings such as enabling or disabling the checkpointing feature or specifying the interval between checkpoints dynamically.

d) Security: If the solution were to be publicly available, security measures should have been taken into account. This could include password-based authentication and validation of users. Furthermore, a user currently has no limitation in the system and could potentially perform a Denial of Service attack by overworking all edge nodes and thus interrupting the normal operation of the solution.

e) Support for more languages: Right now it is only possible to upload python scripts only as jobs. Future versions could add support for more languages. The orchestrators could then create images specific to each language, to be used as a basis for the execution of the users' jobs. For example, a C# image could be created with a pre-installed .Net Core environment as well as the ability to download a specified list of Nuget packages. Likewise, a JavaScript image could have a Node.JS, Dine, ExpressJS or similar environment pre-installed, and download NPM packages.

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] A. Droob, D. Morratz, F. Langkilde Jakobsen, J. Carstensen, M. Mathiesen, R. Bohnstedt, M. Albano, S. Moreschini, and D. Taibi, "Fault tolerant horizontal computation offloading," in *Proc. of IEEE Int. Conf. on Edge Computing & Communications*, 2023.

[2] ORACLE, "What is iot?" 2022. [Online]. Available: https://www.oracle.com/internet-of-things/what-is-iot/

[3] IOT Analytics, "State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally," 2022. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[4] 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.

[5] D. Dossot, *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

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

[7] 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.

[8] A. Avizienis, "Toward systematic design of fault-tolerant systems," *Computer*, vol. 30, no. 4, pp. 51–58, 1997.

[9] J. Miguel, M. Piattini, and A. Vizcaino, "Challenges and improvements in distributed software development: A systematic review," *Advances in Software Engineering*, vol. 2009, 01 2009.

[10] G. Orsini, D. Bade, and W. Lamersdorf, "Context-aware computation offloading for mobile cloud computing: Requirements analysis, survey and design guideline," *Procedia Comp. Sc.*, vol. 56, pp. 10–17, 2015.

[11] 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.

[12] 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.

[13] 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.

[14] Docker, "What is a container," 2022. [Online]. Available: https://www.docker.com/resources/what-container/

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

[16] SQLite, "Sqlite as an application file format," 5 2018. [Online]. Available: https://www.sqlite.org/aff_short.html

[17] M. Albano, L. L. Ferreira, L. M. Pinho, and A. R. Alkhawaja, "Message-oriented middleware for smart grids," *Computer Standards & Interfaces*, vol. 38, pp. 133–143, 2015.

[18] RabbitMQ, "Rabbitmq," 2022. [Online]. Available: https://www.rabbitmq.com/

[19] 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.

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

[21] 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.
[22] 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.
[23] 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.
[24] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of iot edge functions," in *Edge Systems, Analytics and Networking*, 2019.
[25] Agile Business Consortium, "Moscow prioritisation," 2014. [Online]. Available: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html
[26] E. A. Scott Jr, *SPA Design and Architecture: Understanding single-page web applications*. Simon and Schuster, 2015.
[27] RabbitMQ, "Topics," 2022. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-five-dotnet.html
[28] ——, "Routing," 2022. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-four-dotnet.html
[29] Microsoft, "Asp.net core blazor," 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0
[30] A. Bettany, M. Halsey, M. Halsey, and A. Bettany, "Managing file sharing," *Windows File System Troubleshooting*, pp. 55–75, 2015.

APPENDIX A
COMPUTATION OF BEST CHECKPOINT TIMING

This section will calculate and analyse the total expected execution time of a script with and without checkpointing, and provide formal proof for the formulas and the algorithm in subsection IV-E. This section uses the same definitions of subsection IV-E.

We model the fault's distribution as a Poissonian:

$$p(t) = \mu e^{-\mu t} \tag{2}$$

Equation 2 dictates that the greater t is, the smaller value of $p(t)$, and in particular that, given that no fault happened before time t_0, the probability of a fault between $t_0 + t_1$ and $t_0 + t_2$ do not depend on t_0, i.e., $p(t)$ is a memoryless function.

The probability for a fault to happen before time T is:

$$P_F(\mu, T) = \int_0^T p(t)\, dt = 1 - e^{-\mu T}$$

If we have the a priori knowledge that a fault occurs before time T, the expected time for the fault is:

$$T_F(\mu, T) = \frac{\int_0^T t p(t)\, dt}{\int_0^T p(t)\, dt} = \frac{\left[-t \cdot e^{-\mu t} - \frac{e^{-\mu t}}{\mu}\right]_0^T}{[-e^{-\mu t}]_0^T} = \frac{\frac{1}{\mu} - Te^{-\mu T} - \frac{e^{-\mu T}}{\mu}}{1 - e^{-\mu T}}$$

For the sake of clarity, in the following formulas we will use $P_F = P_F(\mu, T)$ and $T_F = T_F(\mu, T)$. The time to complete the task, given that faults lead to restarting the computation, can be computed as the probability of having 0 faults multiplied by T, plus the probability of having 1 fault multiplied by the sum of T and T_F, plus the probability of having 2 faults multiplied by $T + 2T_F$, etc, leading to the expected execution time:

$$E_X(\mu, T) = (1 - P_F)T + (1 - P_F)P_F(T + T_F) + (1 - P_F)P_F^2(T + 2T_F) + ... = (1 - P_F)\sum_{i=0}^{\infty} P_F^i(T + iT_F) = e^{-\mu T}(T\frac{1}{1 - P_F} + T_F\frac{P_F}{(1 - P_F)^2}) = \frac{e^{\mu T} - 1}{\mu}$$

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

$$E_Y(\mu, T, N, C) = NE_x\left(\mu, \frac{T}{N}\right) + (N - 1)\ C \tag{3}$$

It is interesting to study how Eq.3 depends on N:

$$\frac{d\, E_Y(\mu, T, N, C)}{d\, N} = E_x\left(\mu, \frac{T}{N}\right) + N\frac{-e^{\mu T/N}\mu T}{\mu N^2} + C = \frac{Ne^{\mu T/N} - N - \mu T e^{\mu T/N} + C\mu N}{\mu N} = \frac{e^{\mu T/N} - 1}{\mu} - \frac{T}{N}e^{\mu T/N} + C$$

The second derivative is calculated as:

$$\frac{d^2\, E_Y(\mu, T, N, C)}{d\, N^2} = -\frac{T}{N^2}e^{\mu T/N} + \frac{T}{N^2}e^{\mu T/N} + \frac{\mu T^2}{N^3}e^{\mu T/N} = \frac{\mu T^2}{N^3}e^{\mu T/N} \tag{4}$$

Given that we are considering $\mu > 0$, $T > 0$ and $N \geq 0$, Eq. 4 is ≥ 0, thus $E_Y(\mu, T, N, C)$ is convex and it has either one minimum for $N > 0$, or an absolute minimum in $N = 0$. This proves that Algorithm 1 computes the best checkpointing strategy correctly.