Fault Tolerance Support in an R P2P cycle-sharing system

Tiago Alexandre Serafim Monteiro

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Dr. João Coelho Garcia

Examination Committee

Chairperson: Prof. Dr. Paolo Romano

Supervisor: Prof. Dr. João Coelho Garcia

Member of the Committee: Prof. Dr. Luís Manuel Antunes Veiga

January 2021

Acknowledgments

First and foremost, I would like to sincerely thank my advisor, Professor Joao Garcia, for introducing me to this topic and his guidance during this master thesis, for always being available to help as much as possible, and still being honest sincere with me. I would also like to thank my family, who has always unconditionally supported and motivated me throughout this process. I owe every achievement, especially to my wife, who still remembered that I had a master thesis to end, finally, to my friends, who have always been by my side and kept me going through the most challenging times.

Abstract

Volunteer computing has the goal of taking advantage of idle computing cycles to use in big computations. Several systems have already successfully explored this possibility. In some of these systems, it is possible to be a volunteer and a client and only a volunteer for essential projects. This paper introduces a new system that takes this advantage of the R language, providing a market to buy/sell remote computation time. This system focuses on the R language to make these remote computations secure and reliable since the computations are frequently long in R. In this paper, we focus more on the fault-tolerance problems of this new cycle-sharing system, like the possibility of a volunteer leaving the volunteer network causing the loss of this computation. We explore the existing solutions and adapt them to this system, making it fault-tolerant, providing more information on the remote computations to the clients, and using the network's idle cycles to make these computations faster possible.

Keywords: Volunteer Computing, Cycle-Sharing, RemotIST, Partial Results, R-project, R language, Checkpoint, cycle-sharing checkpoint, cycle-sharing parallel computing, fault-tolerance

Resumo

A computação voluntária tem o objetivo de aproveitar as vantagens dos ciclos de computação ociosos para usar em grandes computações. Vários sistemas já exploraram com sucesso essa possibilidade. Em alguns desses sistemas é possível ser voluntário e cliente e em outros apenas voluntário para projetos importantes. Neste artigo, apresentamos um novo sistema que aproveita essa vantagem para a linguagem R, proporcionando um mercado para compra / venda de tempo de computação remota. Este sistema é focado na linguagem R para tornar esses cálculos remotos seguros e confiáveis, já que em R os cálculos são frequentemente longos. Neste artigo, nos concentramos mais nos problemas de tolerância a falhas desse novo sistema de compartilhamento de ciclos, como a possibilidade de um voluntário deixar a rede de voluntários causando a perda desse cálculo. Exploramos as soluções existentes e as adaptamos a este sistema tornando-o tolerante a falhas, capaz de fornecer mais informação dos cálculos remotos aos clientes e utilizando os ciclos ociosos da rede para tornar estes cálculos mais rápidos possível.

Palavras-chave: Computação voluntária, compartilhamento de ciclo, RemotIST, resultados parciais, projeto R, linguagem R, pontos de verificação, pontos de verificação de compartilhamento de ciclo, computação paralela de compartilhamento de ciclo, tolerância a falhas.

Contents

List of Figures

List of Tables

List of Acronyms

VC Volunteer Computing

BW Bandwidth

PC Parallel Computing

SN Super Node

P2P Peer to Peer

PGSQL PostgreSQL

1 Introduction

The opportunity to use idle computer cycles has gotten the computing world's attention due to the immense advantages. Volunteer computing consists of a set of donators that give their resources to projects, which use the resources to do distributed computing and/or storage [61], this is, sharing their CPU cycles and storage. SETI@Home [1] was the first successful project in VC, and later SETI@Home's core software evolved and became the Berkeley Infrastructure for Open Network Computing (BOINC) [6]. BOINC is the largest and most successful "volunteer computing" project, using donator resources to help more than 50 known projects [1]. One of the things that makes this cycle-sharing concept so successful is that many users are willing to provide their resources [3][4]. However, there is no guarantee that the resources will be available in a volunteer network when they are needed. This turns VC into a complex system where choosing the volunteers is critical and managing the resources available is challenging. When a volunteer leaves the system with a running computation, the computation may be lost, and it is necessary to restart it. Therefore having a fault-tolerant system is almost mandatory to make this VC possible.

The RemotIST project [7][8][9] is a project developed at Instituto Superior Técnico (IST) to use some of these available idle cycles to help the R community with their heavy computations using a marketplace for the exchange of credits for computations. R [10] is a language and environment for computing and graphics design increasingly used by scientists and data miners to develop statistics and data analysis. R is one of the leading languages in data science along with Python and therefore increasingly popular. The big problem with R is that its users end up suffering due to the processing time of large amounts of data in which the lack of resources makes the results time-consuming. RemotIST uses a peer-to-peer system to provide a volunteer computation, sending the code to be executed remotely in a host and returning the results at the end of the computations. To make this P2P system, the volunteers must install a client software part of RemotIST that adds them to the client/donor network and starts to provide their resources or use the network's available resources. After this, they communicate with the centralized RemotIST server providing the code to be remotely executed. This centralizer server starts looking for donators who make the best offer to perform this computation. In cycle-sharing systems like BOINC, the volunteers donate their cycles to help projects, but they don't get cycles in return. In RemotIST, any node in the network can be a volunteer or a client. To make this network fair, a market to share the cycles was created. After each computation, the volunteer receives credits that he can spend to make his computations later in the cycle-sharing network. To prevent malicious code in the donators machine,

[1] Known projects are listed at https://boinc.berkeley.edu/wiki/Project_list

RemotIST implemented a sandbox that runs the client code. This sandbox helps protect the host against cases of unknown malware and software vulnerabilities, with security mechanisms that allow hosts to run untrusted programs in an isolated environment with limited access to the machine's resources and other information.

RemotIST has some problems that appear in a VC system like, failure prevention and recovery and using the resources available efficiently. Starting with monitoring, there is no notion of the running programs' status on the donator's machine. remote needs it because the users are paying for the computation and need to know if this is going well. This leads to significant problems with wasted resources. In an R program, it is possible to have computations that are too expensive. If these computations have some buggy code this means that the entire computation is useless for the client. It could be prevented if the client had this notion in the middle of computing and not in the end. The client should have some possibility of monitoring the status of his running application and, with the information from some partial results, can stop the computation at any time. There is also the possibility of the client to decide the deadline to have his computations done has been reached., This can be easy if we track the progress of the computation in the host machine and allow the client to check the status of his computation and can even help RemotIST in the case of clustering decisions in the future. This monitoring is not so trivial since we have massive data associated with the computations that require a lot of memory in R. This means that saving a variable and providing it to the client when he needs scales with the amount of memory that the variable occupies. Also, by tracking the progress, we add overhead to the computations if the progress checkpoints are not well spread.

In a cycle-sharing system, the availability of resources is not something we can take for granted. The volunteers can leave and join the network at any moment, even if they are making a computation. In [11], the host that accepts some job gives credit to make a security deposit and only recovers it after the computation. Even with this, we can't avoid the host's departure, which is the current failure problem of RemotIST. If the host decides to leave the network with some computation running, this computation will be lost. To tolerate this fault, tolerable techniques are required to prevent the loss of the computation or reduce the loss and recover the computation in another host. Recovering a computation in another host is not so easy in R. If we consider that most of the computations require a lot of memory and to recover, we must save all this application data, send it to the new host, and resume. This also includes the files that the client sends to start the application since it is desirable to make a client's computations without forcing it to continue online for the duration of the whole computation running in the hosts.

Using a cycle-sharing environment to make some computations may have the goal that the cycle-sharing system doesn't take more than the client's computer to finish the computations or computing expensive computations that were impossibly joining the resources available form more than one donator. This leads us to the last problem that we want to contribute to RemoteIST the efficient use of the available resources. R is strongly related to data science and mathematical computations, which require many resources like memory and CPU. Such resources may not be enough if only using one volunteer.

The opportunity of using more than one volunteer and guaranteeing that they share the computation and keep the system fault-tolerant is the desirable scenario for RemotIST.

This may collide with the credits systems because it is expected to use one host, requiring some policy adjustments. To focus only on improving the efficient use of the resources, we do not address this.

Our main contributions are the tracking of the status of the computation by adding some monitoring techniques to track the running code in a host, providing partial results to let the client check how is the computation, and provide fault-tolerance to the system by adding a managed network, a central marked and checkpoint that doesn't increase the computation's overhead drastically. And the last contribution is to make it possible for the system to use more than one host to finish a computation.

1.1 Objectives

RemotIST helps to make R computations easier in a cycle-sharing environment by having some progress tracking. It will provide better information to the scheduler and the owner of the computation. By adding checkpointing, it will improve the fault-tolerance of the system and prevent the loss of computations. The waste of resources, including the time, wasted waiting for results. Finally, by introducing some parallel computing and distributing the memory, the system is expected to improve the performance and to be able to make big computations in hosts with few available resources.

Our objectives are to provide some partial results to the client and ensure that we only need him to be connected at the beginning of the computation to transfer the code and files.

Hosts will potentially fail or disconnect. We want to make the system fault-tolerant using checkpointing, not losing any significant computation when a host leaves the network. This checkpointing can also help the resuming of some code stopped by the host.

The limitations of the resources available in the host machines must also be considered. Using some distributed memory and PC, we can reach the goal of making some big computations that were not possible having only one host doing it. This can lead to some credit exchange policies. Still, to focus solely on this objective, we will ignore the credits problem.

2 Related Work

2.1 R Programming

R is a programming language and environment for graphical computation and statistics [10] to develop statistical software and data analysis. R is one of the fastest-growing languages, has grown

incredibly in the last five years [15]. Its growth and popularity give R an essential role in companies such as Facebook and Google due to its success in the problems it solves [16][17]. R was also used in several projects with a positive impact [18], such as Bioconductor, which provides tools for the analysis and understanding of high-performance genomic data [19], Rmetrics, which is an open-source solution for market analysis financial and evaluation of financial instruments [20] and R-forge which provides a central platform for the development of R packages, R related software and other projects [21].

The R language is an S dialect designed in the 1980s and has been widely used in the statistical community [22]. S emerged with the need for a system to support research and substantial data analysis projects in the Bell Labs statistical research group [24]. The initial version was almost entirely composed of Fortran subroutines or subroutines of other libraries. S's development allowed this to become an evolved programming language for functions written in S that dynamically invoked subroutines and that used an interface with C for the subroutines in Fortran. R can be considered as a different implementation of S. Still, S adopts some differences regarding the lexical scoping, models, and other differences that make the R “cleaner” [25]. Like S, R is also an environment because it was thoroughly planned and coherent, rather than an incremental creation of concrete and inflexible tools, often the case with other software for data analysis [26]. R has C-like syntax and is mostly written in C and Fortran [23] but with functional language semantics, thus referring to LISP languages. R is a system for statistical computation and graphics that provides, among other things, a programming language, high-level graphics, interfaces to other languages, and debugging features [22]. All functions in R and datasets are stored in packages. R comes with a package called "base" which is the standard package included in the R source code. This contains the essential functions that allow R to function well as all standard datasets and statistics and graphics functions. R also allows more packages to be created by the community as the main repository to manage these packages created by the CRAN community. This is the main one, but not the only one, since repositories like Bioconductor and Omegahat can also be downloaded. R also allows advanced users to create C code to manipulate R objects directly and bind to intensive tasks written in C, C ++, and Fortran.

Unlike other programming languages, R does not give direct access to computer memory but rather offers specialized data structures called objects in which the C code underlying all R objects are pointers to a structure with typedef **SEXPREC**; the different data types in R are represented in C by **SEXPTYPE**, which determines how information in the various parts of the structure is used. R consists of three types of objects that are calls, expressions, and names.

Names

In this type of objects-symbols are used to be referred to in the language R. Thus, symbols have the mode "name".An example is *as.list(quote(a + b)).*

Expressions

An expression contains one or more statements. A statement, such as *y←20.* Itis a syntactically correct collection of tokens. Expression objects are unique language objects that have R statements that are parsed but not evaluated. The main difference is that an expression object can contain several of these expressions.

Calls

In R, functions are objects that can be manipulated in the same way as any other object. Functions have three components: a list of arguments, a body, and an environment like in Fig. 1. An argument can be a symbol or a "symbol = default" constructor of the special argument "...". The "..." argument is special and can contain any number of arguments.

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

Figure 1- Function components in R

The body is a parsed R statement. Generally, it is a collection of instructions in braces. Still, it can be a single statement, a symbol, or even a constant. All linked symbols in this environment are captured and made available to the role. In R, we can have three function types:

- Closure. A regular function;
- Special. An internal function that does not evaluate its arguments;
- Builtin. An internal function that evaluates its arguments like abs(x) or substr(x, start=n1, stop=n2).

There is a special object called **NULL**. It is used whenever there is a need to indicate or specify that an object is missing. The absence of a data element in a dataset will not be a **NULL** object but the **NA** symbol. Environments can be thought of as consisting of two things: a-frame, consisting of a set of symbol-value pairs, and an enclosure, the pointer to an enclosing environment. When R looks for a symbol's value, the frame is examined and, if a matching symbol is found, its value is returned. Otherwise, the surrounding environment will be accessed, and the process will be repeated. The frame content of an environment can be accessed and manipulated by using *ls* (return a vector of character strings giving the names of the objects in the specified environment), *get* (return the value of a named object), and *assign* (assign a value to a name), as well as *eval* (evaluate an R expression) and *evalq* (evaluates an R expression in the quoted form of its first argument). The *parent.env* function may be used to access

the enclosure of an environment. Unlike most other R objects, environments are not copied when passed to functions or used in assignments. Thus, if you assign the same environment to several symbols and change one, the others will change.

2.2 Progress Track and Partial Results

When a client submits his code to be executed remotely by a cycle-sharing system, he loses the capability of controlling and monitoring the executing code. To give back this power to the client is necessary to track the progress of the running computation in the host machine. Tracking this computation and providing fresh partial results adds some overhead to the execution of the program. The goal is to minimize this overhead using the best tactics to track it. Some tradeoffs like the freshness of the partial results are not immediate but from a computation that ran T time ago.

2.2.1 Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing

A novel method for monitoring a Java application's progress with low overhead is presented in [33]. They use a Pastry [34] p2p network for cycle-sharing with distributed hash tables (DHTs) and monitor the execution of the code in each node with some computation. To monitor the running code, they use beacons, which will periodically emit some indication of the program's progress using the functionality of JVM that monitors the "hotness" execution of methods by using sampling code that is periodically executed or observing current active methods. Fig. 2 presents the system that monitors a running program. These beacons are buffered by a reporting module implemented as a separate process to reply to queries from the owner of the running program. Having this report module allows the query to be immediately answered. This reporting module provides two big advantages:

(i) the application program does not have to suspend itself while waiting to be probed by the job owner. Instead, the data is buffered in the reporting module;
(ii) the design of the beacons is decoupled from the design of the queries.

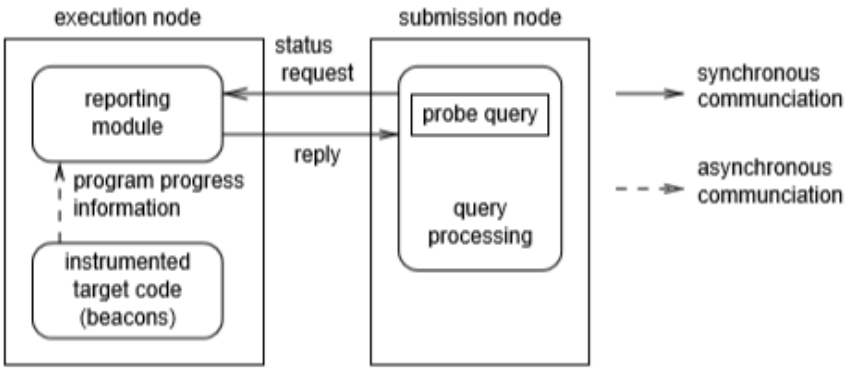

Figure 2 - Compiler instrumentation for monitoring job progress from [33]

Having this reporting module running asynchronously from the running application allows the system to have low overhead since the user queries don't need to be fresh. They can have a small delay that improves the system's performance.

This progress report is also used to search for fraudulent nodes by comparing the reports for each computation. RemotIST is not so easy since statistical computations often use random numbers, which means that two hosts can have different results for the same computations. None of them is a fraudulent node.

2.2.2 Monitoring Remotely Executing Programs for Progress and Correctnes

In the case of [35], despite being written in Java, it does not use the JVM functionality to monitor the running application. Before the program starts to be executed, a tool is transformed into a program that executes in the host machine, the H-code, and another to be executed in the submitter machine, the S-code. The S-code runs in the submitter's machine or another machine that the submitter trusts and uses this code to track the progress and verify the H-code execution. The H-code includes two types of beacons, the location beacons (L-beacons) and the recomputation beacons (R-beacons).

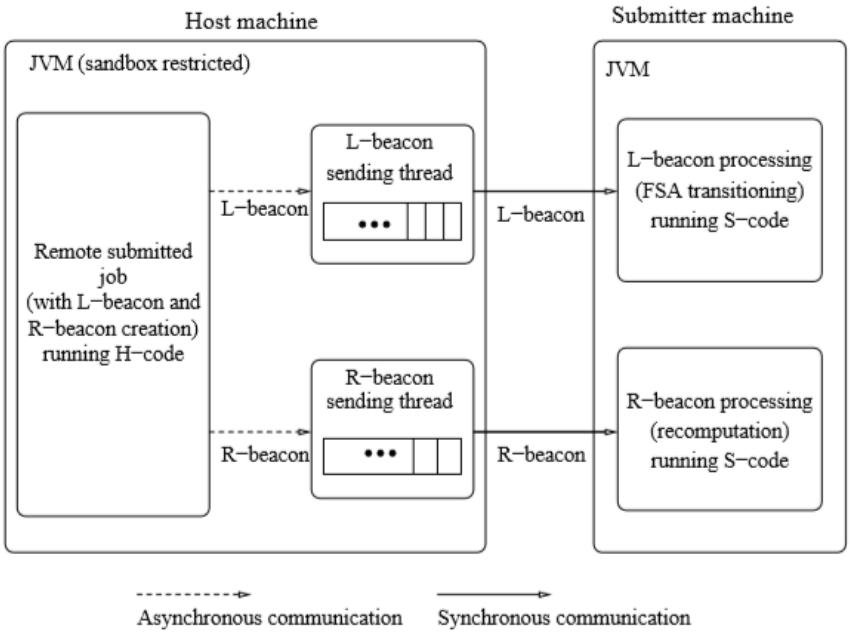

Figure 3 - Monitoring system's Components from [34]

In Fig. 3, the monitoring systems components are presented. We can see how the L-beacons and R-beacons are transmitted. The L-beacons emit information to the submitter at some significant execution points. This allows the submitter to know which parts of the program have been executed at the current moment. To distribute these L-beacons in the H-code program, they start by analyzing the code and inserting at the beginning of the methods that execute a significant amount of words. For the S-code, they generate a finite state automaton to allow the submitter to track the computation. Fig. 4 shows the need for an FSA since the execution of the methods can execute some other methods with L-beacons. Using an FSA, the users can track more precisely the current execution of the method.

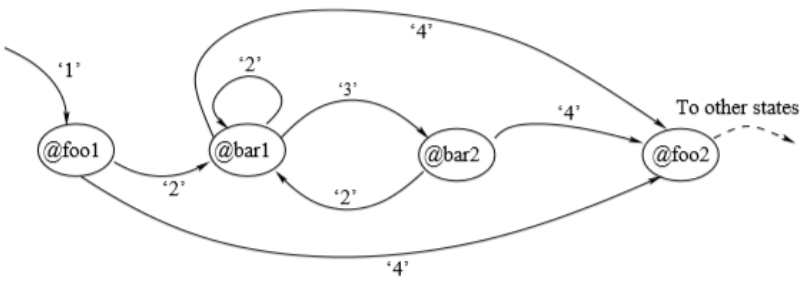

Figure 4 - A part of FSA derived from the Java Grande LU benchmark from [34]

R-beacons are used to check the validity of the host. They use an R-beacon message (IDC, input, result) where IDC is the computation ID and sent it to the submitter to let him verify it. The submitter using the IDC gets the code to be executed and executes it with the input to check the output's validity. To prevent replay attacks from a cheating host, the submitter calculates each input's message digest and the IDC in each R-beacon message using the MD5[36] algorithm. In RemotIST, such a task will not be so trivial due to the presence of randomness.

2.3 Checkpoint

The opportunity to harvest CPU time from idle computers is not a new topic. In the cycle-sharing system, we have the successful example of SETI@home [1] that attracts many volunteers to a scientific effort. In a cycle-sharing system, every computation is associated with a cost, a CPU cost. Losing a computation means losing resources. Such losses are common since volunteer faults are very common. It is usual for volunteers to exit and enter the volunteer market (so-called churn). This makes computation checkpoints essential in a cycle-sharing system to keep track of the computation and allow a rollback, reducing the cost of a lost computation in the system.

Checkpointing is a technique that consists of taking a *snapshot*, an image of the system, of the application's state at some time and saving this snapshot to some storage. Saved checkpoints are used for rollbacks, consisting of using the last checkpoint to recover the last computation and resuming it, using the checkpoint information to construct the previous state of the application by resetting the environment resuming the previously running state.

With this, it is possible to have a fault-tolerant system that saves the computations of a host. When this host leaves the system, we can recover the computation from the last checkpoint and resume it, saving the user resources and not wasting time to compute everything all over again. Checkpoints can also help to move the computation when we predict the failure of resources [41]. By predicting these failures and using checkpoint, we can start moving computation to another host before the current host fails, saving computing time.

Some systems like Condor [12] and Sun Grid Engine [13] use a dedicated host to keep the computations' checkpoints to make sure the cost of a loss is as small as possible. Some hosts are willing to contribute their disk storage. Still, such a solution is complex because of the availability problem, which corresponds to not having the resources available when needed, and the possible ensuing loss of checkpoints. Another problem is the network usage between the volunteers that store the checkpoints and the volunteers that make the checkpoints and send them to the storage volunteer. The potential high network usage may harm the overall system's performance. The last issue is that, since the generation of the checkpoint and the rollback to the last checkpoint have a big impact on the application performance, it is important to control the checkpointing overhead but keep a good checkpoint interval to allow

the rollback loss to be the minimum. It is important to find a good trade-off and find a good checkpoint creation interval.

2.3.1 Fine-Grained Cycle Sharing (FGCS)

Fine-Grained Cycle Sharing (FGCS) [14] systems aim at utilizing a large amount of idle computational resources available on the Internet and solve some problems of a cycle sharing system using failure-aware checkpointing techniques. In FGCS, a checkpoint contains the entire memory state of the application. Due to the lack of dedicated hosts, they store the checkpoints on non-dedicated hosts splitting and encoding a checkpoint of size *n* into *m* + *k* fragments of size *n/m*. The checkpoint fragments are then stored on *m* + *k* different storage hosts. To pick the hosts, they use the Network Weather Service (NWS) [61] to calculate the effective end-to-end bandwidth. Based on this metric, the total network overhead for transferring a checkpoint is the summation of the latencies incurred by each fragment. The goal is to minimize the network overhead (N) of transferring checkpoints and the re-execution cost (R) caused by losing a checkpoint that is needed to recover a guest job. They use the semi-Markov Process (SMP) method for predicting the availability of both CPU cycles and disk storage to monitor the host I/Os and pre-empt the checkpoint read/write if necessary because if fewer than *m* repositories are available for retrieving the checkpoint fragments, the application to be recovered will lose all the computation and restart. To peek at the repository, they also used an optimistic scheme and a pessimistic scheme. The optimistic scheme assumes that most storage hosts present high availability. The pessimistic scheme expects the storage hosts to fail frequently and updates the selection of repositories at each checkpoint interval.

They use a one-step look ahead heuristic for the checkpoint intervals, dividing a job execution into multiple steps of a fixed length. At the beginning of each step, they decide if the job needs to be checkpointed by comparing the cost of checkpointing at that moment and the cost of delaying it to the next step. To split data, they apply Michael Rabin's classic information dispersal algorithm (IDA) [37] that allows coding a vector of size *n* into *m+k* vectors of size *n/m;* thus, it is possible to tolerate failures with a storage overhead of only *k*n/m* elements. To start the recovery, the first step is to pick the new host based on the algorithm from their previous work [38]. The next step is to update the task size based on the available checkpoint fragments. After selecting the new host, new checkpoint repositories are selected. The host can regenerate the state from the available fragments of the checkpoint. FGCS also ensures that the checkpoint remains valid until the next checkpoint is completed using the strategy create-before-destroying, which is not deleting any previous checkpoint before the next one is complete. FGCS was able to improve the application performance by up to 9.4% compared to Condor.

2.3.2 Falcon

Falcon [39] is the continuation of the previous work in [14]. Some problems are using the failure model of compute hosts in storage hosts, the bad load-balancing for the storage hosts, improvements in storage failure prediction, and static BW measures. In Falcon, they decided to make a failure-model to predict the storage hosts' availability and consider the load-balancing of the storage hosts, predicting the reliability of the hosts using absolute temporal reliability. Considering that the network's static measure is not enough, they decided to consider the available BW.

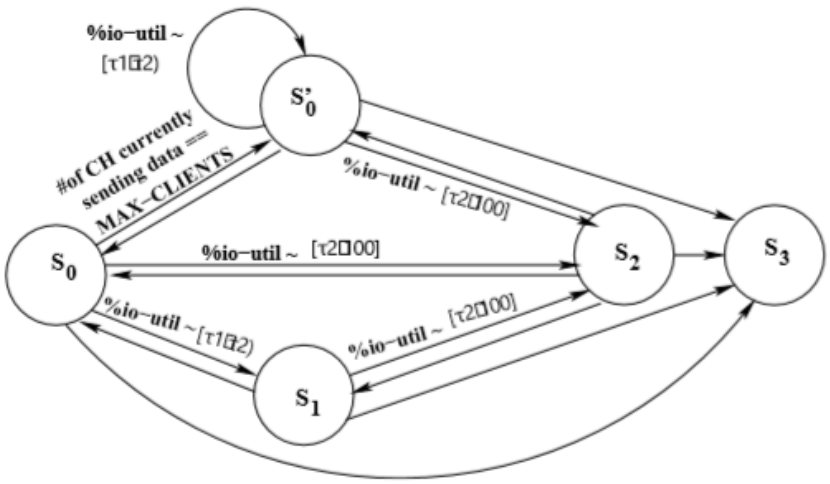

Figure 5 - Storage hosts a multi-state model from Falcon [39]

Fig. 5 presents the multi-state storage model. Using this state model is possible to rank the storage hosts and make the load balancing between them where the states are the following, where τ1 and τ2 are variables that can be adapted to the configuration that represents a load percentage:

(i) S0: storage host is running with *I/O load < τ1* and the number of compute hosts sending checkpoint data concurrently is < *MAX-CLIENTS*

(ii) S′ 0: number of compute hosts sending checkpoint data concurrently *is = MAX-CLIENTS*

(iii) S1: I/O load of storage host is between *[τ1, τ2]*

(iv) S2: I/O load of storage host is between *[τ2,100%]*

(v) S3: storage host is not available due to resource revocation

When a host requests the storage host to save the checkpoint, the storage host replies based on his state. If S0 or S1 reply, ok. Otherwise, it does not accept any requests. In S3, the failure is considered irrecoverable. To predict the hosts' availability, they defined a Correlated Reliability Load Score (CRLS), an equation developed in [39], using this multi-state model and probabilities. The previous work used the effective BW between the compute host and the storage host, which is the maximum BW possible. This may lead to some load balancing problems. They switched to the available BW that is the network's unused capacity, which changes with time. With this approach, they are also able to define the network overhead of transferring a checkpoint. Using an objective function that tries to balance the checkpoint storing overhead and the re-execution cost if that checkpoint were not taken, they use a greedy algorithm based on this objective function to select the storage hosts. This storage selection and the transfer of the checkpoint fragments are executed in parallel. In the previous work, they presented an optimistic and pessimistic storage selection algorithm. In the pessimistic one, they select a new set of storage at the beginning of every checkpoint leading to unnecessary overhead. To improve this problem currently, the pessimistic scheme only selects new storage when it needs, by checking if the current set has any storage in state S2 or S3. Fig. 6 presents the Falcon system based on all these new features. Falcon has significantly better results when compared with a dedicated host strategy based on the evaluation executing on the functional system of Purdue's BoilerGrid[40].

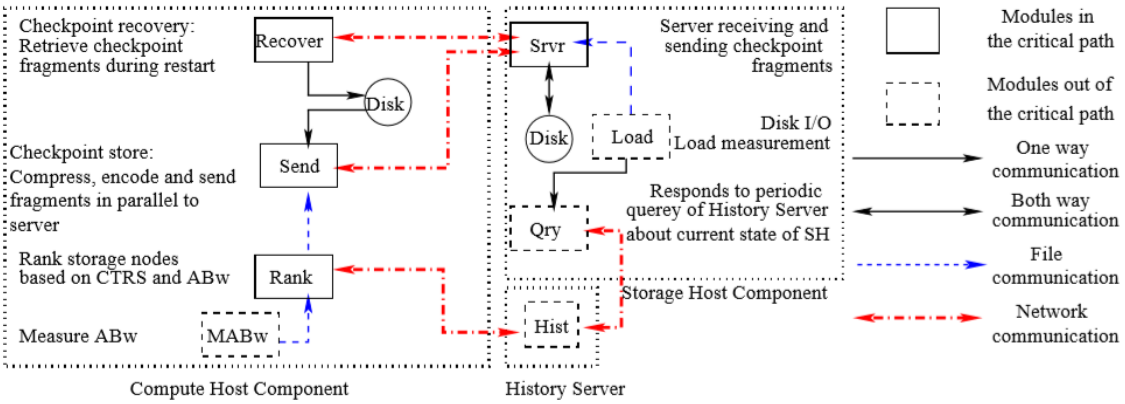

Figure 6 - The falcon system from [39]

2.3.3 CoCheck

Checkpointing techniques are not new. As we saw in the previous papers that we saw, there is always some space to improve it, but these changes when it comes to parallel platforms. Techniques for checkpointing in parallel and distributed systems are not new, like the Chandy and Lamport protocol for "*distributed snapshot*" created in 1985 [42]. However, the implementations of this parallel checkpoint are rare. An example of a parallel checkpoint is CoCheck[43]. CoCheck implements a protocol much like Chandy and Lamport's distributed snapshot protocol and uses Condor's checkpoint ability. Fig. 7 shows how this protocol works. It all starts with the resource management (RM) that decides when to start the checkpoint. This decision can be due to an application request, a change in the state of a resource, or a periodical checkpoint. This RM broadcasts the checkpoint message to all the checkpointing processes. After this, the checkpointing processes send a ready message to all the other checkpointing processes. With all the ready messages delivered, it is safe to use Condor's checkpoint library to save the state.

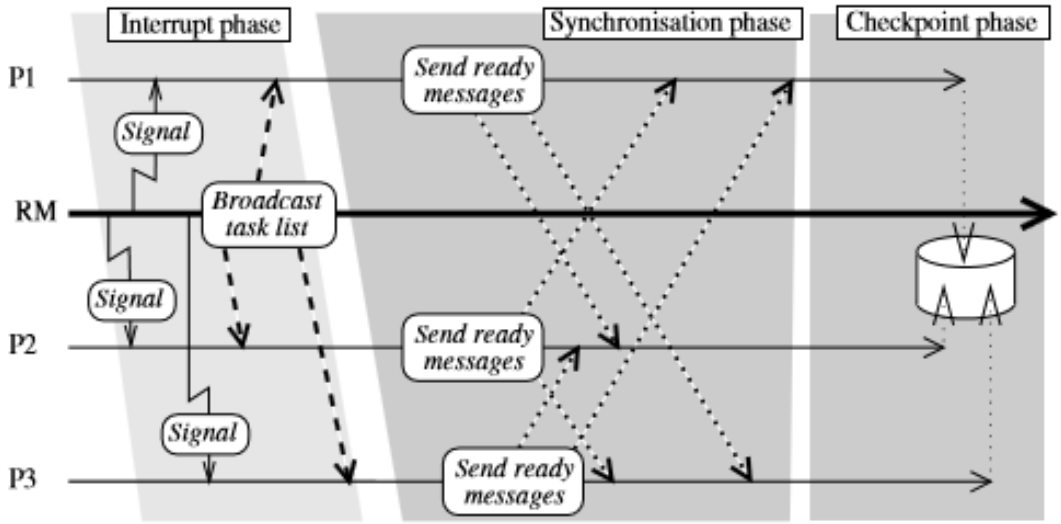

Figure 7 - CoCheck's protocol from [43].

2.4 Parallel Computing

With various cores in a computer being something normal, the need to take advantage of all these cores in a computer began to gain attraction. Still, writing parallel programs were difficult and tedious. PC is a type of computation where many calculations or the execution of processes are made simultaneously. Using the multi-core feature is one example [44].

2.4.1 MPI

The first development of a multi-core model was the Message Passing Interface (MPI[2])[45]. A process is a program counter and address space. These processes can have multiple threads sharing a single address space. MPI unlocks the communication with this process that have separate address spaces. To unlock this communication is necessary to have synchronization and the data present in one address space to another address space. MPI is a library designed for parallel computers, clusters, and heterogeneous networks. The user doesn't need to learn everything about MPI. Fig. 8 is represented a minimal MPI program written in C language, the MPI_Init starts the MPI parallelization, and the MPI_Finalize stops it. Suppose we want to know more about the processes running. In that case, MPI_Comm_size returns the number of processes present in the computation, and MPI_Comm_rank returns a number between 0 and size-1, identifying the process.

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    int size;
    int rank;

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Print a hello world message
    printf("Hello world");

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Figure 8 - MPI example of usage in C

[2] MPI is maintained at https://www.open-mpi.org/

The communication of the process is made with interfaces MPI_SEND and MPI_RECV. In the case of collective operations, we can replace this SEND/RECEIVE by BCAST/REDUCE. Best distributes data from one process to all others. Reduce combines the data from all processes and returns it to one process, very similar to Map-Reduce that will be introduced later in this chapter. MPI nowadays is in version 4.0.0, has a robust interface, and is full-featured.

2.4.2 OpenMP

MPI has standardized the message passing model but has a difficult programming model. It requires data structures to be explicitly sharded to allow the entire program to be parallelized. This and the fact that Pthreads [46], a standard for PC with shared memory, only has limited support in Fortran and C is was difficult than the necessary motivated the development of OpenMP [47], a standard API for shared-memory programming. OpenMP was designed to be flexible and easily implemented with the following features:

- Control structures like parallel, do, and single.
- A data environment with objects that can be shared, private, or reduction.
- Both implicit and explicit synchronization; implicit at the beginning and end of control structures (which can be removed with nowait clause); the user can specify explicit synchronization to manage order or data dependencies.

In Fig. 9, we can see the parallel control structure's use running a simple hello world using three threads and using the data environment shared. Using these control structures makes parallel computation easier for the users since OpenMP manages the shared memory and synchronizations. The user only needs to use the control structures and the data environment.

```
#include <stdio.h>
#include <string>

int main() {

  std::string b = "Hello World\n";

  #pragma omp parallel shared(b) num_threads(3)
  {
    printf(b);
  }
  return 0;
}
```

Figure 9 - OpenMP example in C

2.4.3 Google MapReduce

MPI, OpenMP, and PThreads became the standard for parallel programming. MPI uses distributed memory, and OpenMP and PThreads use shared memory. However, to develop parallel programs remained somewhat complex to make a program parallel with his steep learning curve and potentially low productivity. In reaction to this complexity to implement special-purpose computation to process a large amount of raw data, Google designed a new abstraction that allows simple computations but hides the parallelization, fault-tolerance, data distribution, and load balancing details. Based on the Lisp and many other functional programming languages primitives called map and reduce, Google created a programming model called MapReduce [51] implemented in some open-source systems like Hadoop [50] and Phoenix [48][49]. The programming model of MapReduce takes a set of input with key/value pairs and returns an output of key/value pairs only using the two present functions map and reduce. In the example of Fig. 10 map function emits a word with the associated count of occurrences, and the reduce function sums all the counts together.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Figure 10 - MapReduce functions

Google presents a Master-Worker for the implementation of the MapReduce present in Fig. 11. First, it splits the input files into M pieces and starts many copies of the program on a cluster of machines. One is the master, the others the workers. The master selects idle workers and assign map and reduce tasks. The workers with map tasks read the sliced input of that task, parse the key/value pairs and pass

each pair to a map function defined by the user. The result of map functions is saved in memory. The master is responsible for forwarding this result to the reduce works. The reduce workers make a remote read and pass it to the reduce function that produces an output.

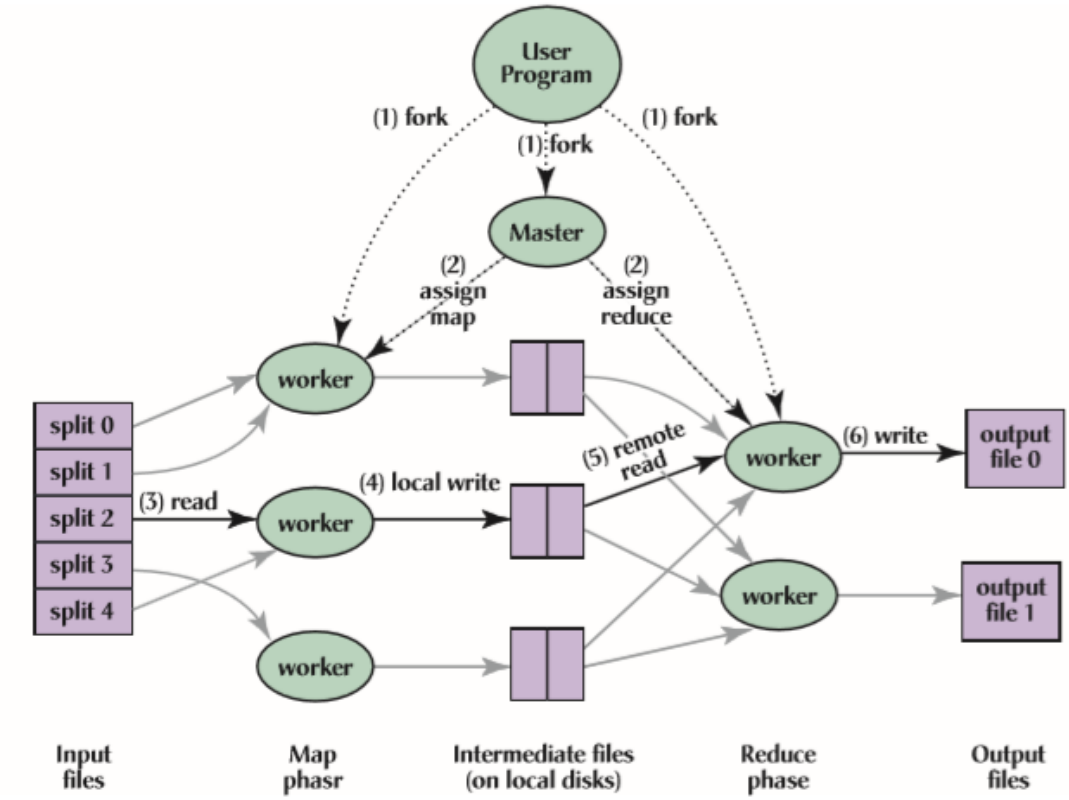

Figure 11 - Google's MapReduce architecture using master/worker scheme from [51]

The simplicity and efficiency of MapReduce made it popular and widely used. Still, Google's master/worker scheme had a potential bottleneck when applying this MapReduce to a VC because of its centralized architecture. BOINC-MR [52] is a BOINC prototype that ran MapReduce jobs like the previous example in a volunteer network but not using the master/worker scheme that google presented. In BOINC-MR, the tasks have no dependencies or shared data between them, allowing traditional scheduling mechanisms. Another performance issue is shared-memory management. Tiled-MapReduce [53] modified the MapReduce paradigm combines the tasks to minimize resource usage. Creating sub-jobs from large MapReduce jobs replaces the Map phase with a loop of Map and Reduce phases. Each iteration processes a sub-job and generates a partial result that can be saved for computation reuse. The reduce phase process these partial results of all iterations and produces an output equal to the traditional reduce phase. A Map-Reduce System with an Alternate API for Multi-Core Environments (MATE) [54] uses a different approach using an API that requires the user to write combined map/reduce functions. This API is implemented on top of Phoenix.

Both map and reduce steps are combined into a single step called Reduction. Each data element is processed and reduced before the next element is processed. MATE outperformed Hadoop and Phoenix by reducing the original MapReduce's overhead and minimizing the memory resources needed. MATE exposed some problems with the Phoenix MapReduce implementation, and Phoenix++ [55] improved Phoenix's performance with a different approach from MATE and Tiled-MapReduce. Phoenix++ started by analyzing the problems of Phoenix. The first problem was the inefficiency of the key-value storage. They identified different key distribution types, the *:* any map task emits any key, the *:k any map task can emit a fixed number of k keys, and 1:1, each task outputs a single key. Phoenix used a fixed-width hash table that was not good for a *:* key distribution. With this, Phoenix ++ created containers that provide a group by functionalities based on the key distribution:

- Hash containers for *:* that can be resized
- Array container for *:k with a fixed size and a priori know range *[0:k-1]*
- Common array container for 1:1

The other problem with Phoenix was an ineffective combiner. The combiner function has the goal to reduce the key-value pair. However, in Phoenix, this approach tends to increase the memory instead of minimizing it as expected because the generated key-value pairs must be stored and when the combiners are run, those pairs may not be in cache, causing a big impact on accessing the memory, and because of this, the memory allocation was a big problem. To solve this problem, Phoenix++ created combiner objects making combiners not only function but stateful objects. These combiner objects are sent to the reducer functions. To reduce memory pressure to the key-value storage, Phoenix++ used the strategy to invoke the combiner immediately after each map function eliminating the need to copy values when concatenating buffers from different threads. Because it doesn't buffer the values, it uses the fact that each combiner function will run immediately after the map function to maintain a single aggregate value, incrementally combined with each emitted value, eliminating the overhead of having a buffer. Phoenix++ achieved higher performance without changing the MapReduce paradigm instead of changing it as Tiled-MapReduce and MATE did.

PC had a big evolution from the traditional fully handmade solution, to the message passing interface and to the simplistic, easy to lean and apply, and with big performance MapReduce. Even MapReduce needs some performance improvements depending on the context that we apply it. Now that we have a good knowledge of parallel programming, since R is our programming language, the last part of this chapter explores some packages[3] that make parallel programming in R possible. R is growing. With it, the need for high-performance computations because of the larger datasets and the increase of the computational requirements. As an area of growing datasets, we have the example of bioinformatics where data sets appear to be growing at a rate that is faster than the corresponding performance increases in hardware and some computationally demanding solutions like simulations and resampling. There are a lot of techniques for PC like Shared Memory and Distributed Memory. One example of shared memory is OpenMP. One example of distributed memory is MPI.

[3] Packages available in R for parallel programming are listed in https://cran.r-project.org/web/views/HighPerformanceComputing.html

2.4.4 R Packages

The most relevant packages in R for parallel programming in clusters are Rmpi, an R interface (wrapper) to MPI used as the core for many more packages in R and snowfall, which is built as an extended abstraction layer above the well-established snow package [57] that supports simple PC in R.

For grid computing, there is no package with a good stable version. Still, there are available packages like GridR [32], MultiR [59], and Biocep-R [58]. GridR submits R functions to be executed in a grid environment (on a single computer or a cluster) and provides an interface to share functions and variables with other users (grid.share()) [27], and for its execution, it requires some software components like Globus Toolkit 4 grid middleware, an installation of the R environment on the grid machines, and a GRMS-Server installation from the Gridge toolkit. An example of usage of GridR is the EU project Advancing Clinico Genomics Trials on Cancer (ACGT). In ACGT, the R environment is used both as a user interface (client) on the client-side. As a tool in the grid environment and GridR, it is used as a tool for the remote execution of R code in the grid. GridR was removed from CRAN, the package repository for R, but it is possible to obtain older versions. In MultiR, the implementation should have some similarity to aspects of snow and gridR. Still, it should be independent of the many different types of hardware and software systems. It requires no additional software components (Globus, CoG, etc.) to be installed before it can be used. The main differences involve job submission (via Grid middleware). MultiR was presented in 2008 in userR! conference but ten years later still has no release. The Biocep-R project is a toolkit written in Java that allows R to be used as a Java object-oriented toolkit or as a Remote Method Invocation (RMI) [56] server. The Biocep-R virtual workbench provides a framework enabling the connection of all the elements of a computational environment.

For multi-core systems, we have pnmath and pnmath0 that use OpenMP and Pthreads. On loading, the packages replace built-in math functions with hand-crafted parallel versions. Another package is fork that provides wrappers around the Unix process management API calls: fork, signal, wait, waitpid, kill, and exit. And the last one R/parallel uses a master-slave architecture implemented in C++ and provides functions in R runParallel(), which enables automatic parallelization of loops without data dependencies. A data dependency occurs when the calculation of a value depends on the result of a previous iteration.

About the MapReduce application in R, it is possible to use some packages that provide an interface for Hadoop like rmr and RHIPE. Still, the package BatchJobs is the most relevant in this case because it provides map, reduce, and filter variants to manage R jobs and not only an interface for Hadoop and the Rhpc package permits **apply()* style, the R function, dispatch via MPI.

2.5 Distributed Consensus and Consensus Algorithm

One of the fundamental primitives for constructing fault-tolerant, strongly consistent distributed systems is distributed consensus [73]. Distributed consensus ensures consensus of data among the nodes of a distributed system to agree on a proposal. A consensus algorithm [74] is a mechanism through which a network agrees on something proposed on the network. For example, imagine that you and your family are going on a trip. You have two possible destinations. A consensus algorithm aims to ensure that you and your family reach a consensus about which destination you will travel to.

One algorithm implementing consensus is the traditional algorithm Paxos [75,76,77]. Paxos is a consensus algorithm that aims to achieve replica consistency, using 2 phases: Prepare and Promise and Propose and accept. In Prepare and Promise, a proposer sends a message with a given number to most of the replicas asking them to make a Promise that they will not accept any changes with a number lower than the given number. If the given number is the highest that the participant has seen, he accepts it. He replies with the last highest number seen and last accepted value. A Promise is fulfilled if most of the replicas accept it. In phase 2, Propose and accept, the proposer sends a value v to all the acceptors. If phase 1 had more than one value, the value with the highest number is sent. If this is the highest number seen by a participant, v is accepted by it, and else the request is rejected.

Another algorithm, simpler and recent, implementing consensus is Raft [78]. Raft approach can be divided into two subsections, the leader election, and log replication. The leader election is triggered when a Follower did not receive any heartbeat from a Leader. The Follower changes its state to Candidate and uses the RequestVote RPC in parallel to the other nodes requesting a vote to be elected as a leader. If it receives the majority of the votes, it becomes the Leader and starts sending heartbeats with the AppendEntries RPC. If it did not receive the majority of the votes or receives an AppendEntries from another Leader, it becomes a Follower again. In the log replication, the leader receives an entry from the client and starts sending it parallel to the network. Once the entry is received by most nodes using the AppendEntries RPC from the Leader, it is considered committed, and the leader replies to the Client.

Paxos and Raft are consensus algorithms that try to solve the same problem and where Raft can be more understandable and have a more lightweight leader election [79].

3 Architecture

This thesis aims to provide partial results to the clients, make RemotIST fault-tolerant to volunteer faults, and improve the computations on the remote executor.

In this chapter, we aim to describe the system's capabilities and how its architecture was influenced by the requirements for a fault-tolerant P2P sharing system. Sharing a cycle for voluntary computing in R requires attention to many requirements that require an architecture that can deal with problems such as remote execution, the transport of environments, the constant entry and exit of nodes in the network, and the tolerance to failures throughout the remote task execution process.

This work intends to be mainly concerned with the system's fault tolerance, recover old projects, and integrate them into this system. During the analysis of past work, it was concluded that some problems would be described in the architecture that made it impossible to advance quickly to the fault tolerance that had been planned since the beginning. It was important to review the state of the network and recover the basis of a system that guarantees to continue the objectives initially established. In this chapter, you will find some keywords like Client, Volunteer, and Market. The Client is someone that wants to use the network to execute his computation in exchange for credits. A Volunteer provides resources to the network to execute computations in exchange for credits, and the Market works as a central entity that manages the computations to be executed and the credits exchange.

3.1 Previous Work

During the review of the system, architecture to start the development of fault tolerance, it was concluded that the system did not incorporate features that were important to add to the scope of this project to add fault tolerance.

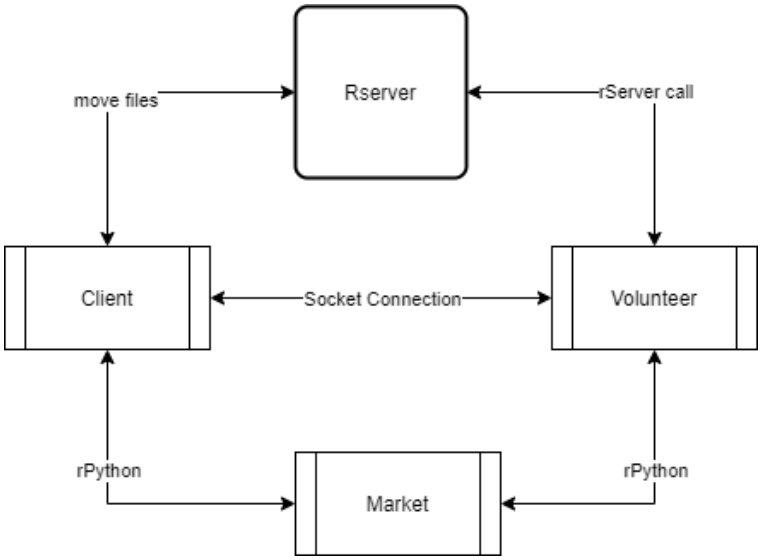

Figure 12 - Previous Architecture of the system

As shown in Fig. 12, the system was composed of 3 processes, Client, Volunteer, and Market and an external entity called RServe [71]. The Client connects directly to Volunteer via sockets connection, and both execute python functions using rPython [72] in the same process. The external entity RServe executes the files that are in its folder. For this, the Client moved its files to the RServe folder and notified the Volunteer process that executed the RServe. In the end, the Client moved the resulting files from RServe to his folder and loaded them into his RStudio environment. Some of the points that made it impossible to start developing fault tolerance were:

- **The system only ran on a single computer.**
  There was no notion of distributed systems, the whole system and voluntary computing was done on a single computer that made the Client, Volunteer, and Market simultaneously. Running such a system on more than one computer was impossible.
- **Blocked client.**
  Also, because the entire system only works on one computer, the Client process contacted the volunteer process and waited for his response. If any errors occurred during this waiting for a response, all computing was lost.
- **It was based on a local RServe**
  Also, one of the factors why computing was only possible on a computer was that whenever the Client wanted to do a voluntary computation, he moved his files to the RServe folder (possible because it was the same computer) and notified the Voluntary. The Volunteer then executed RServe, which in turn executed the files that the Client had placed there.

- **There was no concept of network and market.**
  The Client made calls to a python code using a rPython library (which has never been officially launched and is deprecated) that performed python functions that simulated a Market and accessed the database. There was not any notion of Market as any Client was in a way a Market. Another important factor is that there was no concept or network management. It was not known who was on the network, and there was no connection between the Volunteers.

It was concluded that the base of the system was composed of a code that simulated a Client that was at the same time Market that spoke with a process that simulated Volunteer but that in reality, the basis of everything was the RServe that executed the R code that the Client there and that later the Volunteer process would execute, all on the same machine.

3.2 Overview

Based on the previous architecture and the objectives of system fault tolerance, the system architecture was designed to attend to the previous architecture problems and enable the development of a stable platform that can be scalable and fault-tolerant. To provide such a platform, the system's design was based on the following requirements:

- The client may fail during computing or may not even be online during computing
- The Market, Client, and Volunteer must be independent
- The Market must be able to handle multiple Client orders and be scalable
- A Volunteer can join and leave the network at any time
- A Client's computing should not be lost when a Volunteer leaves during a computation
- The Client must be able to access partial results of the computation, know its status and cancel it
- The growth of the network should not affect the Market
- The network must be ready to enable checkpoint and PC

The system design, fig 13, introduced new entities and reused some existing ones, so when we refer to Client we are referring to the client who wants his job to be executed remotely, Volunteer the entity that wants to share its resources in exchange credits in a remote execution, Market to the system that controls the work of the client and the network, Worker the entity responsible for controlling and monitoring the works that are being performed by the Volunteer and supernode a Volunteer who is important in the network and who monitors the state of the other nodes connected to you.

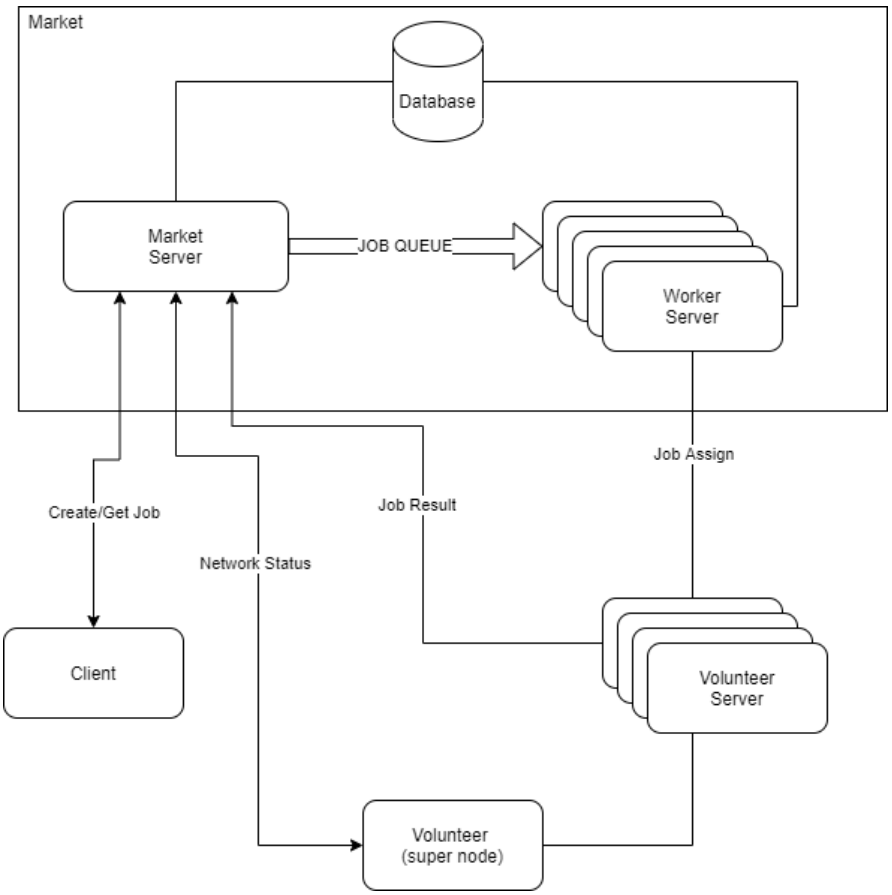

Figure 13 - Architecture of the system

The Client to create a job communicates with the Market and sends the necessary files/data so that the execution of that job can be remote. After receiving this information, the Market inserts the work in a Queue [62] that is accessible by Workers who, when receiving a new job from Queue, choose the best Volunteer available to perform the work. The list of online volunteers consulted by Workers is constantly updated by Super Nodes (SNs). When a Volunteer finishes the job, he sends the results to the Market to store the results. The Client can request the Market results at any time after the work is completed and upload these results to his local environment. A more detailed explanation of how the requirements were met will be made in the following chapters that explain in more detail each of the entities indicated in Fig. 13.

3.2.1 Client

As the objective is voluntary computing in exchange for credits for a Volunteer, there is always the person interested in having their computing performed remotely. This person is the Client. The Client aims to run a code using the resources that exist on the network for better CPU, memory, or simply running in parallel. In contrast, the Client is concerned with other computing. The Client must create a job, defining the requirements, and from there, the Market is responsible for choosing a Volunteer who will be responsible for Computing. There are some concerns that we try to address, such as:

- The possibility of the client going offline while computing the requested job
- The volunteer disconnects while computing the client's work
- The client code has errors that do not allow the job to be completed

So that the Client does not have to go online during this whole process, the Market, as soon as it receives all the necessary information for the Client's computing, starts its search for a Volunteer who meets the requirements and assigns a worker who is responsible for monitoring the work being done by the Volunteer. As soon as the Volunteer finishes the work, the results are stored in the Market so that the Client can later upload them to his personal computer.

There is a possibility that during the process, a Volunteer will not complete the job so that the Client will never be without his work due to a Volunteer connection failure. Whenever the network verifies that the Volunteer has dropped, a new Volunteer is chosen by the Worker to initiate a Client's work order. Ideally, the new Volunteer would continue computing the old Volunteer code using a created checkpoint.

Due to computation that would not be possible on the Client, but possible on a volunteer or even since no code review or tests have been done, the code to be executed remotely contains errors. Whenever a computation is interrupted in a volunteer due to an error in the code, the state of the environment is saved, saving the variables and saving the information of what caused the error so that the Client can consult later. Errors are not always exceptions. They can often have mutations in variables that do not go according to expectations. Another mechanism for solving this problem is the partial results and history of changes in the variable. The Client can define what he wants to observe a variable. Whenever it changes, that variable creates a record of what was changed and when thus keeping a history of changes in the variable that allows the Client to understand what happened to the variable and if it was supposed to. The Client can also load the state of the variable by indicating the time point in the history that he intends to load to do tests in his environment, so the Client can do tests on a variable in the middle of computing the volunteer.

The client can interrupt a job at any time. Credit collection issues are not addressed in this document. Whenever a computation/work is completed, a record is created in the market for the result of the same, so the client can have more than one job to be performed simultaneously without having the

consequences of losing the results of some work. The Client can then define which results to load into their environment and only delete the market results if they want. They are never automatically deleted.

3.2.2 Volunteer

A volunteer wants to make his resources available in exchange for credits without compromising his machine's safety. To perform a client's job, it is important to guarantee the following features in a volunteer:

- Execution of work by the client
- Partial results
- Checkpoint creation

A volunteer can ask the Market to enter the network. After receiving authorization, the Volunteer waits to contact a worker to distribute jobs that meet the parameters established by the Volunteer to accept a job (CPU, RAM, credits, etc.).

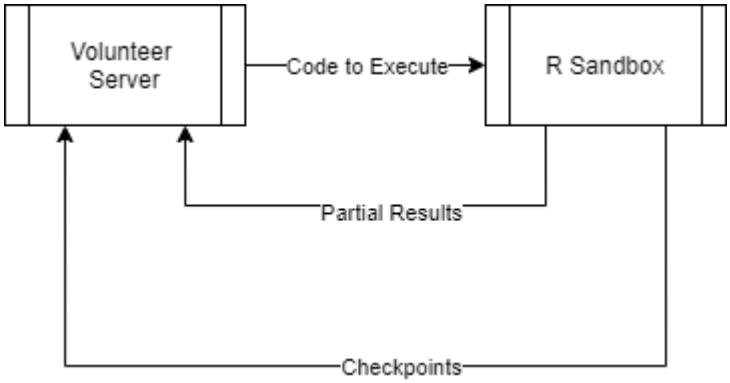

Figure 14 - How the computing works on a volunteer

It is important to ensure that the execution of this computation is non-blocking. The execution of this computation never blocks the volunteer or compromises the volunteer's security. For this, it was decided that the work computation was done in a sandbox controlled by the volunteer, Fig. 14, which informs the sandbox of the resources it needs to perform the computation and starts it. In this way, if there is malicious or blocking code, the volunteer process is not affected. The sandbox guarantees the security of the code you are executing. A code required for partial results and checkpoint is also inserted in the sandbox.

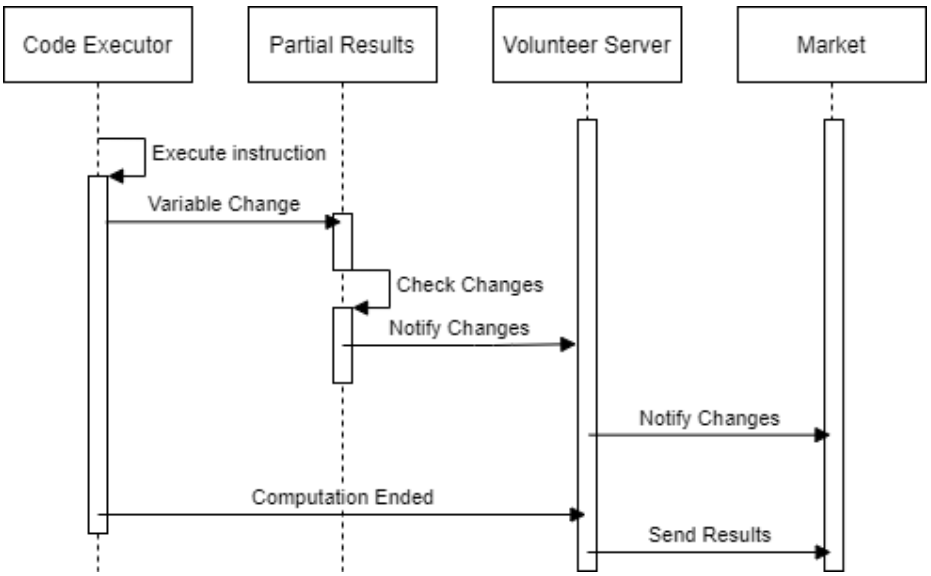

Figure 15 - How partial results work

During computing, it is necessary to track variables that were requested by the Client. This tracking should also not be blocking and create the least possible overload in computing. For this, whenever the Volunteer will start computing in the sandbox, he informs the partial results module which variables are to be observed, whenever one of these variables changes and only when they change, the partial results are responsible for registering this change and for to send to the market and the nearest nodes. As can be seen in Fig. 15, during code execution, when a variable is changed and belongs to the tracking variables, the partial results process is notified and validates the changes made to the variable, after computing the changes, notifies the Volunteer Server warning that there is a new partial result that must be sent over the network, the Volunteer Server sends it immediately to the market and later in synchronization processes to the nodes to which it is connected or to the SN to distribute.

The creation of checkpoints should also not be blocked. For this, there is also a process that is informed when it is intended to run a checkpoint responsible for creating a checkpoint with information about the state of the environment and the point where the execution goes to the send asynchronously to the market and close nodes.

3.2.3 Market

The fact that there is an exchange of credits for a job makes it necessary to control the transactions and manage and monitor the client's jobs. Since the network is super volatile and it would be difficult to guarantee security in a network where nodes are always connecting and disconnecting, it was decided that it would be better to have a market entity responsible for managing the work circulating between Client and Volunteer. So, the market must manage several features:

- Authentication & Registration
- Credit control
- Management and monitoring of nodes in the network
- Job management by the client

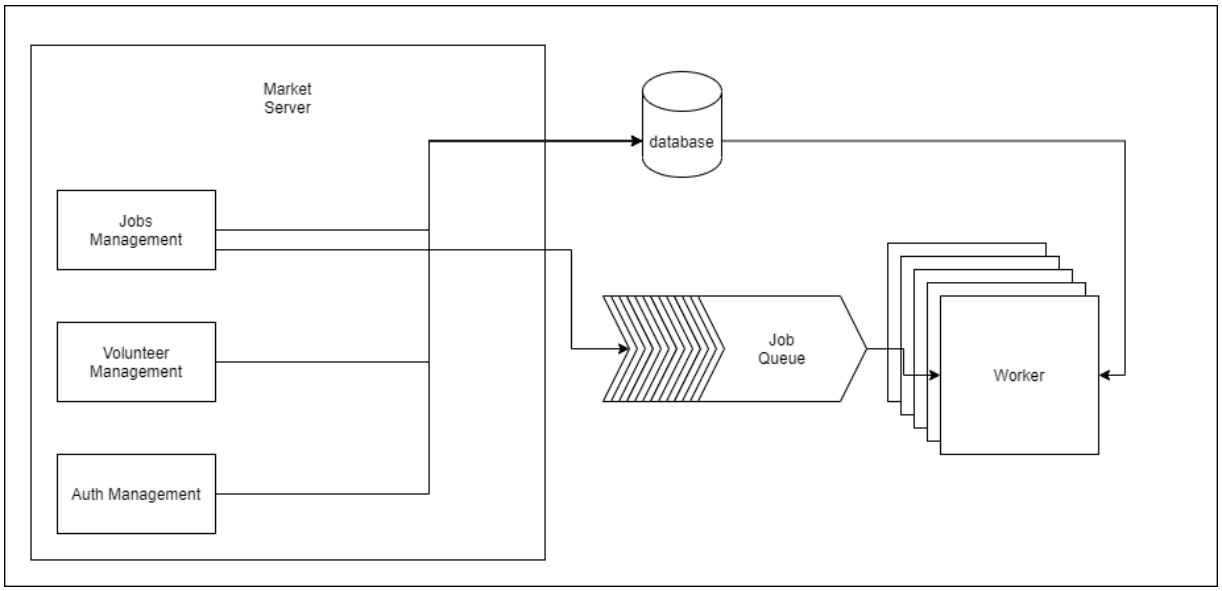

Figure 16 - How Market Works

Fig. 16 shows the architecture resulting from the market. A Client/Volunteer can create an account and authenticate using the marketplace, which validates the account data and issues a temporary session token to communicate with the market. In the voluntary case, it can then associate several machines to your account, whenever it starts a voluntary node, it chooses the associated machine, and the market issues a token for the machine to use during communication between the market and another token that serves for the other nodes to validate the reliability of the volunteer, all these tokens expire.

There is not much in-depth credit management in the market. The market keeps the credit information for each account and carries out credit transactions when a job is done. Some modules made from previous work were reused and adapted and improved to the market.

Being a P2P network, it was assumed that it would be quite volatile, the Market is responsible for monitoring a Volunteer, if he stays from the network to create a metric that classifies the node in the future, Volunteer data is also kept every minute and saved in a time series table, to unlock predictions about the node in the future to assist in the decision of the node for a certain job.

Finally, the flow of data and orders for a job is also managed by the Market. The Market is responsible for receiving the client's orders, saving the information necessary for the remote execution of the job, saving the information of partial results, and finally save the final state of the computation so that the Client can request this information from the market at any time if none of the nodes does have it available.

In addition to the Market, there is another important entity that is the Worker. The Worker maintains the responsibility of granting that a job is assigned to a node. All the information about the computation that a node needs to perform the job is sent. The job is completed, either successfully or in error.

3.2.4 Network

The goal of having a network is essential to use the resources that exist for its organization. One of the serious problems of a P2P network, which is no exception, is the entry and exit of nodes and the way the network is organized. Since our goal is to group nodes to share resources based on a metric such as available BW, we chose to create a structured network.

Another objective was to remove some load from the Market regarding the control of nodes and their state. If there are 1000 nodes, the Market does not have to receive 1000 pings every second to inform its status. We chose to create SNs in the network responsible for grouping and managing the remaining nodes to create small clusters of nodes that have a strong connection between them and allow the sharing of partial results, checkpoints, and PC of closest nodes. These SNs are nodes that have demonstrated stability in the network and that have a set of resources that the market considers relevant for an SN. With SNs, we reduced the market load. If in 1000 nodes, 10 are SNs, the market only receives information from the network of 10 orders and not 1000. This group of SNs grows together with the network. The SNs themselves do not have much of a burden on monitoring and requests received from the nodes it controls.

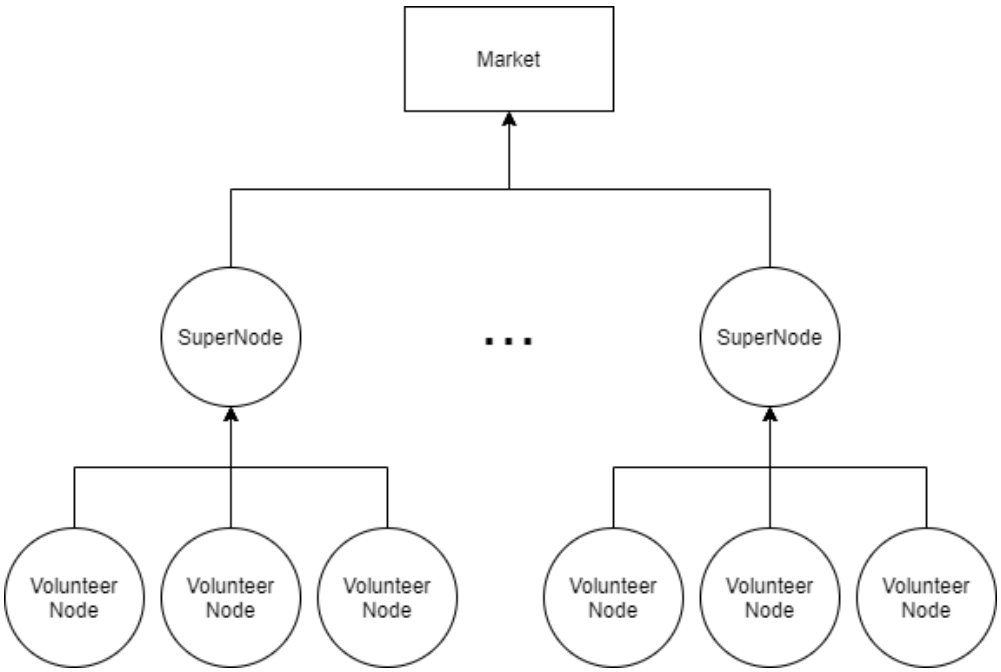

Figure 17 - Structured Network with Super Nodes

Therefore, we opted for a structured network represented in Fig. 17 with SNs that aims to reduce the market load using the SNs to obtain the nodes' status that it manages and inform the market and improve the grouping of nodes. To improve a computation, either by sharing checkpoints for when a node fails, another node in that cluster is immediately ready to resume computing, or by sharing the computing when executing tasks that can be done in parallel.

The network is organized in a hierarchical form of the market, SNs, and nodes. An SN is a node with a very good market confidence metric so that each SN can manage the remaining nodes. The use of a raft adaptation serves to make it possible to manage the network of SNs and their communication so that a node can be in one of the following states:

- Leader when the node leads the SN network
- Candidate when a node does not obtain contacts from a leader and applies for a leader
- Follower normal state of an SN that follows a leader
- State of a node that does not even belong to the SN network but must respond to requests from SNs

So, the cycle of a node can be Node -> Follower -> Candidate -> Leader. The Follower network can also demote a Leader.

In the Node state, a volunteer tries to enter the network, informing the market, the in-form market then the leader who must find an SN to add the new node to the network. In turn, the leader asks each SN why it should stay with this node using a metric such as the network band between the SN and the new node to be added. The leader receiving the metric chooses which SN the new node will be assigned to and informs the SN that the new node must be added. The SN, in turn, informs the node that it will be your SN, and the node is added to the network. A node that is no longer able to communicate with its SN asks the market again to enter the network, maintaining the computations it has to run.

The leader is responsible for managing the network and ensuring that the number of SNs follows the stipulated. Hence, the leader has a timer in which he scrolls through the list of SNs to check their status. Suppose one of the SNs fails the timer for a certain number of consecutive times. In that case, the leader decides that this SN should be removed from the SN network, starting by informing the other SNs that the network will transition to a new state and asking for validation of SNs if the number of positive acknowledges corresponds to the majority, the SN that does not respond to the leader is removed and added to the block list of nodes that can pass to SNs to ensure that the same node is not added back to the network SNs in the next election if the leader does not receive the number of approvals necessary to remove the node from the network, it is assumed that the leader has a network failure that does not allow him to communicate with the SN he wants to remove and for this, he withdraws as the leader. He becomes a normal node, with the former leader also being added to the temporary blocking list of the election of SNs. In case of a tie, the vote of the leader counts as two. Suppose the SN network does not have the SNs amount. In that case, we wish the leader also initiates a request to each SN to suggest a new node that wants to be added to the SN network and not on the temporary block list. Each SN suggests the best node in the list of nodes it manages, according to the metric used to describe a node. Then the leader chooses the best node from those proposed informing the network that a new node will be added if the majority of the network approves the entry of the new node in the SN list, the node transitions to SN if it does not approve this node is added to the temporary block list, and the process is restarted.

The choice of a leader follows the same logic as Raft, a node that does not receive heartbeats from a leader after a certain time decides to change its status to the candidate and decides to ask the network for votes, in this case, the SNs, if it has the most of them emerge as a leader and begin to manage the network if they do not receive enough votes, they return to follower status. Whenever a candidate receives enough votes, he also asks the market for authorization to proclaim himself as a leader. The market is used to make decisions when there are network consensus problems. When there is only one node in the network, the market chooses that node as a leader, or when a new leader tries to immerse itself, and the market knows that there is currently another leader.

If the leader continues to exist, but this SN has just ceased to communicate with the leader. Eventually, the leader will remove the node from the network, and the next time this node asks for votes, the other SNs will inform him that he does not belong to the network.

4 Implementation

It was defined during implementation that the system should be scalable, independent, resilient, and realistic. Therefore, as can be seen in Fig. 18, it was decided an approach that allowed a client to be at the same time a volunteer through a single R library.

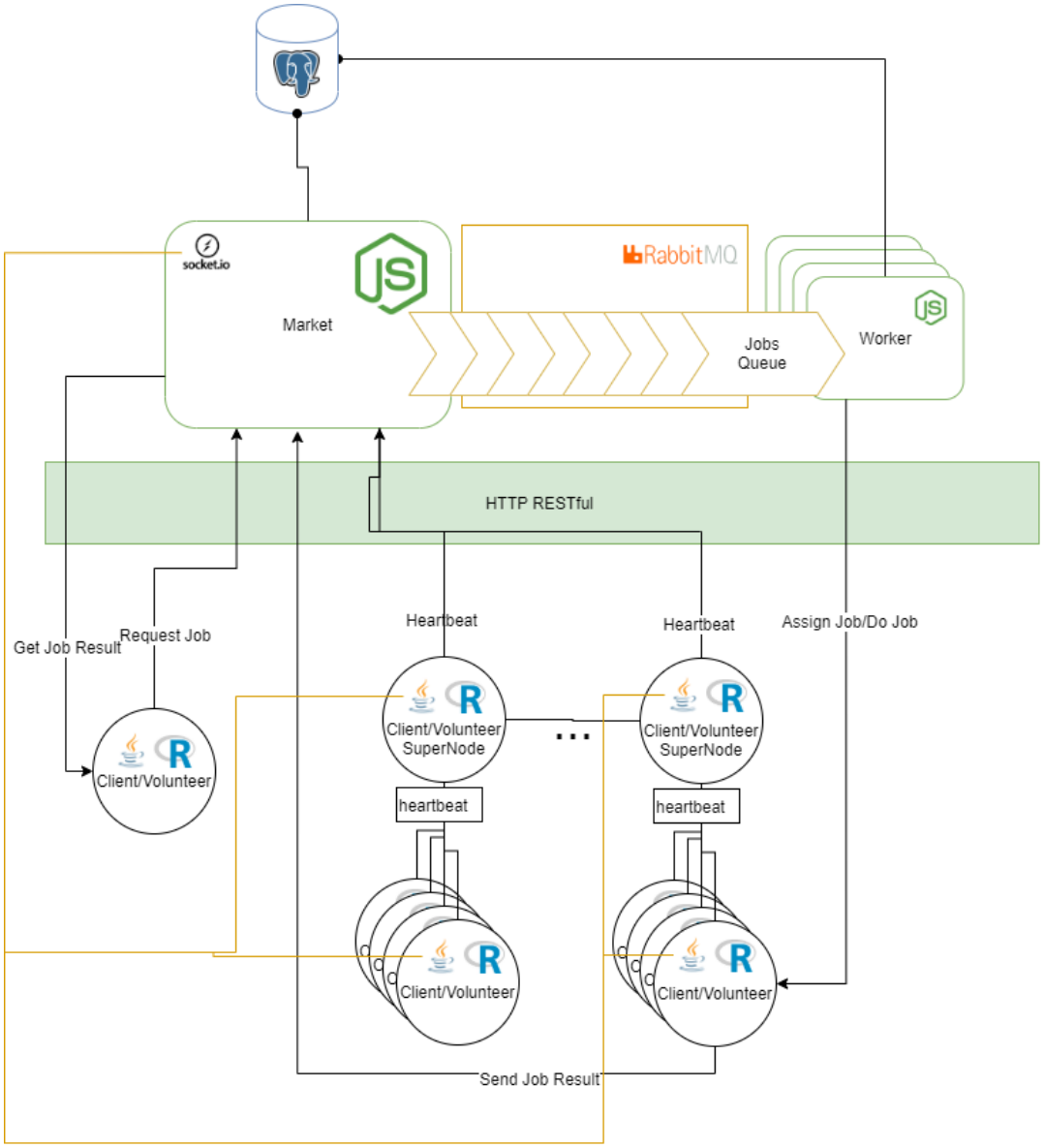

Figure 18 - System's implementation

The Market would be a service that exposes a REST API [63] that allows the Client/Volunteer to communicate with him for everything related to remote execution. It was decided to implement the market in node.js [64] for the speed, scalability, and easy maintenance of REST APIs.

The worker was also implemented in node.js, with a queue in RabbitMQ [65] to manage jobs due to its scalability and fault tolerance. As a database, it was decided to the relational database PostgreSQL [66] that also allows at the same time to have time series tables with a TimescaleDB [67] extension that that transforms a normal PostgreSQL table into a time series table called hypertable. Finally, the management of the P2P network was also very important. At first, we tried to use R to carry out this management using an adaptation of the Raft algorithm to our needs. Still, it ended up being a very difficult task due to the low multi-threading capacity of R. So, we looked for a solution that would allow us to have a good threading capacity but at the same time a good communication between the Market and R itself. The best solution turned out to be Java using the rJava [68] library, which allows calling Java code in R. For the communication between the nodes written in Java, the Java RMI [81] protocol was used and finally for the communication between a node and the Client, socket.io [80] was used. In the next sections, we will explain how each of these modules was implemented and what features can be used in each one.

4.1 Client

The client was implemented in R and is a library that has a set of functions to allow the client's use of the network for his computations. RemotIST controls and manages the session information and data of the requested jobs. The library implemented for the Client consists of a set of functions that allow the Client to communicate with the Market using the REST protocol.

```
Functions
  addJob                function (name, exec_file, cpu, disc, ram, price, dead…
  listJobs              function (page = 1)
  loadJob               function (id, envir = .GlobalEnv)
  loadPartialResult     function (jobId, partialVar, loadToVar = NULL)
  loadPartialResultHi… function (jobId, partialVar, loadToVar = NULL)
  login                 function (email, pw)
  logout                function ()
  register              function (email, pw)
  viewJob               function (id)
```

Figure 19 - R functions for Client

In Fig. 19, we can see the set of functions that the client can call. The function of each one of them is:

register <- function (email, pw)
Allows the creation of an account indicating the email and password.

login <- function (email, pw)
Allows the login of an already registered account by providing the email and password.

logout <- function ()
Allows the client to log out of the session.

addJob <- function (name, exec_file, cpu, disc, ram, price, deadline, folder_path, partialResultsVars = c ())
It allows the creation of a Job that is the remote execution indicating optional variables about the volunteer who can execute the Job such as CPU, disc, and RAM, indicating the maximum credits the client is willing to pay and the deadline until the job can be executed and for end a list of variables that must be tracked for partial results, at the end of the function execution the client will be able to see a table in his environment R with the created job and its identifier(id).

viewJob <- function (id)
Allows you to obtain information about a job from the job's id.

listJobs <- function (page = 1)
Allows you to list client jobs per page.

loadJob <- function (id, envir = .GlobalEnv)
Allows the client to upload the result of a job to an environment, by default GlobalEnv.

loadPartialResult <- function (jobId, partialVar, loadToVar = NULL)
Allows loading the last partial result of a job variable to a local variable.

loadPartialResultHistory <- function (jobId, partialVar, loadToVar = NULL)
It allows loading the history of a variable of partial results of a job to a variable in the client's local environment, which makes it have a variable locally with a list of the different values of the variable that was being tracked in the partial results.

4.2 Market

The Market is composed of several components, the first of which is the Market server that has the function of interacting with the Client, the second component is the database that stores Jobs' information, accounts, partial results, and node information, the third component is the queue of Jobs to be solved, and finally, the last component is the worker whose function is to control the Jobs that come from the queue and distribute it to the Volunteers on the network.

4.2.1 Market Server

The Market server written in node.js is composed of a REST server that provides the endpoints shown in Fig. 20. Most of the configurations can be changed in the file .env like DB config, network config, etc.

Figure 20 - Market endpoints

Besides, to be a REST API, it has two modules:

- Job Manager: this module is notified when a job creation request appears in the REST API. This module ensures that the job is inserted in the queue so that an available worker can handle it
- Peer Manager: this module implements socket.io to communicate with volunteer nodes. It is here that a new node requests to enter the network, and almost always the communication between the Market server are made or by the leader of the network or SNs, a common node only notifies the market when it wants to enter. You can see the communication interfaces between market and peer in Fig. 21, where PEER is the volunteer's inter-faces, SUPER_NODES_SIZE the maximum number of SNs, and MAR-KET the communication interfaces of the market.

```
const PEER = {
    HELLO: "HELLO",
    SET_LEADER: 'SET_LEADER',
    MARKET_PING: 'MARKET_PING',
    PING: 'P2P_PING',
    SET_NODE: 'SET_NODE',
    NEW_PEER: 'NEW_PEER',
    REQUEST_JOB_ASSIGN: 'REQUEST_JOB_ASSIGN',
    SEND_SESSION_TOKEN: 'SEND_SESSION_TOKEN',
    LEADER_LIVE_PROOF: 'LEADER_LIFE_PROOF'
};
const SUPER_NODES_SIZE = 5;
const MARKET = {
    LEADER_HZ: "LEADER_HZ",
    SET_LEADER: 'SET_LEADER',
    PEER_CONNECTION: 'PEER_CONNECTION',
    PONG: 'P2P_PONG',
    JOB_ASSIGNED: 'JOB_ASSIGNED',
    SEND_PEERS_STATE: 'SEND_PEERS_STATE',
    LEADER_LIVE_PROOF: 'LEADER_LIFE_PROOF',
    REQUEST_LEADER_EMERGE: 'REQUEST_LEADER_EMERGE'
};
```

Figure 21 - Market and Peer interfaces

4.2.2 Database

The database chosen was PostgreSQL, for being one relational database that can have some time series tables with the TimescaleDB extension. Although PostgreSQL itself is already quite enough, it was considered that a huge amount of data on the volunteers' historical status could be stored in the future. For this, it was decided to convert the table that stores the nodes' information into a time series table using TimescaleDB, which transforms a PGSQL table into a time series table. In Fig. 23, you can see the relational model of the database created, and in Fig. 22, an example of a continuous aggregate view created in the time series table allows querying thousands of aggregated records per time in a few milliseconds.

```
CREATE  VIEW public.volunteer_beats_per_minute
WITH (timescaledb.continuous) AS
 SELECT time_bucket(INTERVAL '1 minute', timestamp) as bucket,
    hearthz.machine_id,
    hearthz.session_id,
    count(*) AS number,
    avg("CPU") AS cpu,
    avg(disc)AS disc,
    avg("RAM") AS ram,
    avg(network) AS network
   FROM hearthz
  GROUP BY bucket, hearthz.session_id, hearthz.machine_id;
```

Figure 22 - continuous aggregate view created with TimescaleDB

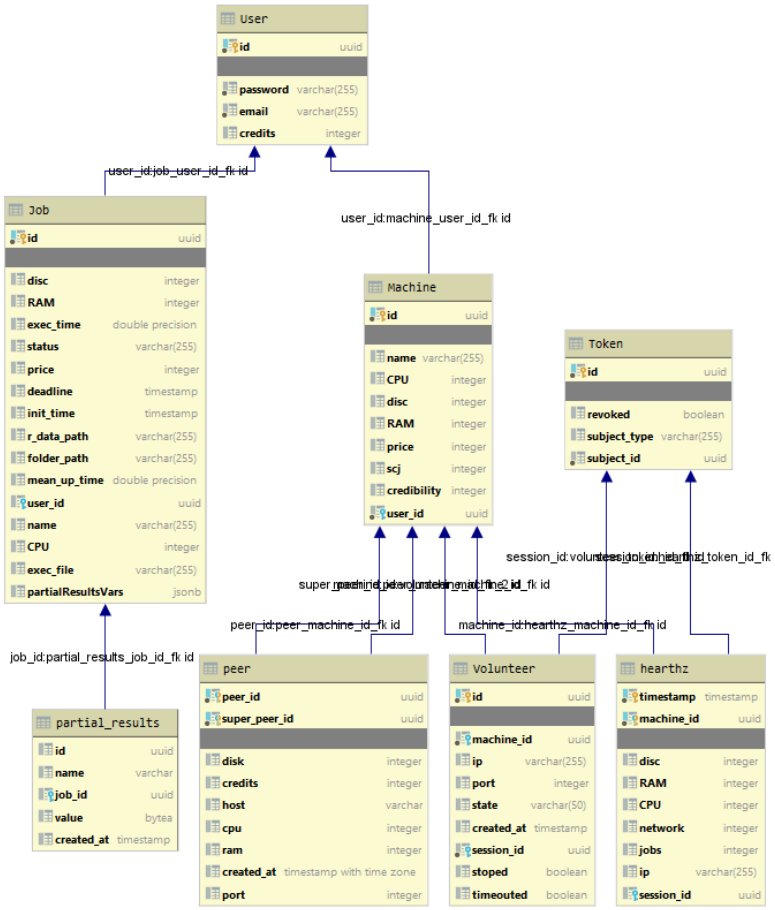

Figure 23 - Relational Model of Market's database

4.2.3 Queue and Worker

A queue is an ordered list of items where the first item entering the list is the first item. The queue was implemented using RabbitMQ due to its fault tolerance. A Worker consumes this queue to ensure that each worker handles a job and that job is never lost. Can-and should-have more than one worker to consume the queue. Worker, implemented in node.js, consumes the queue using AMQP[69]. Whenever the worker receives a new item from the queue, he chooses a volunteer who matches and transfers the necessary files to run the job. When the job ends, the worker acknowledges that the item has already been completed. If it is impossible to find a volunteer, the worker sends the job to the end of the queue.

4.3 Volunteer

The volunteer implementation was focused on three important points:

- Communication interfaces to be used in the R library
- Creation of a volunteer server
- Communication with Java code

For the communication in R, it was decided that a client could also be a volunteer. Therefore it would be important that the same library created for the client also had the volunteer interfaces. Thus, it was decided that the volunteer's logic should be implemented as an addition to the Client's library and thus sharing the same authentication system. The library thus adds interfaces for managing volunteer machines and interfaces for starting the volunteer, so the functions added to the library were:

addMachine <- function (name, cpu, disc, ram, price)
Adds a machine to the volunteer account

viewMachine <- function (id)
Allows to view the data of the machine added as CPU, disk, RAM, and price

listMachines <- function (page = 1)
Allows you to list the volunteer machines

startVolunteer <- function (machineId, port)
Starts the process that marks a machine as a machine available for vol-untitled computing on the network

setVolunteerState <- function (state)
A private function used to mark the machine's status, mainly used to mark the machine as busy/free

sendVolunteerComputingError <- function (id, err)
A private function used when a computation gives an error

startJava <- function (token, port, machineId, machineToken)
A private function used to start the network module

stopJava <- function ()
A private function used to terminate the network module

The startVolunteer function also creates a REST server that exposes some endpoints for the communication between the volunteer, and a worker, the endpoints that the worker can use are:

@post /addJob
Used to inform the volunteer who has been assigned a computation

@get /healthz
Used for the worker to be able to validate that the volunteer is reachable

@get /result/<id>/<var>
Used to obtain the current status of a variable in each computation that is taking place

@post /data/upload/<id>
Used for the worker to send the data needed for computing

@post /code/upload/<id>

Used for the worker to send the Code that is necessary to run in the computation

@post /run/<id>/<main>

Used for the worker to indicate to the volunteer, which is the file to start the computation, thus allowing the code to be broken by several files

Finally, the startJava and stopJava functions are used to communicate with the network module that uses a lib called rJava. Fig. 24 shows the example that performs the start and stop function.

```
obj <- .jnew("Main")
//start java conn module
s <- .jcall(obj, returnSig = "V", "main", .jarray(c(token, port, machineId, machineToken)) )

.remotIST$JavaMachine <- obj

//stop java conn module
s <- .jcall(.remotIST$JavaMachine, returnSig = "V", "stop")
```

Figure 24 - Communication example of R with Java

Thus, the volunteer who can also be a Client uses the same library that reuses the authentication part, creates a REST server to communicate with the worker, and finally creates a bridge between R and Java for the network module, which will be explained in the next chapter.

4.4 Network

The network module was implemented in Java, one of the important points in the communication between nodes. For this Java RMI, it was decided to create three interfaces that extend from the Remote Interface that allows the RMI communication. These three interfaces are:

- Raft - to implement the functions of the raft algorithm
- ServerMembership - for managing the network by obtaining superPeers, adding a new node to the network, removing a node, or even blocking a node.
- ServerRMI - to manage the state of the node

There is also a connection between the market and the node using a Socket.io client for Java that allows the node to communicate with the market and vice versa.

The Raft implementation in java implements the state information necessary for Raft like currentTerm, votedFor, log, commitIndex, lastApplied, nextIndex, and matchIndex and the two most important functions in Raft:

- Pair<Long, Boolean> appendEntriesRPC(long term, String leaderId, long prevLogIndex, ArrayList<String> entries, long leaderCommit) throws RemoteException;
- Pair<Long, Boolean> requestVoteRPC(long term, String candidateId, long lastLogIndex, long lastLogTerm) throws RemoteException;

The first one is invoked by the Leader to replicate the log and to send heartbeats. A Candidate uses the second to gather votes.

The Server Membership is an addition that controls our network. It receives messages from the Market through the Socket.io connection and spreads the information to the other nodes using Java RMI.

5 Evaluation

This chapter will evaluate how our market works, partial results, and remote computations on the network.

It is important that the market can be scalable and guarantee good response times. For that, we will start by evaluating:

1. How the market behaves with a lot of information in the database
2. How the market behaves with many workers
3. Market storage of partial results
4. Computing overhead

The first point is to test the obtaining of the list of available nodes and their current data and past statistics. For that, we will test the behavior with 100k rows, 1M rows, and 10M rows. The second point is the scalable part of the market. Here we intend to test the market response times for many jobs based on the available workers. We intend to test for 1k, 10k, and 100k Jobs and 5, 10, 15 workers. The third point corresponds to the test of passing results from the market to the Client, here we will test how the market behaves with 1k, 10k, and 100k Clients asking for results simultaneously. In the last point, we intend to test the difference between using the network or running locally for each of the indicated test scripts.

It is important to understand the information gain in the partial results compared to the added computation added. It is intended to understand the addition to the computation time when there are 1, 5, and 10 partial result variables to be tracked so that the client can later visualize during the computation.

Finally, it is intended to evaluate the network and its behavior with the entry and exit of nodes, so it is important to evaluate:

- Reaction time for node entry in the network
- Time to choose an SN
- Time to elect a leader

In the first point, we intend to identify how long the network takes to detect a node entrance and to complete and a node exit. A network with ten nodes and 2, 3, and 4 SNs will be used for this case. In the second point, we intend to understand the times of election of an SN based on a network with ten nodes and 2, 3, and 4 SNs. Finally, we intend to understand how long it takes for SNs to find a new leader in a network with ten nodes and 2, 3, and 4 SNs.

R provides good packages to make a benchmark of running applications like rbenchmark and microbenchmark. To evaluate the project, we used different computations for the tests such as:

- Fibonnaci calculation
- Prime factorization
- Knn model

All tests were made on a server with 2CPUs and 4GB RAM and one personal computer with CPU Intel Core i5-8250U 1.60GHz, 8GB RAM, and SSD. Database and RabbitMQ are running using docker containers. We will have the Database, RabbitMQ Queue, Market Server, and Workers running on the server and the Clients and Volunteers running on the personal computer.

5.1 Timeseries Table vs. Normal Table

To evaluate using a time series table or a normal table for the peers' statistics, we used two types of tables with similar queries. The goal was to understand how the system would behave with a lot of data and how time series could help.

	100k	1M	10M
HYPERTABLE	Planning Time: 1.267 ms Execution Time: 0.260 ms	Planning Time: 0.099 ms Execution Time: 1.499 ms	Planning Time: 0.124 ms Execution Time: 19.020 ms
TABLE	Planning Time: 0.083 ms Execution Time: 0.397 ms	Planning Time: 1.397 ms Execution Time: 2179.911 ms	Planning Time: 9.790 ms Execution Time: 17129.179 ms

Table 1 - Hypertable vs. normal table

In Tab. 1, we can find similar queries on tables with 100k, 1M, and 10M rows. Using a normal table until 1M was doable without a time series table. However, if we start growing more up to 10M rows using a normal table can be a bottleneck, and find statistics about one peer can take

17s. Using a time series table, the times of execution for 10M rows are 19ms, which are more acceptable for a scalable platform. Based on the results of table 1, we can conclude that using a time series table to handle big amounts of data about the peers on the network is a good and scalable approach.

5.2 Market

To evaluate the market, we used a script that would create 1k, 10k, and 100k jobs simultaneously. To understand how it would behave, we used the RabbitMQ monitoring system.

Figure 25 - RabbitMQ monitoring results

Fig. 25 shows the monitoring graphics of RabbitMQ. The red one is the number of queue messages waiting to be handled. The green one is the rate of messages by each

consumer/worker (for example, if you have five consumers and 16k/s, your system is handling 16*5k messages per second, 80k/s). In the green one, we cal also see three groups of 3 spikes. The first spike is for 100k jobs, the second one for 10k jobs, and the last small one for 1k jobs. In both red and green graphs, you have three groups, the first group (closer to the left) is the results using 15 workers, the middle one using ten workers, and the right one using five workers. Increasing the number of workers shows that we can keep our system running without failing any tasks and keep it scalable. Even with five workers, the system can handle 100k jobs in a few milliseconds. We can also see that increasing the number of workers improves the system and his fault tolerance.

Another important measure is to understand how the market behaves when having multiple requests of the clients. To test these behaviors, we used a script to simulate a Client call 1k, 10k, and 100k times to get a partial result for a big variable and measured the average response time.

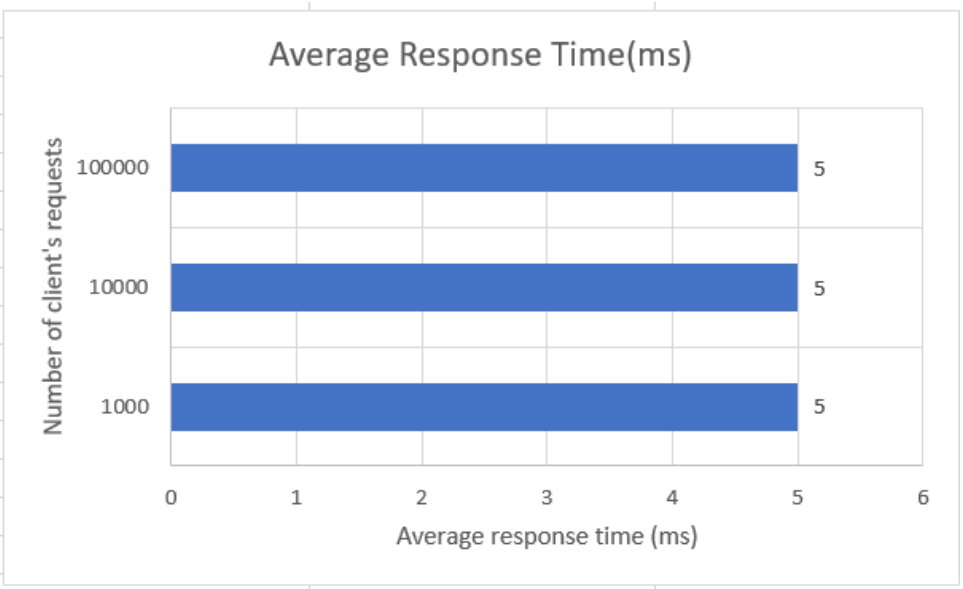

Figure 26 - Market's average response time to a partial result call

The market showed up a good performance to handle up to 100k requests, as shown in Fig. 26, without any failure or increase in the average response time. This test was only made with one market, and we can always add more instances of the market to make it more scalable.

Finally, it was important to understand our system's overhead comparing the local running against running in the volunteer network. To evaluate this, we used some scripts of a Fibonacci calculation, prime factorization, KNN [70] model calculation, and Fibonacci with system sleep.

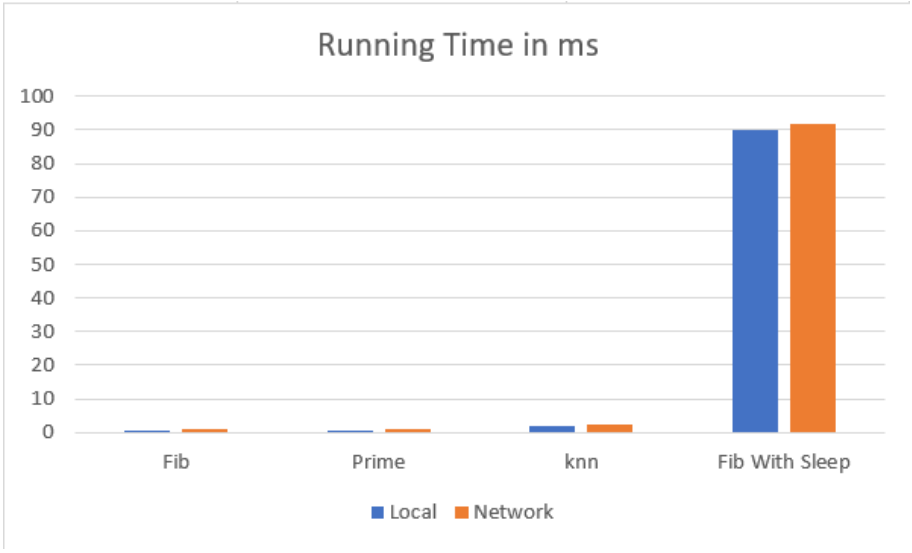

Figure 27 - Script running this in local and using volunteer network in ms

Fig. 27 shows that the overhead is a continuous value of around 2-6ms. Fibonacci with sleep showed up this behavior more because, in the other script, this 2-6ms represented about 50% increase of time. Still, we can see that if you have a more expensive computation, the RemotIST's overhead is around 2-6ms, making it useful for heavy computations.

5.3 Partial Results

To evaluate the partial results, we decided to run the scripts using 5, 10, and 15 partial results variables to track. Fibonacci with sleep also had more than 100 mutations compared to the Fibonacci normal script.

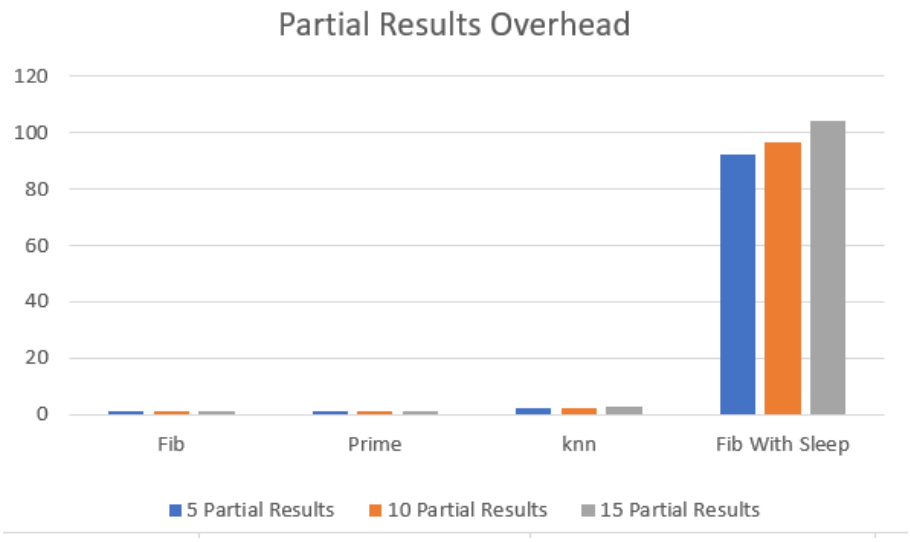

Figure 28 - Partial Results Overhead in ms

Fig. 28 shows that the overhead of tracking a partial result is about 1ms per tracking variable. This value can be explained because all the tracking is made parallel to not increase too much the overhead of normal computing. Even for Fib With Sleep that had some extra mutations, the results showed that each variable's increase to be tracked doesn't add too much overhead to the running code.

5.4 Network

To evaluate the network behavior, we configured the system to have 2,3, and 4 SNs. Then we started ten nodes. During the start, it would already select the leader and the SNs, but we want to evaluate the system on running and not on start. After having the ten nodes, we created one more to evaluate the new entry of a node. We crashed an SN to evaluate the reaction of the network to choose a new SN. Finally, we crashed the leader to evaluate the reaction of the SNs to the election of a new leader.

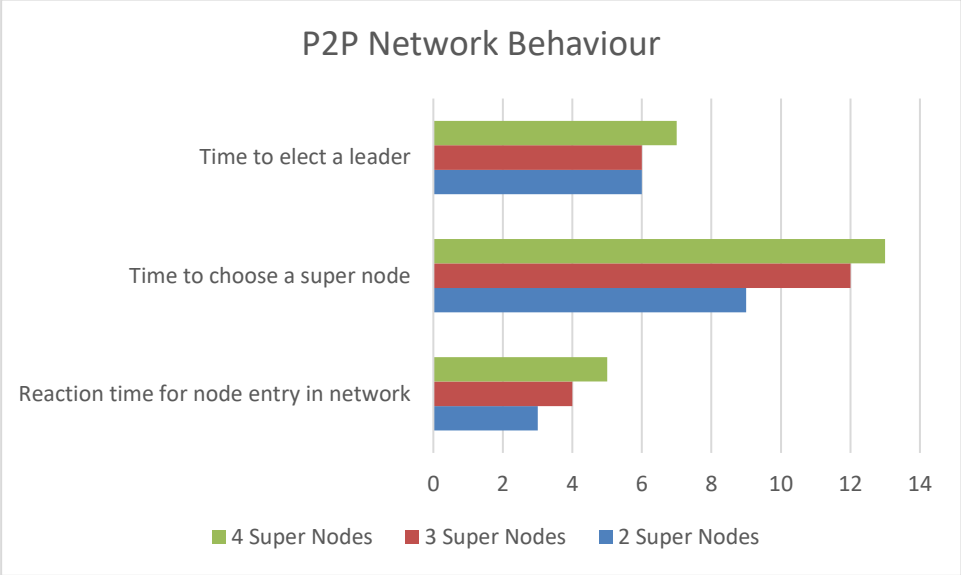

Figure 29 - Network running behavior evaluation in ms

Fig. 29 shows the results for 2, 4, and 4 SNs to the proposed tests. We can see that increasing the number of SNs increases the time to choose an SN and the new node's reaction. This is expected and should be a concern to limit the number of SNs because to add a new node or choose a new SN, the leader needs the feedback of every other SN, having more SNs can increase this time, even if the call is made in parallel (they already are). The leader election time will also increase but less than the others because the others will need all SNs' feedback to choose the SN to control the new node or the node to be promoted to SN. The leader election will only need the majority feedback of the SNs to elect a new leader.

6 Conclusion

VC makes it possible to have idle resources being used to help the ones who need them. RemotIST is a VC system applied to the R community to support their big computations. Like other available systems, RemotIST has some problems related to the loss of computation, resource usage, and providing more information to the clients. It was possible to see how other platforms use some methodologies to make partial results using a parallel process to deal with it, to select storage hosts to make checkpoint and how to make it possible in a way that making a checkpoint doesn't have big overhead and at the evolution of the PC, with these new functionalities based on the needs of RemotIST and the available solutions that we studied.

During the analysis of the system's previous implementation, it was found that the system was not ready to proceed with checkpoint and PC because there was no notion of a network. We decided to change our objectives to create the market, make the client and volunteer real, and create a scalable and fault-tolerant network. The checkpoint implementation was started but unfortunately not finished due to the amount of work. However, the system is now ready to have a checkpoint and is super configurable.

The implementation Marked shows good results for scalability and fault tolerance. It was possible to see that having some job logic separated and creating a new independent process called the worker to handle jobs made It possible. With workers now, jobs will not be lost, and even if a worker crashes, you have more workers to handle the jobs and keep the market running. The market also showed up that it can handle a considerate number of requests without failing. One of the important features that also showed good results was the partial results. Using a parallel process to handle it showed us that it doesn't have much overhead to the system, making it doable. The network also showed fault-tolerant. It can handle node leaves. The node enters and crashes, keeping the consensus of the network.

Overall, the system showed some signs that it is on a good path to make it possible to be used one day as an R package by everyone using R to use a cycle-sharing P2P network.

6.1 Future Work

Since we could not keep the established objectives during the first phase of the project, it is necessary to give a future to all the effort made to build the system. For future work, we a few major topics.

Credits Management System

Create the logic for the credits and metrics, such as when the client stops the computation or runs in parallel or uses other volunteers to store checkpoints.

Finish Checkpoint Implementation

We were not able to finish the checkpoint implementation. Still, we did good research about how to do it in R. We were able to split the computations and choose if we want to run the next computation or if we want to run a checkpoint. We have a system that stores the environment with the current variables of it. With the information of the last computation and the last environment snapshot, it is possible to load the environment to another volunteer and proceed with the computation where it had stopped. Furthermore, it is also possible to use the network to spread the checkpoints among a group of closer nodes.

Use network groups to run parallel jobs

The network is now ready to create a group of closer nodes. The communication with R and Java is already provided. For the future, it is a good idea to start analyzing the code before each computation to understand if we can run some jobs in parallel and more than one volunteer and to spread huge data to make huge computations possible following a map-reduce approach.

Prediction of P2P nodes

The P2P network is volatile. One good strategy is to detect some peer's patterns to understand if the following volunteer will disconnect during the computation or not have the necessary resources. We already store peer's information in a time series table to allow future work to create a prediction model system for peers.

7 References

1. Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky (2001) SETI@HOME—MASSIVELY DISTRIBUTED COMPUTING FOR SETI
2. May P, Ehrlich H-C, Steinke T (2006) ZIB structure prediction pipeline: composing a complex biological workflow through web services. In: Nagel WE, Walter WV, Lehner W (eds) Euro-Par 2006. LNCS, vol 4128. Springer, Heidelberg, pp 1148–1158. doi:10.1007/11823285_121
3. D.Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. Communications of the ACM, 45:56–61, 2002.
4. Folding@Home Distributed Computing, http://folding.stanford.edu/
5. National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov
6. D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In 5th IEEE/ACM International Workshop on Grid Computing, pages
7. Francisco Banha (2017) Secure Remote Execution for the R Programming Environment
8. Ricardo Wagenmaker (2017) Computational Cost Estimation using Volunteer Computing in R
9. Ricardo Maia (2018) Mercado de computação voluntária para R
10. What is R? https://www.r-project.org/about.html
11. Ali Shoker (2017) Sustainable Blockchain through Proof ofeXercise
12. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. Concurrency - Practice and Experience, 17(2-4):323–356, 2004.
13. W. Gentzsh. Sun Grid Engine: towards creating a compute power grid. In Int. Symposium on Cluster Computing and the Grid, pages 35–39, 2001.
14. Ren, Xiaojuan, Rudolf Eigenmann, and Saurabh Bagchi. "Failure-aware checkpointing in fine-grained cycle sharing systems.", 2007.
15. The Impressive Growth of R https://stackoverflow.blog/2017/10/10/impressive-growth-r/
16. How Big Companies Are Using R for Data Analysis https://www.northeastern.edu/levelblog/2017/05/31/big-companies-using-r-data-analysis/
17. Understanding How R is Used in Data Science https://www.datasciencegraduateprograms.com/data-science-with-r/
18. R Related Projects https://www.r-project.org/other-projects.html
19. Bioconductor. https://www.bioconductor.org/

20. Rmetrics Open Source Project. https://www.rmetrics.org/
21. R-forge http://r-forge.r-project.org/
22. Team, R. Core. "R language definition." Vienna, Austria: R foundation for statistical computing (2000).
23. R-source https://github.com/wch/r-source
24. Stages in the Evolution of S http://ect.bell-labs.com/sl/S/history.html
25. What are the differences between R and S? https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-are-the-differences-between-R-and-S_003f
26. W. N. Venables, D. M. Smith and the R Core Team " An Introduction to R" (2018).
27. Schmidberger, Markus, et al. "State-of-the-art in Parallel Computing with R." (2009).
28. Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering 5.1 (1998): 46-55.
29. Barker, Brandon. "Message passing interface (mpi)." Workshop: High Performance Computing on Stampede. Vol. 262. 2015.
30. Yu, Hao. "Rmpi: parallel statistical computing in R." R News 2.2 (2002): 10-14.
31. Knaus, Jochen, et al. "Easier parallel computing in R with snowfall and sfCluster." The R Journal 1.1 (2009)
32. GridR: An R-based gridenabled tool for data analysis in ACGT clinico-genomic trials
33. Ali Raza Butt, Xing Fang, Y. Charlie Hu, and Samuel Midkiff (2014) Java Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharin
34. M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peerto-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002, 2002. h http://research.microsoft.com/˜antr/PAST/ localtion.ps i (17 Oct 2003).
35. Shuo Yang, Ali R. Butt, Y . Charlie Hu and Samuel P . Midkiff (2005) Trust but Verify: Monitoring Remotely Executing Programs for Progress and Correctnes
36. R. L. Rivest. RFC 1321 –MD5 Message-Digest Algorithm, 1992.

37. M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. Journal of the ACM, 36(2):335–348, 1989.
38. X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of resource availability in fine-grained cycle sharing systems and empirical evaluation. To appear in the Journal of Grid Computing, 2007.
39. T.Z. Islam, S. Bagchi and R. Eigenmann, FALCON: A system for reliable checkpoint recovery in shared grid environments, ACM, 2009
40. Boilergrid: http://www.rcac.purdue.edu/boilergrid/
41. Ren X, Lee S, Eigenmann R, and Bagchi S. “Resource Failure Prediction in Fine-Grained Cycle Sharing System,” HPDC ’06, 2006
42. K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributes systems”, ACM, 1985
43. Livny. Jim Managing Checkpoints for Parallel Programs, 1996
44. Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.
45. William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.
46. Sakamoto, Chikara, et al. "Design and implementation of a parallel pthread library (ppl) with parallelism and portability." Systems and Computers in Japan 29.2 (1998): 28-35.
47. Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering 5.1 (1998): 46-55.
48. Yoo, Richard M., Anthony Romano, and Christos Kozyrakis. "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.
49. Ranger, Colby, et al. "Evaluating mapreduce for multi-core and multiprocessor systems." High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. Ieee, 2007.

50. Hadoop: Open source implementation of MapReduce. https://hadoop.apache.org/
51. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM51.1 (2008): 107-113.
52. Costa, Fernando, Luís Veiga, and Paulo Ferreira. "Boinc-Mr: Mapreduce in a volunteer environment." OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Springer, Berlin, Heidelberg, 2012.
53. Chen, Rong, Haibo Chen, and Binyu Zang. "Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling.", 2010.
54. Jiang, Wei, Vignesh T. Ravi, and Gagan Agrawal. "A map-reduce system with an alternate api for multi-core environments." Cluster, Cloud and Grid Computing (CCGrid), 2010.
55. Talbot, Justin, Richard M. Yoo, and Christos Kozyrakis. "Phoenix++: modular MapReduce for shared-memory systems." Proceedings of the second international workshop on MapReduce and its applications. ACM, 2011.
56. Pitt, Esmond, and Kathy McNiff. Java. rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc., 2001.
57. Tierney, Luke, et al. "snow: Simple network of workstations." R package version 0.3-3, URL http://CRAN. R-project. org/package= snow (2008).
58. Biocep-R Team (2010). Biocep-R. https://r-forge.r-project.org/projects/biocep-distrib/
59. Grose (2008). Multi-R
60. "VolunteerComputing – BOINC". boinc.Berkeley.edu. Retrieved November 18, 2017.
61. R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. 1999.
62. Cohen, Jacob Willem, and Anthony Browne. *The single server queue*. Vol. 8. Amsterdam: North-Holland, 1982.
63. Masse, Mark. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.
64. Tilkov, Stefan, and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs." *IEEE Internet Computing* 14.6 (2010): 80-83.

65. Videla, Alvaro, and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.
66. Momjian, Bruce. *PostgreSQL: introduction and concepts*. Vol. 192. New York: Addison-Wesley, 2001.
67. Stefancova, Elena. *Evaluation of the TimescaleDB PostgreSQL Time Series extension*. No. CERN-STUDENTS-Note-2018-137. 2018.
68. Urbanek, Simon. "rJava: Low-level R to Java interface." (2013).
69. Naik, Nitin. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP." *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017.
70. Guo, Gongde, et al. "KNN model-based approach in classification." *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, Berlin, Heidelberg, 2003.
71. Urbanek, Simon. "Rserve--a fast way to provide R functionality to applications." *PROC. OF THE 3RD INTERNATIONAL WORKSHOP ON DISTRIBUTED STATISTICAL COMPUTING (DSC 2003), ISSN 1609-395X, EDS.: KURT HORNIK, FRIEDRICH LEISCH & ACHIM ZEILEIS, 2003 (HTTP://ROSUDA. ORG/RSERVE*. 2003.
72. Ancona, Davide, et al. "RPython: a step towards reconciling dynamically and statically typed OO languages." *Proceedings of the 2007 symposium on Dynamic languages*. 2007.
73. Ren, Wei, and Randal W. Beard. *Distributed consensus in multi-vehicle cooperative control*. Vol. 27. No. 2. London: Springer London, 2008.
74. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
75. Lamport, Leslie. "Paxos made simple." *ACM Sigact News* 32.4 (2001): 18-25.
76. Lamport, Leslie. "Fast paxos." *Distributed Computing* 19.2 (2006): 79-103.
77. Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 2007.

78. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
79. Howard, Heidi, and Richard Mortier. "Paxos vs Raft: Have we reached consensus on distributed consensus?." *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 2020.
80. Rai, Rohit. *Socket. IO Real-time Web Application Development*. Packt Publishing Ltd, 2013.
81. Pitt, Esmond, and Kathy McNiff. *Java. rmi: The remote method invocation guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.