Fault Tolerance Support in an R P2P cycle-sharing system

Tiago Alexandre Serafim Monteiro
tiago.alexandre.monteiro@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
DECEMBER 2020

Abstract

Volunteer computing has the goal of taking advantage of idle computing cycles to use in big computations. Several systems have already explored successfully this possibility. In some of these systems it is possible to be a volunteer and a client and in some only a volunteer for important projects. In this paper, we introduce a new system that takes this advantage of the R language, providing a market to buy/sell remote computation time. This system is focused on the R language to make these remote computations secure and reliable since in R the computations are frequently long. 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, able to provide more information of the remote computations to the clients and using the idle cycles in the network the make these computations faster as possible.

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

1. Introduction

The opportunity to use idle computer cycles has gotten the attention of the computing world due to the big advantages that it gives. 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 [19], 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) [4]. 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 the fact that many users are willing to provide their resources [2][3]. However, in a volunteer network, there is no guarantee that the resources will be available when they are needed. This turns VC into a complex system where choosing the volunteers is critical and managing the resources available is hard. Also, when a volunteer leaves the system with a running computation this means that 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 [5][6][7] 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 [8] is a language and environment for computing and graphics design increasingly used by scientists and data miners for the development of 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 to use the network's available resources. After this, they communicate with the centralized RemotIST server providing the code to be remotely executed and this centralizer server starts looking for donators that 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 and 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, 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.

Currently, 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 status of the running programs 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 big problems with wasted resources. In an R program, it is possible to have computations that are too expensive and if these computations have some buggy code this means that the entire computation is useless for the client and

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

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, give the ability to 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 in R we have huge data associated with the computations that require a lot of memory, this means that saving a variable and provide it to the client when he needs scales with the amount of memory that the variable occupies. Also, by tracking the progress we are adding 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 [9] the host that accepts some job gives a credit to make a security deposit and only recovers it after the computation. Even with this, we can't avoid the departure of the host and this 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 at least to reduce the loss and to 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 having to force 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 to computing expensive computations that were not possible without 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 a lot of 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 only use one host and it may need some policy adjusts. To focus only on the improvement of 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.

2. Related Work

R is a programming language and environment for graphical computation and statistics [8] used to develop statistical software and data analysis. R is one of the fastest-growing languages, has grown incredibly in the last 5 years [12]. Its growth and popularity give R an important role in companies such as Facebook and Google due to its success in the problems it solves [13][14].

A novel method for monitoring the progress of a Java application with low overhead is presented in [15]. They use a Pastry [16] p2p network for cycle-sharing with distributed hash tables (DHTs) and monitoring the execution of the code in each node that has some computation.

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 which consist 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, and resuming the previously running state.

Some systems like Condor [10] and Sun Grid Engine [11] use a dedicated host to keep the checkpoints of the computations to make sure the cost of a loss is as small as possible.

With the presence of various cores in a computer being something normal, the need to take advantage of all these cores in a computer began to gain attraction but 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 [17]. The first development of a multi-core model was the Message Passing Interface (MPI[2])[18].

One of the fundamental primitives for constructing fault-tolerant, strongly consistent distributed systems is distributed consensus [31]. Distributed consensus ensures consensus of data among the nodes of a distributed system to agree on a proposal. A consensus algorithm [32] 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 and you have two possible destinations, the goal of a consensus algorithm is to ensure that you and your family reach a consensus about which destination are you going travel to.

One algorithm implementing consensus is the traditional algorithm Paxos [33,34,35].

Another algorithm, simpler and recent, implementing consensus is Raft [36].

3. Architecture

Based on the previous architecture and with the objectives of system fault tolerance, the system architecture was

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

designed to attend to the previous architecture problems and to 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 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.

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. The Market after receiving this information inserts the work in a Queue [20] 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 the status 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 results from the Market 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.

3.1 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 piece of code using the resources that exist on the network for reasons of better CPU, memory, or simply to run in parallel while the Client is concerned with other computing. The Client must be able to 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.

Something common, 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, is that the code to be executed remotely contains errors. Whenever a computation is interrupted in a volunteer due to an error in the code, the information of the state of the environment is saved, saving the variables and also saving the information of what caused the error so that the Client can consult later. Errors are not always exceptions, many times they can be 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 and that variable whenever it changes 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 there is the possibility of the Client doing 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 at the same time without having the consequences of losing the results of some work. work and the Client can then define which results to load into their environment and only delete the results from the market if they want, they are never automatically deleted.

3.2 Volunteer

A volunteer wants to make his resources available in exchange for credits without compromising the safety of his machine. 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 for the contact of a worker to distribute jobs that meet the parameters established by the Volunteer to accept a job (CPU, RAM, credits, etc.).

It is important to ensure that the execution of this computation is non-blocking, that is, that 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, 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, and 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.

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. 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 that is 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.

3.3 Market

The fact that there is an exchange of credits for a job makes it necessary to control the transactions and to manage and monitor the jobs requested by a client. 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 that would be 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

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 and 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.

Last but not least, 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, that all the information about the computation that a node needs to perform the job is sent, and that the job is completed, either successfully or in error.

3.4 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 this is no exception, is the entry and exit of nodes at any time 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 so that 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 that is responsible for grouping and managing the remaining nodes to create small clusters of nodes that have a strong connection between them and that 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, and if in 1000 nodes, 10 are SNs, the market only receives information from the network of 10 orders and not 1000, and this group of SNs grows together with the network so that the SNs themselves do not have much of a burden on monitoring and requests received from the nodes it controls.

Therefore, we opted for a structured network represented with SNs that aims to reduce the market load using the SNs to obtain the status of the nodes that it manages and inform the market and improve the grouping of nodes. to improve a computation, either by sharing checkpoint for when a node falls 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 that has 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
- Node: 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. A Leader can also be demoted by the Follower network.

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. The leader in turn 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 is following the stipulated, so the leader has a timer in which he scrolls through the list of SNs to check their status. If one of the SNs fails the timer for a certain number of consecutive times, 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 and 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. If the SN network does not have the super number 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 that is not on the temporary block list. Each SN suggests the best node in the list of nodes it is managing, according to the metric used to describe a node, and 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 such as when there is only one node in the network, the market chooses that node as a leader or as 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 be able 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 that he has does not belong to the network.

4. Implementation

During implementation, it was defined that the system should be scalable, independent, resilient and realistic. Therefore, it was decided an approach that allowed a client to be at the same time a volunteer, through a single R library.

The Market would be a service that exposes a REST API [21] 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 [22] for the speed, scalability and easy maintenance of REST APIs.

The worker was also implemented in node.js, with a queue in RabbitMQ [23] for the management of jobs due to its scalability and fault tolerance. As a database, it was decided to the relational database PostgreSQL [24] that also allows at the same time to have time series tables with a TimescaleDB [25] 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. We tried at first to use R to carry out this management using an adaptation of the Raft algorithm to our needs, but 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 [26] library which allows calling Java code in R. For the communication between the nodes written in Java, the Java RMI [38] protocol was used and finally for the communication between a node and the Client, socket.io [39] was used. In the next sections, we will explain in more detail 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.

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. Most of the configurations can be changed in the file .env like DB config, network config etc. 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. We have communitaction interfaces with the market and the peer 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.

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 in the future a huge amount of data on the volunteers' historical status could be stored. For this, it was decided to convert the table that stores the information of the nodes into a time series table using TimescaleDB which transforms a PGSQL table into a time series table.

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 leaving the list. 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[27]. Whenever the worker receives a new item from the queue, he chooses a volunteer who matches and transfers the necessary files for the volunteer to run the job. When the job ends the worker acknowledges that the item has already been completed, if it is not possible to find a volunteer, the worker sends the job to the end of the queue.

4.3. Volunteer

The volunteer implementation was focused on 3 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 and therefore it would be important that the same library created for the client also had the faces for the volunteer. Thus, it was decided that the logic of the volunteer 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 The startVolunteer function also creates a REST server that exposes some endpoints for the communication between the volunteer and a worker.
Finally, the startJava and stopJava functions are used to communicate with the network module that uses a lib called rJava.

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

network module was implemented in Java, being one of the important points the communication between nodes it was decided to use for this Java RMI creating 3 interfaces that extend from the Remote Interface that allows the RMI communication. These 3 interfaces are:

- Raft - to implement the functions of the raft algorithm
- ServerMembership - for managing the network such as 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. The second is used by a Candidate 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

In this chapter, we will evaluate how our market works, partial results, and remote computations on the network. 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 as well as 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 at the same time. In the last point, we intend to test what is the difference between using the network or running locally for each of the indicated test scripts.

In the partial results, it is important to understand the information gain when 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, as well as a node exit. For this case, a network with 10 nodes and 2, 3, and 4 SNs will be used. In the second point, we intend to understand the times of election of an SN based on a network with 10 nodes and 2, 3 and 4 SNs. Finally, we intend to understand how long it takes for SNs to find a new leader also in a network with 10 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 the performance of 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 could time series help with it.

	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 the results of the 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. By 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 at the same time. To understand how it would behave we used the RabbitMQ monitoring system.

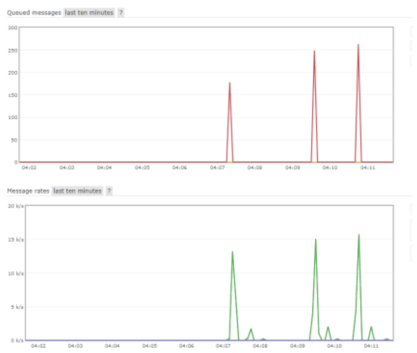

Figure 1 - RabbitMQ monitoring results

Fig. 1 shows the monitoring graphics of RabbitMQ, the red one is the number of queue messages waiting to be handled and the green on is the rate os messages by each consumer/worker (example if you have 5 consumers and you have 16k/s your system is handling 16*5k messages per second which are 80k/s). In the green one, we cal also see 3 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 3 groups, the first group (closer to left) is the results using 15 workers, the middle one using 10 workers and the right one using 5 workers. Increasing the number of workers shows that we can keep our system running without failing any tasks and keep is scalability, even with 5 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.

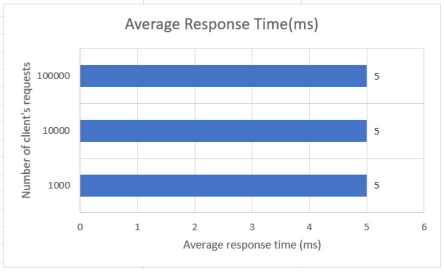

Figure 2 - Market's average response time to a partial result call

The market showed up a good performance to handle up to 100k requests, as showed in Fig. 2, without any failure or any 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 the overhead of our system comparing the local running against running in the volunteer network. To evaluate this, we used some scripts of a Fibonacci calculation, prime factorization, KNN [28] model calculation, and Fibonacci with system sleep.

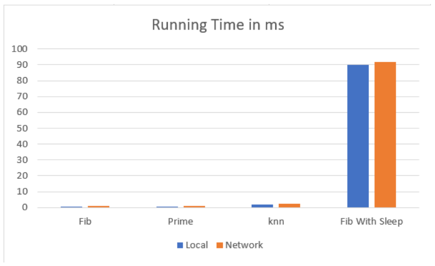

Figure 3 - Script running this in local and using volunteer network in ms

Fig. 3 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 but we can see that if you have a more expensive computation the RemotIST's overhead is around 2-6ms making is really 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.

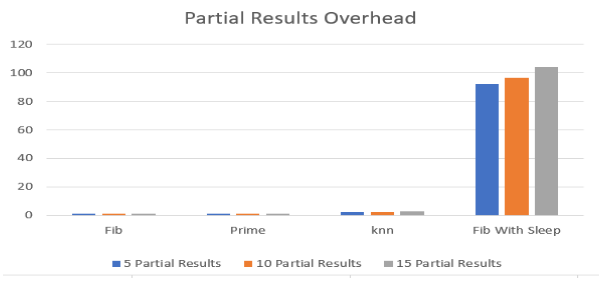

Figure 1 - Partial Results Overhead in ms

Fig. 28 shows that the overhead of tracking a partial result is about 1ms per each tracking variable, this value can be explained because all the tracking is made in parallel to not increase too much the overhead of the normal computing, even for Fib With Sleep that had some extra mutations the results showed that the increase of each variable 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 10 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 havin

g the 10 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 and finally, we crashed the leader to evaluate the reaction of the SNs to the election of a new leader.

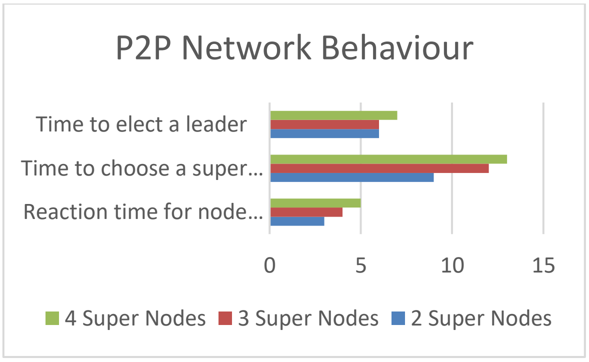

Figure 4 - Network running behavior evaluation in ms

Fig. 4 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 reaction to the new node. 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). For the leader election time, it 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 and 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 using it. 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 previous implementation of the system, 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, to make the client and volunteer real and to create a network that is scalable and fault-tolerant. The checkpoint implementation was started but unfortunately not finished due to the amount of work although the system is now ready to have a checkpoint and is super configurable.

The implemented Marked showed 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 were 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.

Since we were not able to 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 when it runs in parallel or uses other volunteers to store checkpoints.

Finish Checkpoint Implementation: We were not able to finish the checkpoint implementation, but we did however 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 and 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 and one good strategy is to detect some peer's patterns to understand if the following volunteer will disconnect during the computation or if it will not have the necessary resources for it. 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. 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.
3. Folding@Home Distributed Computing, http://folding.stanford.edu/
4. D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In 5th IEEE/ACM International Workshop on Grid Computing, pages
5. Francisco Banha (2017) Secure Remote Execution for the R Programming Environment
6. Ricardo Wagenmaker (2017) Computational Cost Estimation using Volunteer Computing in R
7. Ricardo Maia (2018) Mercado de computação voluntária para R
8. What is R? https://www.r-project.org/about.html
9. Ali Shoker (2017) Sustainable Blockchain through Proof ofeXercise
10. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. Concurrency - Practice and Experience, 17(2-4):323–356, 2004.
11. W. Gentzsh. Sun Grid Engine: towards creating a compute power grid. In Int. Symposium on Cluster Computing and the Grid, pages 35–39, 2001.
12. The Impressive Growth of R https://stackoverflow.blog/2017/10/10/impressive-growth-r/
13. How Big Companies Are Using R for Data Analysis https://www.northeastern.edu/levelblog/2017/05/31/big-companies-using-r-data-analysis/
14. Understanding How R is Used in Data Science https://www.datasciencegraduateprograms.com/data-science-with-r/
15. Ali Raza Butt, Xing Fang, Y. Charlie Hu, and Samuel Midkiff (2014) Java Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharin
16. 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).
17. Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

18. William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.
19. R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. 1999.
20. Cohen, Jacob Willem, and Anthony Browne. *The single server queue*. Vol. 8. Amsterdam: North-Holland, 1982.
21. Masse, Mark. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.
22. Tilkov, Stefan, and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs." *IEEE Internet Computing* 14.6 (2010): 80-83.
23. Videla, Alvaro, and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.
24. Momjian, Bruce. *PostgreSQL: introduction and concepts*. Vol. 192. New York: Addison-Wesley, 2001.
25. Stefancova, Elena. *Evaluation of the TimescaleDB PostgreSQL Time Series extension*. No. CERN-STUDENTS-Note-2018-137. 2018.
26. Urbanek, Simon. "rJava: Low-level R to Java interface." (2013).
27. Naik, Nitin. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP." *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017.
28. 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.
29. 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.
30. Ancona, Davide, et al. "RPython: a step towards reconciling dynamically and statically typed OO languages." *Proceedings of the 2007 symposium on Dynamic languages*. 2007.
31. Ren, Wei, and Randal W. Beard. *Distributed consensus in multi-vehicle cooperative control*. Vol. 27. No. 2. London: Springer London, 2008.
32. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
33. Lamport, Leslie. "Paxos made simple." *ACM Sigact News* 32.4 (2001): 18-25.
34. Lamport, Leslie. "Fast paxos." *Distributed Computing* 19.2 (2006): 79-103.
35. 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.
36. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
37. 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.
38. Rai, Rohit. *Socket. IO Real-time Web Application Development*. Packt Publishing Ltd, 2013.
39. Pitt, Esmond, and Kathy McNiff. *Java. rmi: The remote method invocation guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.