DeCI - Decentralized Computation Infrastructure

Georgios Fotiadis - SCIPER 271875
João Correia - SCIPER 343955

DSE Project – Master
Team name:
Jenny from the block(chain)
Supervised by:
Pasindu Nivanthaka Tennage

Lausanne, Academic year 2021-2022

Contents

Chapter 1

Introduction

In recent years, decentralized solutions' popularity like the blockchain has surged. At the same time, ideas like the Web3.0 [1] have been gaining momentum and a new trend of replacing traditionally centralized services with their decentralized counterparts is slowly being created. One of the most centralized services of today is the Cloud.

In this work we demonstrate DeCI, a Decentralized Computation Infrastructure designed to allow for the distribution of computation workloads among a network of collaborating peers. With DeCI, we seek to provide to users a decentralized version of a service that has normally been only available through centralized providers like Amazon AWS and Microsoft Azure: a computation cluster to which the user can distribute parallelizable computations.

DeCI is a collaborative framework where nodes help each other when performing computations. To incentivize this collaboration, we've implemented an underlying cryptocurrency system, where to issue a computation, an amount of money must be payed by the issuer to the nodes which will perform the computation.

As such, DeCI presents itself as an advantageous framework for both types of users. Users that need to distribute heavy computations, that are impossible to perform locally, can use the network's resources, while users with idle computation resources can lend them to the network in exchange for financial compensation.

We as a team, distributed the workload of this project among us in the following way:

- Feature to allow new nodes to join the network - **Georgios**
- Feature to allow nodes to estimate the total cost of the desired operation - **João**
- Feature to find available nodes to distribute the computation to - **João**
- Feature to distribute the computation fairly among the available nodes - **João**
- Feature to enable the update of the balances of everyone involved in the computation- **Georgios**
- System performance evaluation and analysis - **Georgios**

Chapter 2

Background

The origins of decentralized computation, in our definition of it being the distribution of workloads among nodes, can be a bit hard to uncover. That is due to the fact that, in the early days, many decentralized system were classified as decentralized computation systems when, in fact, didn't align with our definition of distributing execution, and, as such, could be seen more as decentralized services. Some of these services were the Mix Network system [2], which later served as a foundation to anonymous networks such as TOR [3] and FreeNet [4], as well as decentralized file sharing services like Bittorrent [5].

Based on our definition of computation, the frameworks that better match it are actually distributed computing software, like Apache Hadoop [6] and Apache Spark [7]. These frameworks allow for the distribution of heavy workloads among a group of computation nodes belonging in the same cluster. These services are only distributed within the well defined scope of their cluster, not fully decentralized and publicly accessible for anyone to use. Thus, they're not cooperative, meaning that normally all the nodes are owned by a single entity (typically a company). Additionally, users can only distribute a computation and not perform computation for a third party, in exchange for a financial reward.

Our implementation, looks to be inspired by how these frameworks perform the workload distribution, but takes a different approach on building the service's network. Specifically, we build a cooperative, peer to peer network, where multiple users can benefit by either distributing their computation, or by using their resources to perform a computation for a third party in exchange for financial compensation.

However, one problem arises: What stops a node from joining the network, using it to perform its desired workload and then leave it, thus adding no value and just draining resources? To solve this incentivization problem, we take inspiration from cryptocurrency systems. Typically, these systems use financial benefits to reward good behaviour and network participation, which is what we are also aiming for. In our framework, this financial system is used to "pay" for computations, that is, a node which wishes to distribute a workload, must pay available nodes to do so. Looking through the participants' lenses, lending their computational resources to the network rewards them with financial tokens. All these transaction are recorded in a blockchain, in the same fashion as other cryptocurrency systems.

Our solution, is a decentralized computation platform and it should not be confused with smart contract platforms like Ethereum [8]. While smart contracts platforms focus on the execution and validation of pieces of code by all elements in a network, our system is designed to distribute heavy workloads between different participants. As such, while the main objective of smart contract platforms is to remove a central point of trust by having computations recorded on a blockchain, our main focus is to be able to speed up computations in a collaborative manner.

Chapter 3

System Model & Architecture & Functionalities

From a user's prespective, our framework allows for two main actions:

1. Issue computations to other nodes;
2. Participate in issued computations.

We tried making DeCI's interface as simple as possible for users and hide the complexity of the underlying system. Users can access DeCI through its command line interface and either stay idle and perform computations for third parties or issue a computation by providing the following inputs:

1. The path to the executable;
2. The path to the data that will be used as the input to the executable;
3. The number of nodes that they wish to distribute the computation to.

When a new node wants to issue a computation, the following concrete steps are taken (here, the student which implemented each step will be noted):

- The node joins the network - **Georgios**
- The node estimates the total cost of the operation - **João**
- The issuing node tries to collect the desired amount of nodes to distribute the computation to - **João**
- The issuing node distributes the workload fairly among those nodes - **João**
- The budgets of everyone involved in the computation are updated- **Georgios**

In the following sections, we will analyze how we implemented each step.

3.1 Joining the network

In most decentralized systems, in order for a new node to join the network, it must know the nodeID of a node already participating in the network. In DeCI, we use the IP address of the node as its ID. To facilitate the joining process for users, we have created a few public nodes with known IP addresses that act as a link between the new node that tries to join the network and the rest of the network.

When a user tries to join the network, DeCI, automatically tries to contact one of these public nodes. After this has been done successfully, the new node broadcasts a special Join message and all the nodes

that receive it respond with a AckJoin. The node is blocked until a majority of nodes responds, and when they do, it unblocks and its ready to issue a computation or to perform computations for other peers.

3.2 Cost estimation

To estimate the cost of a computation, the issuer node runs locally its executable on some samples of the input. This is due to the fact that, in our system, the cost of a computation is directly proportional to the amount of time it takes to execute.

As such, the input list is shuffled to prevent any type of bias or trickery caused by the input ordering provided by the issuer. Next, a sample of the input list is is chosen at random and executed, and we measure the time it took to execute.

If the time taken exceeded 5 seconds no more inputs are executed as not to further slowdown the whole process. If it took less than 5 seconds, two more samples are chosen at random and executed, and their times are averaged out with the duration of the first input.

The estimated cost per unit will then be the number of seconds it takes, in average, to execute one sample of the input data. Our system allows for decimal costs, since it wouldn't be fair for a computation which takes 0.001 seconds per input to execute, to cost as much as a computation which takes 0.9 seconds.

There's also a base cost per node. That is, for each node requested, there's a base price to be payed, which is 1 coin per requested node. This is meant to deter issuers from simply requesting the full network to perform a computation and to make a better estimate of an adequate number of nodes to request.

Therefore the full cost is: **number of inputs * cost per input + 1 * number of requested nodes**. If the node has sufficient funds, it proceeds to the node aggregation step.

3.3 Node aggregation

The purpose of this step is to gather the desired number of nodes to perform the computation. Since we're in a decentralized system, where only direct neighbours are known, a node discovery mechanism had to be implemented. This system relies on a similar logic to the budgeted search requests implemented on homework 2. The main message of this mechanism is the **AvailabilityQueryMsg**. This message has the following fields: **RequestID**, **Source**, **Budget** and **AlreadyVisited**.

The RequestID field is used to identify the computation, and the Source field is used to advertise the IP address of the issuer of the computation. The two other fields are more complex. Here, budget stands for *the number of nodes still needed to be reserved* and we explain what AlreadyVisited means later on.

When the issuing node wants to reserve nodes for its computation, it specifies the number of nodes it wants to reserve and this is what we define as the original budget in this context. Then, it divides this budget among its neighbours and sends an AvailabilityQueryMsg to each one.

The neighbours then process this message in the following way:

1. If node is not reserved yet for any computation:
   (a) Reduce message budget by 1
   (b) Reserve itself for this computation and answer back with a confirmation of reservation
2. If budget is 0, drop the message, else, split the budget among neighbours and resend it

Looking at this algorithm, we can make one observation: the budget isn't reduced if the node is already reserved for some computation. This means that if the network is fully reserved for some other computation this message will be indefinitely re-transmitted. That's where the **AlreadyVisited** field comes into play. Every time a reserve message is processed, the peer adds its own IP to this field's list, such that when messages are forwarded, IPs that already appear on this list are not resent these messages, which implies that even in a fully reserved network, messages will be dropped eventually.

If, after a given amount of time, the issuing node was unable to gather the desired number of nodes, it will need to send a reservation cancellation message to each of the nodes it managed to reserve, to signal to them that they can become available again. If it managed to gather sufficient nodes, it proceeds to the workload distribution step. Figure 3.1 illustrates a simple example of the availability gathering process, where blue lines represent availability request queries and red lines represent responses.

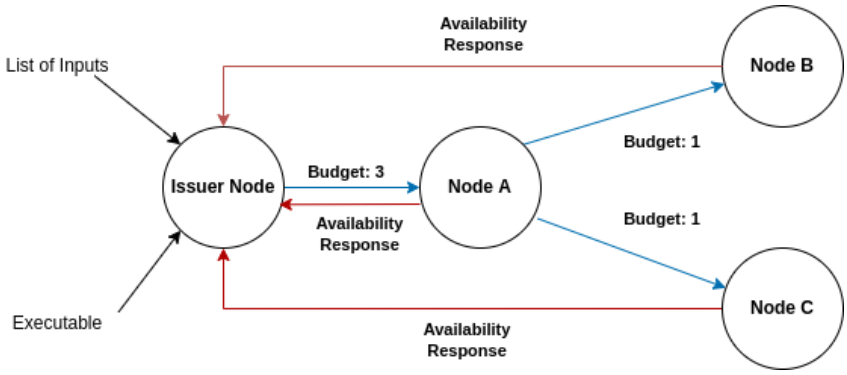

Figure 3.1: Process of spreading availability queries and getting repsonses

3.4 Workload Distribution

After gathering the desired number of nodes, the issuing node must distribute the workload among them. The node tries to do it as fairly as possible by splitting the inputs evenly among the reserved nodes.

A **ComputationOrderMessage** is sent to each of the reserved nodes. This message contains, for each node, the following fields: **requestID**, the **executable** code as a byte array, the list of **inputs** the node must run with the executable and some **instructions** on how to run the executable.

Each of the reserved nodes will gather the outputs generated by the executable and return them to the issuing node via a **ComputationResultMessage**. Finally, the issuing node will store the results and make them available to the user.

Figure 3.2 showcases a very small example with 6 inputs and an executable which duplicates the given input. Blue lines illustrate the issuing node sending the executable and the designated inputs to each node and the red lines represent the worker nodes' responses after completing the computations.

3.5 Balance update

Incentivizing network participation to decentralized applications is always a challenge. In DeCI, we use blockchain based financial compensation to achieve that. Specifically, we use Paxos to achieve consensus on the transaction amount and how it's distributed among the participating peers and we record all transactions on a blockchain.

When the issuer node receives the result of its computation, it first removes the respective amount from its budget. It then broadcasts a special message containing a budget map, which simply is a

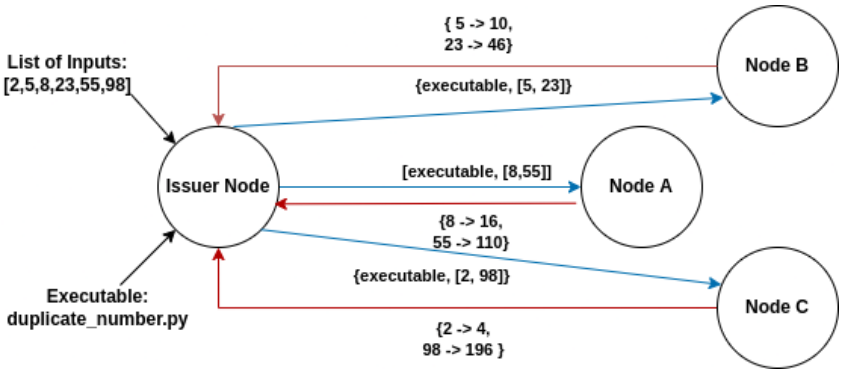

Figure 3.2: Process of sending inputs and the executable and receiving the results

mapping from IP addresses to amount due. All nodes receive this message and record the transaction in a blockchain. If the receiving node participated in the computation, it also updates its budget by the amount in the map. This design is similar to how we implemented the private message mechanism in homework 1 and consensus on homework 3.

Chapter 4

System evaluation and analysis

DeCI's decentralized nature means that, besides the pure computation time spent on by the computational nodes to run the desired computation on the provided data, there is the additional overhead of Paxos to achieve consensus. This overhead doesn't exist in centralized cloud services but it's necessary in a decentralized system like ours. In this section we will provide an analysis of DeCI's performance and specifically, how the overhead introduced by its decentralized nature, evolves as we increase the amount of nodes we distribute the computation to.

We evaluated DeCI on two workloads: a lite one and a heavy one. Our expectation (in accordance with Amdahl's law [9]) was that distributing the computation would yield benefits when done with a few nodes, but the more the amount of nodes increased, the more the overhead for achieving consensus would as well, eventually dominating the total computation time and canceling the benefits of distributing the computation.

This insight was validated. In Figure 4.1, the graphs showcase how the computation duration evolved, as we increased the amount of nodes we distributed the computation to. We observe that the computation time initially drops, but as we increase the amount of nodes, it also increases. We also see, that this happens a lot faster in the case of the lite workload than in the heavy one.

To understand why this is happening, we need to look at Figure 4.2. We observe that the pure computation time in the lite workload, is already small to start with and doesn't reduce significantly when distributed. At the same time, the Paxos overhead continues to steadily increase, eventually overtaking the pure computation duration. In the case of the heavy workload on the other hand, we can clearly see that the more we spread the computation, the smaller the pure computation time becomes. Nevertheless, the Paxos overhead eventually dominates in this workload as well.

To better understand, how the pure computation time and the Paxos overhead relate to each other we have created Figure 4.3. This figure shows what percentage of the total duration of the computation is dedicated to the pure execution time and how much to Paxos. Note that we define the total duration of the computation to be from the moment we start distributing the computation, until all the balances of the involved nodes have been updated. Again we can clearly see, how the paxos overhead steadily increases, eventually occupying the biggest percentage of the total computation time.

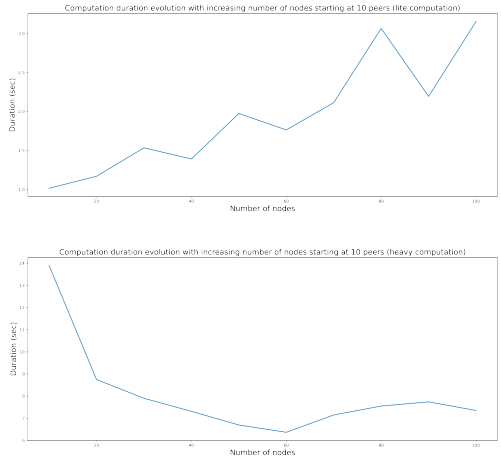

Figure 4.1: Evolution of total duration as we increased the number of nodes, on the lite (top) and heavy (bottom) workload.

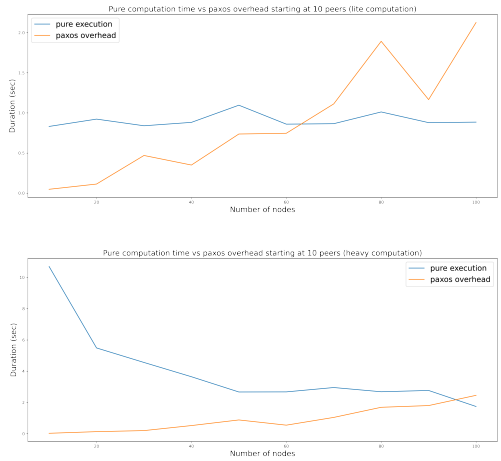

Figure 4.2: Evolution of pure computation time vs paxos overhead as we increased the number of nodes, on the lite (top) and heavy (bottom) workload.

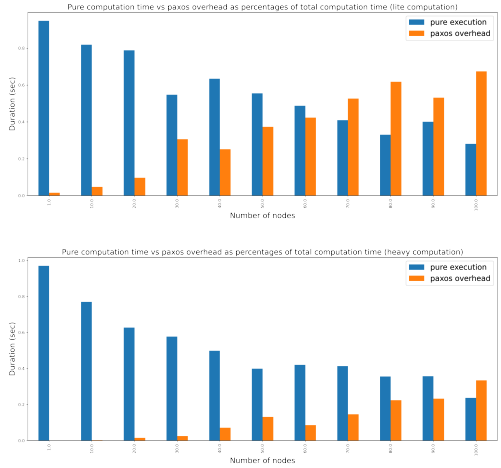

Figure 4.3: Division of total computation duration between pure execution time and paxos overhead, and evolution as the number of nodes increases, on the lite (top) and heavy (bottom) workload.

Bibliography

[1] Shermin Voshmgir. *Token Economy: How the Web3 reinvents the Internet*. Vol. 2. Token Kitchen, 2020 (page 1).

[2] Markus Jakobsson and Ari Juels. "An optimally robust hybrid mix network". In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 2001, pp. 284–292 (page 2).

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004 (page 2).

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. "Freenet: A distributed anonymous information storage and retrieval system". In: *Designing privacy enhancing technologies*. Springer. 2001, pp. 46–66 (page 2).

[5] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. "The bittorrent p2p file-sharing system: Measurements and analysis". In: *International workshop on peer-to-peer systems*. Springer. 2005, pp. 205–216 (page 2).

[6] *Apache Hadoop*. URL: https://hadoop.apache.org (page 2).

[7] *Apache Spark*. URL: https://spark.apache.org (page 2).

[8] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32 (page 2).

[9] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533 (page 7).