Cloud nonce discovery

a report by **Tim Nguyen**

https://github.com/nt1m/cnd

1 Introduction

Blockchain is a domain that is increasingly gaining momentum, especially in applications like security or in the financial market. One of the core components of blockchain is proof of work (POW). It is a way to verify transactions on the blockchain: nodes are assigned a computationally heavy task to perform and the first node to solve this task verifies the block. Typically, the given task is golden nonce discovery. This report shows how the task of nonce discovery can be parallelised on the cloud using Amazon Web Services (AWS).

2 Nonce discovery

2.1 The task

The nonce is a number that is appended to a block of data. The result is then hashed twice using SHA-256. The golden nonce is the number giving the result that starts with a specified number of zero bits, also known as the difficulty. The task of finding this nonce is called "golden nonce discovery".

2.2 The problem

Finding this golden nonce is typically done by bruteforcing the hashes for the range of integers $[0, 2^{32}]$. Figure 1 illustrates the execution time for the implementation of the task being ran on a single machine. For difficulties under 20, the golden nonce will typically be small as the probability of getting a golden nonce is high, meaning the golden nonce can be found quickly on any single computer. However, when the difficulty grows, the probability of getting a golden nonce gets lower, so the execution time on one machine to find a nonce increases exponentially.

Given that the Bitcoin protocol currently uses a difficulty of 74, the computation time on a classic machine could go up to years for such a difficulty. This shows that running the program on a single classic machine does not scale.

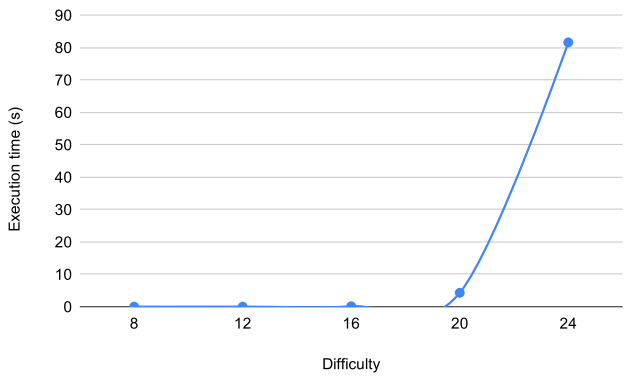


Figure 1: Exponential single-machine runtime growth as difficulty increases

2.3 Parallelising the task

In order to solve the long runtimes, the task can be parallelised: the search space is divided in equal ranges across multiple machines, where each machine only tries to find the golden nonce on the range it is assigned to. For instance, with 4 machines, the ranges would be:

```
[
    [0, 1073741823],
```

```
    [1073741824, 2147483647],
    [2147483648, 3221225471],
    [3221225472, 4294967295]
]
```

3 Putting it in the cloud

Now that a way of parallelising the task is found, the next step is to implement the solution on the cloud. The main benefit of using the cloud is that it is easily possible to do programmatic control of virtual machines. The virtual machines are also available on-demand, allowing as many to be spawned as wanted.

There are many services online which provide cloud computing services, such as Amazon Web Services, Google Cloud Platform, Microsoft Azure or Oracle Cloud Infrastructure. For this task, Amazon Web Services (AWS) is going to be chosen given its popularity.

3.1 A quick overview of AWS

AWS provides a large range of services, but only two of them will be needed for this task: Elastic Compute Cloud (EC2) and Simple Queue Service (SQS).

EC2 allows renting virtual machines on the cloud, also called instances. Instances can be launched, meaning they are created on the go from Amazon Machine Images (AMI). AMIs are images that come pre-installed with an operating system and some basic software.

On the other hand, SQS is a message queuing service that can be used to communicate between different machines. Messages can be sent in JSON format through queues and other consumers can read the messages and delete them. Once a message is delivered, a receipt handle is created and given to the recipient. This handle must be provided in order to delete messages. SQS also solves other hurdles with messaging on the cloud. It ensures at-least-once delivery, messages cannot be lost in-flight since they are stored different servers. A visibility timeout can also be specified to ensure different consumers cannot consume the same message at the same time. This hides the message for a specified amount of time from other consumers once the first recipient has received it. Having these two features is crucial to ensure the reliability of the system.

Amazon provides an API to interact with those services. Tasks like creating EC2 instances or sending and receiving messages through SQS queues can be done through this API. There is a Python wrapper around the API, maintained by Amazon, called Boto. Since the system will be programmed in Python, this library will be useful later on.

3.2 Architecture

The system has two Python scripts:

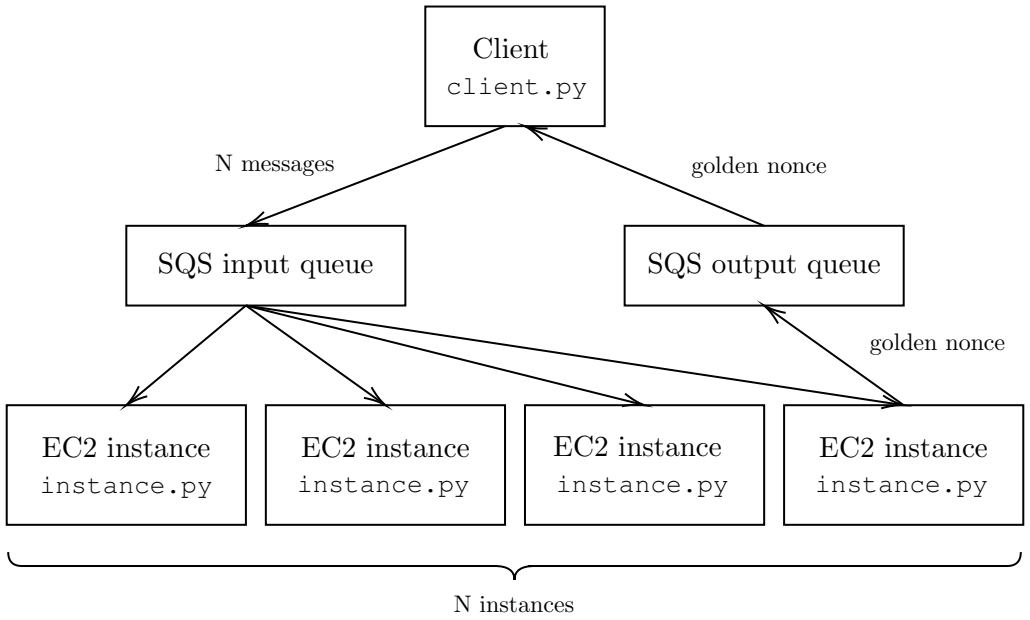
- `client.py`: This is the script being ran from the user's computer:

  ```
  usage: client.py [-h] [-n INSTANCES] [-d DIFFICULTY] [-t TIMEOUT]

  optional arguments:
    -h, --help            show this help message and exit
    -n INSTANCES          Number of VMs to spawn
    -d DIFFICULTY, --difficulty DIFFICULTY
                          Difficulty (number of leading zeroes) for the nonce
                          discovery algorithm
    -t TIMEOUT, --timeout TIMEOUT
                          Maximum time spent in seconds.
  ```

- `instance.py`: This is the script being ran on each of the EC2 instances. This script contains the cloud nonce discovery algorithm.

The overall architecture of the system is described in the diagram below:



Using Boto, `client.py` first creates two SQS queues:

- the input queue, which handles messages *to* the EC2 instances
- the output queue, which handles messages *from* the EC2 instances

The client script then launches the EC2 instances with the `instance.py` script being passed as `UserData` parameter. This executes the `instance.py` on each of the instances that are created. Afterwards, the client script computes the division of the search space into N equal-sized ranges. Each of those ranges is then sent as a message through the SQS input queue. Once the messages are sent, the client script waits for one message from the output queue containing the golden nonce and deletes it once received. `client.py` can also perform scrams: the script makes Boto calls to terminate the EC2 instances and deletes the SQS queues. The scram is being performed once the script receives the golden nonce, but also when it receives an `SIGINT` signal from the system or the user. It is also performed after a certain number of seconds if the user specifies a timeout option.

On the other hand, the `instance.py` script first waits for a message from the input queue. Once that message is received, it is deleted and the nonce discovery algorithm is executed for the range specified in the message. If the golden nonce is found, it is sent back to the client through the output queue and the script stops executing.

However, some setup is needed in order to make the system work.

3.3 Setup

After signing up for an AWS account, the first task that needs to be done is configuring access control. This is needed to use AWS' API. Access control can be configured from AWS' Identity and Access Management (IAM) console. IAM groups are groups of users that share a same set of permissions. Roles are similar, but are set on EC2 instances for a limited amount of time rather than on users. For security purposes, it is important to only grant the permissions that are needed by each part of the system. The system will use the following:

- A *client* group, which has the `AmazonEC2FullAccess`, `AmazonSQSFullAccess` and `IAMFullAccess` permissions.
- A *instance* role, which has the `AmazonSQSFullAccess` permission.

Once the group is created for the client, a set of credentials can be generated by creating a client user on the IAM console. Those credentials will be put on the client's machine in `/.aws/credentials`. The EC2 instances does not need credentials as their IAM role grants the needed permissions.

The next part of the setup that needed to be done was creating an AMI. While the most basic AMI had enough computational power to run the program, it did not have Python or Boto pre-installed to run the program. To create an AMI for this system, an instance of the most basic AMI can be launched. The setup commands can then be ran by connecting to the EC2 instance via SSH:

```
sudo su
yum update
yum install python-pip
pip install boto3
```

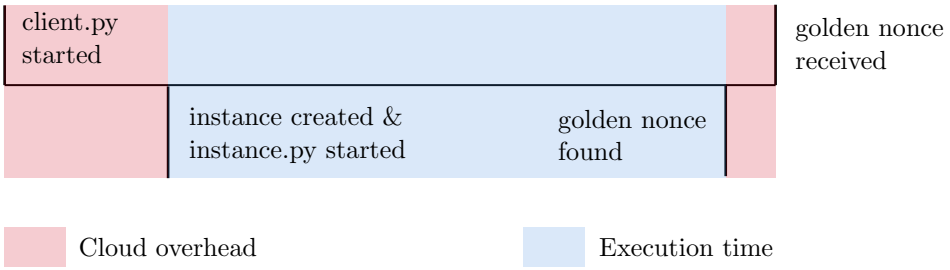
Once these steps are done, the AMI can be created from the EC2 console by selecting the instance and clicking on `Actions > Image > Create Image`.

4 Performance analysis

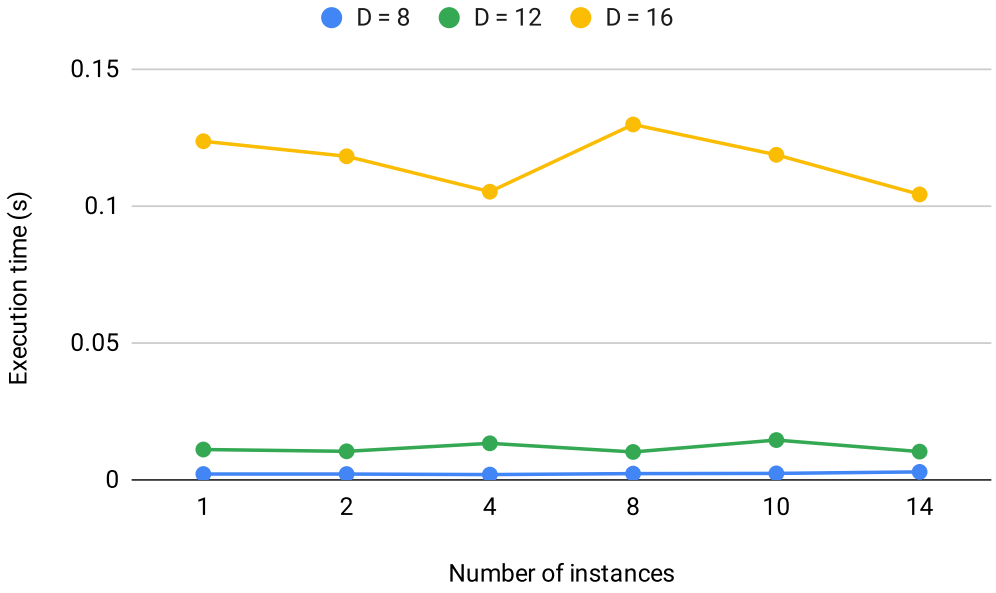
In order to assess the performance of the system, two metrics are measured:

- the average "cloud overhead": the average time over all the number of instance, taken by AWS to create EC2 instances and to communicate via SQS
- the execution time of the nonce discovery: the time taken by the successful EC2 instance to find the golden nonce

The sum of those two metrics gives the total time taken from when the client script is started to the time it receives the golden nonce. This is illustrated in the timeline:



The two metrics were measured for different number of machines $N = \{1, 2, 4, 8, 10, 14\}$ and for different difficulties $D = \{8, 12, 16, 20, 24\}$. Some results have not been measured since they were timing out.

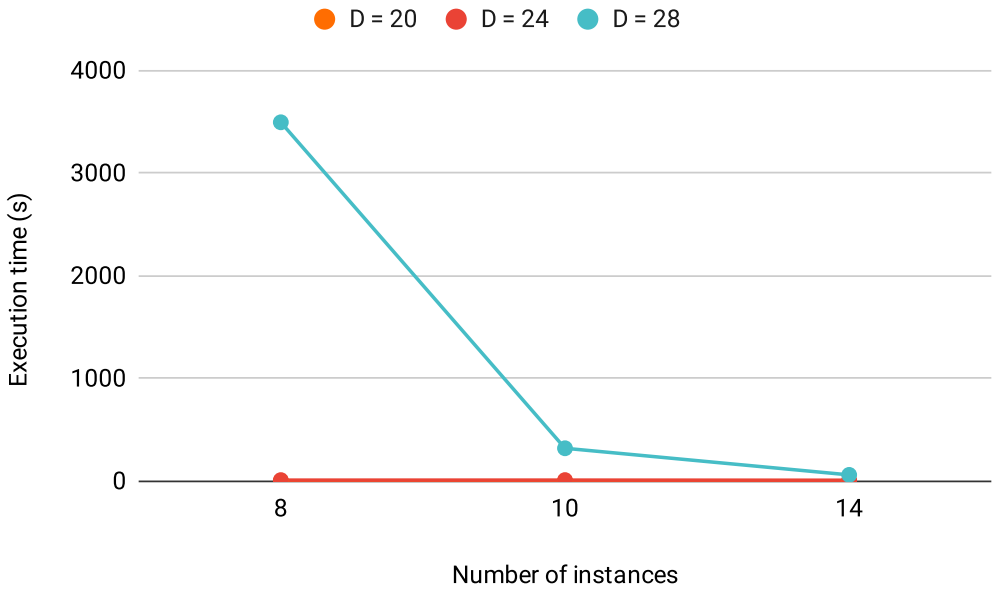


N	Overhead (s)
1	43.44850903
2	44.25972285
4	43.98906817
8	39.6427134
10	44.75104146
14	40.38114576

Figure 2: Execution time and cloud overhead for $D = \{8, 12, 16\}$

As seen in Figure 2, the cloud overhead is more-or-less constant and is around 41 seconds. As for the execution times, they do not vary a lot across different values of N for low difficulties. This is because the golden nonce is always a small number for small difficulties, meaning it will always be found by the first instance, regardless of the number of instances. The execution times are also only of the order of a fraction of a second, as opposed to the average cloud overhead which is around 41 seconds. Since the average cloud overhead is 400 times the execution times, we can conclude that for low difficulties, it is not worth running the task on the cloud since it would take more time for Amazon to setup instances than to run the program locally.

However, this is not the case for larger difficulties:



N	Overhead (s)
8	42.08443395
10	44.86284304
14	43.48186493

Figure 3: Execution time and cloud overhead for $D = \{20, 24, 28\}$

Figure 3 shows that for low difficulties, there is again not much difference between the execution times. However, for a difficulty of 28, on 8 machines, the execution time is 3497 seconds, and it decreases down to 321 seconds for 10 machines, then down to 63 seconds for 14 machines. The execution times for $N \leq 8$ were too high, so they weren't measured. The exponential execution time decrease shows the benefit of parallelising the task on the cloud, especially given that cloud overhead remains constant for any number of instances.

5 Conclusion

The report was able to show an exponential decrease in runtime for difficulties higher than 24, when testing with number of instances N from 1 to 14.

The Bitcoin protocol initially started with a difficulty of $D = 32$ and this difficulty keeps incrementing once in a while. Given this fact, parallelising golden nonce discovery can be very beneficial especially since nowadays, large clusters of $N = 50$ or more machines can be cheaply instantiated on the cloud.

However, the average cloud overhead is non-negligible, which is one of the reasons nonce discovery is typically done on local clusters of GPUs nowadays, eliminating completely the communication overhead with the cloud providers.