Sistemas de Gestión de Seguridad de Sistemas Informáticos

Blockchain cliente-servidor paralelizado

Informe y resumen

Josu Lorenzo Hernández

Facultad de Informática
Grado de Ingeniería Informática

7 de Noviembre, 2022

Índice

Antecedentes y descripción del problema a resolver

En la época contemporánea del mundo de la informática, se han creado tecnologías nuevas y expandido las ya existentes. Desde los smartphones hasta la adopción total de redes, en la última década la informática ha vivido en una lluvia de ideas constante. Una de estas innovaciones se trata del conocido como “dinero digital”, el cual se basa en la tecnología y técnicas del blockchain. Lo que empezó siendo una nota a pie de página sobre métodos de pago y monedas virtuales, se ha convertido en un sistema monetario que se ha establecido permanentemente.

En este contexto, en la asignatura Sistemas de Gestión de Seguridad de Sistemas Informáticos (SGSSI) del grado de Ingeniería Informática de la FISS se ha explorado los fundamentos del blockchain, a la vez que se han realizado actividades prácticas en torno al tema. El profesorado de la asignatura ha propuesto realizar un trabajo optativo de cara a complementar lo que ya se ve en la asignatura, y en mi caso he optado por realizar dicho trabajo.

La actividad es abierta, y he decidido, cogiendo como base uno de los laboratorios realizados, crear una aplicación modelo cliente-servidor para el minado de bloques de la asignatura SGSSI. El lenguaje utilizado ha sido Python, y para realizar la interconexión de las dos partes se ha usado la interfaz de Sockets. A su vez, se han utilizado conocimientos aprendidos en asignaturas de cursos anteriores, en concreto de Servicios y Aplicaciones en Red (SAR). Por otro lado, el trabajo realizado no se basa en sólamente en las características de red; se ha usado el paquete *multiprocessing* que Python ofrece para paralelizar el programa localmente en cada cliente. Esto se ha hecho de cara a mejorar aún más las probabilidades de encontrar mejores hashes, ya que, al ser aleatorios, cuantos más intentos mejor.

Por último, cabe destacar que, realmente se trata de una aplicación en red, es decir, que se comunican diferentes equipos físicamente separados (no localhost). También mencionar que, el código fuente del programa está público en el repositorio de GitHub encontrado en la parte final del documento. Además, en la última versión del código, éste ha sido comentado y documentado para una mejor interiorización del mismo. Finalmente, en el apartado de conclusiones y líneas futuras se comenta y hacen proposiciones sacadas de un trabajo optativo del curso pasado (autor: Ander Gómez).

Análisis y diseño de la solución

Tal y como ya se ha mencionado, el programa que se ha pensado se trata de un software modelo cliente-servidor. Además, para una mejor utilización del hardware disponible se ha paralelizado el programa localmente en el cliente.

Aclarada la idea del programa, primero se va hablar del diseño de la tarea principal; minar bloques. Esta parte de la solución está directamente sacada del laboratorio 5 de la asignatura, pero se han realizado mejoras; por un lado, originalmente raramente se conseguían hashes con prefijos de más de dos 0's. Principalmente optimizando la escritura y lectura que el programa hacía del fichero de entrada, se consiguió reducir considerablemente el tiempo tomando en cada iteración. El funcionamiento del programa base se resume en los siguientes pasos: se lee el fichero, y después se entra en un bucle de cálculo comprobando en cada iteración el hash conseguido.

Entrando en la parte más interesante del trabajo optativo, después se procedió a paralelizar el código localmente, de cara a usar todos los recursos de la máquina en cuestión. Para esto, se hizo uso del paquete *multiprocessing* de Python, el cual es distinto al *multithreading*. Básicamente, se trata de usar todos los hilos del procesador. Gracias a esto, se consigue un uso de la CPU del %100 (más adelante se ve en las pruebas). Cada hilo genera y calcula hashes de forma independiente, y cuando se llega al tiempo especificado, todos los hilo dejan de trabajar y se comparan entre sí los mejores hashes de cada hilo, consiguiendo así el mejor de todos.

Por último, el aspecto de red del software desarrollado. Tal y como he especificado anteriormente, el apartado de red de la aplicación se ha basado en la interfaz de Sockets (proporcionada por Python mediante el paquete *Socket)*. El modelo es cliente-servidor; el servidor transmite el archivo sobre el que realizar los hashes a los clientes, y los clientes realizan lo especificado en los dos párrafos anteriores localmente. El cliente, tras conseguir el mejor hash de todos los hilos locales, transmite el resultado (el nonce) al servidor, el cual nuevamente hace comparaciones; comprueba todos los hashes que han emanado de los nonces enviados por los clientes, y se queda con el mejor resultado. Para finalizar, se crea en el equipo del servidor el archivo final.

Implementación

Las herramientas usadas en la implementación han sido Python y Visual Studio Code. Adicionalmente, el software ha sido desarrollado sobre Windows, pero también funciona en Linux. Naturalmente, se ha usado la terminal de Windows para probar el programa. Cabe mencionar el uso de una librería para el cálculo del propio hash sha-256.

Python: multiprocessing

El paquete multiprocessing se ha usado para realizar la paralelización local en la parte del cliente. El funcionamiento es simple: se envía a los hilos hardware una tarea a realizar (una función a ejecutar), y la realizan independientemente (respecto al resto de hilos). Sin embargo, hay una forma de comunicar todos los núcleos; memoria compartida. Esto se ha usado para que cada "worker" deje en un vector su mejor nonce (en la posición correspondiente al ID del hilo hardware). Gracias a esta memoria compartida, una vez terminado el trabajo el resultado puede ser procesado por el hilo "main", e incluso se puede saber quién lo consiguió concretamente. Por último, cabe destacar el uso de una barrera después de lanzar las tareas a los recursos hardware. Con esto, el programa asegura que sólo va a proceder a comparar los resultados de los cálculos independientes una vez todos los núcleos hayan terminado individualmente.

Python: sockets

El paquete socket se ha usado para realizar la comunicación en red. Con las herramientas proporcionadas por el paquete, se ha desarrollado un protocolo personalizado muy básico pero eficaz. El servidor central, al cual se le especifica qué archivo se va a analizar y a cuántos clientes tiene que esperar a que se conecten, realiza el siguiente procedimiento:

1. Esperar a que se conecten todos los clientes.
2. Enviar señal de inicio.
3. Transmitir el archivo de entrada a los clientes.
   3.1. Avisar al cliente actual de la finalización de la transmisión.
4. Esperar y recibir los resultados de los clientes.

Por otro lado, el otro extremo de la conexión, es decir, los clientes, realizan el siguiente esquema protocolario:

1. Esperar a que el servidor acepte la conexión.
2. Esperar la señal de inicio del servidor.
3. Recibir el archivo (contenido del archivo) a usar para calcular el hash.
   3.1. Comprobar el final de la transmisión del archivo.
4. Enviar el resultado al servidor (el nonce del mejor hash).
5. Cerrar la conexión.

Cabe destacar que la retransmisión de los datos se realiza mediante TCP, y no UDP. Las razones son obvias; si cambiara un sólo bit en la transmisión de la información, fuera en la dirección que fuera, el resultado final sería erróneo dada la naturaleza de la sensibilidad del cálculo de resúmenes hash. Toda la aplicación usa TCP; desde el envío del fichero a usar desde el servidor al cliente, hasta el envío del mejor nonce de cada cliente al servidor. Simplemente, no sería correcto usar UDP en esta aplicación debido a la necesidad de que la información se retransmita íntegramente y sin errores.

Paralelización

El proceso de paralelización no ha resultado muy difícil gracias al paquete que se ha mencionado anteriormente. Dicho paquete permite hacer paralelización a nivel de proceso; se le indica qué tarea (función) se quiere lanzar y con qué argumentos (parámetros de la función). Esto se repite tantas veces como hilos hardware hay en el equipo del cliente.

```
arrRef = []
for i in range(multiprocessing.cpu_count()):
    arrRef.append(multiprocessing.Process(target=calcular, args=(best_nonce_global,)))
    arrRef[i].start()
```

De forma natural, la paralelización conlleva desincronización. Para asegurar que a la hora de calcular el mejor hash local todos los hilos han terminado su tarea, se usa una barrera (la ejecución se queda parada ahí hasta que todos lleguen):

```
multiprocessing.Barrier(multiprocessing.cpu_count())
```

Por último, la paralelización naturalmente también trae independencia de memoria (un espacio de memoria privado para cada proceso). Debido a esto, los procesos no pueden compartir información a no ser que se usen espacios de memoria compartidos creados explícitamente. Es decir, variables compartidas explícitas. Precisamente este tipo de variables se ha usado para que los procesos trabajadores puedan devolver su resultado al programa principal:

```
best_nonce_global = multiprocessing.Array(ctypes.c_int64, multiprocessing.cpu_count())
```

Adicionalmente, es necesario hablar de cómo se ha compartido la información del fichero a los procesos, ya que esta información la recibe el proceso padre, y no está compartida con los hijos (espacios de memoria diferentes). Para evitar tener que enviar explícitamente la totalidad del fichero usando la librería multiprocessing, se ha optado por crear un fichero temporal (temp.txt) en el cual se escribe la información recibida. Después, los procesos paralelos abren y leen ese archivo. Al terminar el programa del cliente, se borra dicho archivo temporal:

```
if (os.path.exists("temp.txt")):
    os.remove("temp.txt")

with open("temp.txt", "wb") as escribir:
    escribir.write(fileContentOG)
```

(En el proceso paralelo:)

```
fileContentOG = None
with open("temp.txt", "rb") as escribir:
    fileContentOG = escribir.read()
```

Cliente

El extremo cliente del programa tiene tres partes; establecimiento de la conexión y recibo de los datos, cálculo y procesamiento de los hashes, y finalmente comparar y conseguir el mejor hash (y por ende el mejor nonce) de entre todos los hilos.

Se empieza por establecer la conexión con el servidor. Para ello, se usa la IP que se ha proporcionado mediante parámetro; el programa va a establecer conexión con esa IP. Cuando se consigue conectar, el programa espera a que le llegue la señal de inicio; cuando llegue, quiere decir que todos los clientes están conectados y listos para recibir el archivo sobre el cual realizar los resúmenes hash. Este archivo se transmite en trozos de 512 bytes. El servidor indica la finalización del envío del fichero mediante un mensaje “END”.

```
print("\n  >> Esperando al servidor...")
dir_serv = (sys.argv[1], PORT)
s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.connect( dir_serv )

print("\n  >> Esperando señal del servidor...")
buf = s.recv( len("START".encode()) ).decode()

print("\n  >> Recibiendo datos del archivo...")
fileContentOG = ""
while True:
    chunk = s.recv( 512 ).decode()

    chunkLen = len(chunk)
    if chunk[chunkLen-3:chunkLen] == "END":
        chunk = chunk[:-3]
        fileContentOG += chunk
        break

    fileContentOG += chunk
```

Ahora, el cliente realiza las tareas explicadas en el apartado anterior de paralelización. Una vez terminados los cálculos, se comparan los resultados de los hilos y se consigue el máximo local del equipo. Finalmente, se envía el nonce al servidor.

```
for i in range(multiprocessing.cpu_count()):
    nonceFinal = "{:08x}".format(best_nonce_global[i]).lower()
    lastLine = nonceFinal + " G06"
    local_fileContent = fileContentOG + lastLine.encode()

    final_hash = generate_hash(local_fileContent).hex()

    print("  [" + str(i) + "] >> " + lastLine)
    if (bestHash > final_hash):
        bestHash = final_hash
        bestLastLine = lastLine

print("\n ============================")
print("    || RESULTADOS LOCALES ||   ")
print(" ============================\n")
print("  >> Mejor hash: " + bestHash)
print("  >> lastLine: " + bestLastLine)

nonce = bestLastLine[0:8]
s.sendall( nonce.encode() )
```

Servidor

Principalmente, el servidor tiene dos partes; el envío de datos, y el procesamiento de las respuestas individuales, consiguiendo así el mejor hash.

Primero, el envío de datos. Se espera a establecer conexión con el número de clientes especificado por parámetro de entrada, y se informa de las conexiones mediante un mensaje por consola. Cuando todos están conectados, se le envía a todos un mensaje para hacerles saber que ahora van a recibir el contenido del archivo sobre el que hay que calcular el hash. Tras eso, se realiza la transmisión del contenido del fichero en envíos de 512 bytes. Tal y como se explicará en el apartado de pruebas del documento, para asegurar el correcto envío de los datos se ha elegido hacer envíos de únicamente 512 bytes cada vez. Por último, se envía un mensaje de finalización de transmisión. Ahora el servidor está a la espera del envío de los resultados por parte de los otros extremos de las conexiones.

```
print("\n  Esperando clientes... (" + str(CLIENTS) + ")")

for i in range(CLIENTS):
    dialogo, dir_cli = s.accept()
    dialogos.append(dialogo)
    direcciones.append(dir_cli[0])
    print( "  >> Cliente conectado desde {}:{}.".format( dir_cli[0], dir_cli[1] ) )

for i in range(CLIENTS):
    dialogos[i].sendall( "START".encode() )

for i in range(CLIENTS):
    chunkCont = 0
    while True:
        chunk = fileContentDec[chunkCont:chunkCont+512]
        chunkCont += 512
        dialogos[i].sendall( chunk.encode() )
        if not chunk:
            break

    dialogos[i].sendall( "END".encode() )
```

Tras realizar las tareas de cálculo, los clientes responden al servidor con el envío del mejor nonce que han conseguido calcular localmente. Se recrea el contenido con el que se usó el nonce en el cliente (contenido del fichero de entrada + nonce + identificador), y se calcula el hash. El proceso se repite para todos los clientes conectados, y se comparan resultados consiguiendo así el prefijo con más 0's. Se cierran todas las conexiones, y por último se crea el fichero resultado localmente.

```
for i in range(CLIENTS):
    buf = dialogos[i].recv( 512 ).decode()
    lastLine = buf + " G06"
    print(" >> [" + direcciones[i] + "]: " + lastLine)
    local_fileContent = fileContent + lastLine.encode()
    new_hash = generate_hash(local_fileContent).hex()

    if (bestHash == None or bestHash > new_hash):
        bestHash = new_hash
        bestLastLine = lastLine

for i in range(CLIENTS):
    dialogos[i].close()
```

Pruebas

Las pruebas se realizaron originalmente en un sólo ordenador, pero tras comprobar el correcto funcionamiento se empezaron a realizar en equipos físicamente independientes conectados a la misma red. Primero se lanza el servidor, y después los clientes.

A pesar de que el código en ambos extremos de la comunicación fuera correcto, se estuvo una suma de tiempo considerable intentado entender el porqué de la mala transmisión del contenido del archivo especificado en el servidor. Originalmente, se le indicaba a la interfaz de sockets que enviara todo el contenido en un sólo envío de 8192 bytes, es decir, 8KB. Sin embargo, tras varias horas haciendo pruebas resultó que intentar hacer envios por encima de 1024 bytes (1 KB) no es seguro, ya que posiblemente no se pueda realizar todo el envío (lo cual no significa que los caracteres enviados sean erróneos; literalmente el envío se corta, pero la información que se ha llegado a transmitir es correcta). Por lo tanto, se decidió particionar el envío único de 8KB en envíos más pequeños de 512 bytes. Gracias a este cambio, se asegura el envío íntegro del contenido del fichero, sin cortes arbitrarios.

Antes de mostrar los resultados, los programas se lanzan de la siguiente manera:

El servidor
python sha256-server-mp-nodis.py <archivo a retransmitir> <número de clientes>
ejemplo: python sha256-server-mp-nodis.py SGSSI-22.CB.01.txt 1

El cliente
python sha256-cliente-mp-nodis.py <IP del servidor>
ejemplo: python sha256-cliente-mp-nodis.py 192.168.0.103

Para conseguir la IP a especificar en el cliente, se ha usado el comando ipconfig (ifconfig en sistemas Linux).

Servidor

```
 Esperando clientes... (1)
 >> Cliente conectado desde 192.168.0.103:1767.

 ==============================
    || CLIENTES MINANDO... ||
 ==============================

>> [192.168.0.103]: 9ba04820 G06

 ==========================
  || MEJOR RESULTADO ||
 ==========================

Mejor hash (4): 000062f87d116b73eda012dabb7c294be11400f01686ca4f41203c6f43ec2e3e
Archivo resultado generado: "SGSSI-22.CB.02.Paralelo-Red.txt"
```

Cliente

```
 >> Esperando al servidor...

 >> Esperando señal del servidor...

 >> Recibiendo datos del archivo...

==============================
   || CALCULO COMENZADO ||
==============================
```

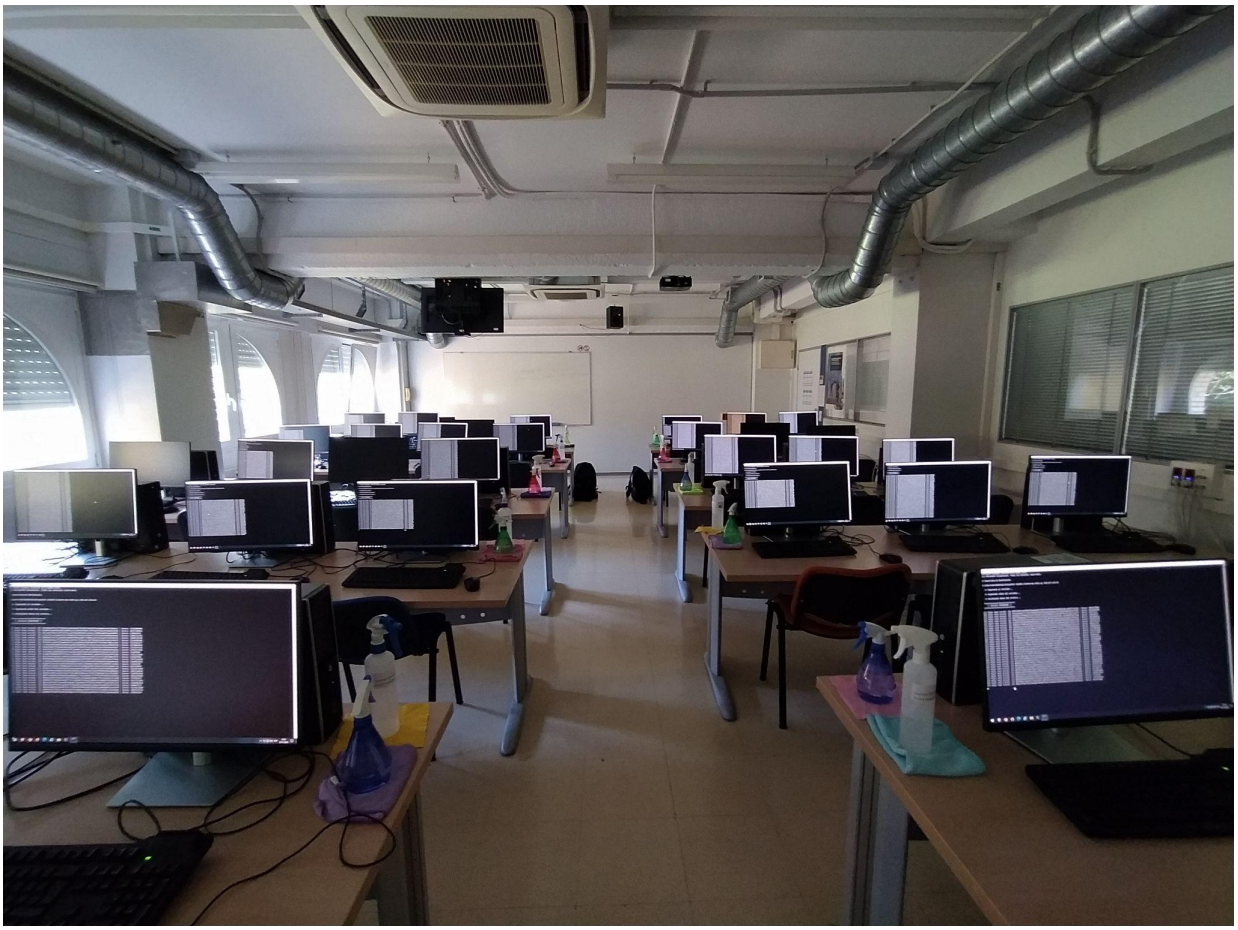
...

```
[4] Mejor hash actual: (2) 00208c91f6a2e2e433d0fa76d7fccb98af60a3e7d9e366f08cdc0844f39e3ebb | Nonce: 3951124465
[6] Mejor hash actual: (3) 00037f54c9e73a9f90a8bae1402035ac12782fef6163837893e4230ef4321d24 | Nonce: 1224224393
[4] Mejor hash actual: (2) 00161b2c74658037606ef422623ce0a25777915dbedc8a49cff3db20fac9c653 | Nonce: 1392421114
[0] Mejor hash actual: (2) 001252cf21d798eced8b82e793eb2943b2cad9fc22028c87944fe3069bc17d70 | Nonce: 2123358667
[2] Mejor hash actual: (2) 003467f33d70c0eb457039d30a569e4555702229f96958f420e49d794caf978a | Nonce: 147692496

 [0] >> 7e8fe1cb G06
 [1] >> 3a816684 G06
 [2] >> 08cd9bd0 G06
 [3] >> 9ba04820 G06
 [4] >> 52fea8fa G06
 [5] >> ccb039cb G06
 [6] >> 48f82e89 G06
 [7] >> 04279b04 G06

============================
   || RESULTADOS LOCALES ||
============================

 >> Mejor hash: 000062f87d116b73eda012dabb7c294be11400f01686ca4f41203c6f43ec2e3e
 >> lastLine: 9ba04820 G06
```

Es necesario mostrar la utilización del sistema del cliente. Gracias al multiprocessing, se usa la totalidad de recursos CPU en el sistema cliente. En la primera imagen se refleja el como al ejecutar el programa, todos los hilos hardware (en mi caso, 8) son utilizados al 100%. También se puede observar en el desplegable de la izquierda que el procesador está trabajando a casi 3 GHz. Por el contrario, en la segunda imagen se puede observar cómo al terminar el programa, la CPU se descongestiona y su frecuencia baja a menos de 2 GHz. A su vez, cabe destacar la subida de temperaturas; en ejecución, el programa hace que la

CPU llegue a casi 90ºC. Antes de comenzar la prueba, el procesador se situaba en los 50ºC (aclarar que estos valores son orientativos y dependen del equipo que se utilice; en mi caso, he utilizado mi portátil personal).

Por último, se ha adjuntado una foto de una red a escala más real en la que se podría usar el programa. En concreto, el programa se ha ejecutado en el aula E04 en 29 ordenadores; 28 clientes y 1 servidor.



Conclusiones y líneas futuras

El área del blockchain es relativamente nuevo, y que esta tecnología se enseñe en una asignatura del grado es de agradecer. Además, no sólo nos quedamos únicamente en el estudio teórico, si no que también entramos en la implementación. Gracias a esto y a conocimientos adquiridos en asignaturas anteriores, decidí crear este software, ya que este área de la informática me ha resultado muy interesante.

En lo que respecta a posibles mejoras y líneas futuras del software, se ha pensado en una serie de mejoras las cuales resultarían en una mejor usabilidad y personalización:

1. Implementar en el cliente una forma de detectar que el fichero no ha sido completamente transmitido.
2. Especificar en el servidor el identificador que el cliente tiene que usar para realizar el cálculo del hash.
3. Especificar en el cliente por parámetro un seudónimo para que el usuario pueda después en el resultado del servidor identificar fácilmente qué equipo ha sido el que ha calculado el mejor resumen.
4. Usar las carpetas de archivos temporales que ofrecen los SO para almacenar ahí el archivo temporal creado por el cliente (Windows, AppData\Local\Temp; Linux, /tmp).
5. Introducir otro parámetro de programa en el servidor, que sirva para indicarle a los clientes cuánto tiempo tiene que estar minando (es decir, tiempo configurable remotamente).
6. Introducir otro parámetro en el programa del servidor, que sirva para indicar al cliente cuántos experimentos tiene que realizar.
7. Reescribir el programa en un lenguaje que se ejecute más rápido (por ejemplo, C).
8. Reordenar, estructurar y comentar el código.

Siguiendo con las posibles ramas futuras, el profesorado de la asignatura proporcionó el informe de una práctica optativa realizada el año pasado por un alumno. Ese trabajo está relacionado con este, razón por la que el profesor recomendó leer, coger ideas y estudiar conceptos de él.

Tras haber analizado y leído el documento, he concluido que la aplicación e implementación de las siguientes ideas en mi proyecto resultarían en un trabajo más profesional y completo:

1. Tener una interfaz gráfica para poder visualizar mejor los equipos conectados a la red de mineros (y no simplemente usar la terminal del SO).
2. Implementar el servidor como un servidor en la nube real, hosteado y desplegado, al cual se puede llegar tanto dentro como fuera de la red local.
3. Usar formatos estandarizados para guardar la información relativa a la minería, tal y como se hace en el trabajo analizado (JSON).
4. Tener varios tipos de ejecución para no usar siempre obligatoriamente todos los recursos de la máquina cliente.
5. En lugar de siempre escoger nonces aleatorios, optimizar la creación de estos para conseguir prefijos con más 0’s, ya sea con generación secuencia o pseudosecuencial.
6. Hacer que el servidor tenga un mayor y mejor control sobre la distribución de la carga de trabajo, haciendo que cada cliente envíe las características de su hardware para que el servidor pueda balancear y distribuir mejor la carga.

En definitiva, hay una rama de mejoras realmente útiles si se quisiera mejorar el programa. En lo que respecta al tiempo invertido, tanto en implementación como en aprendizaje de librerías y redacción de este documento, el trabajo optativo ha sido desarrollado en aproximadamente 10-12 horas.

Bibliografía, referencias y código

Código fuente del Software - Repositorio GitHub
Minero-Cliente-Servidor-Paralelizado (GitHub/JosuLor)

Trabajo optativo mencionado - Ander Gómez
SGSSI-21.TrOp.AGOM.pdf - Google Drive

Multiprocessing - Documentación Python
multiprocessing — Process-based parallelism — Python 3.11.0 documentation

Sockets - Documentación Python
socket — Low-level networking interface — Python 3.11.0 documentation

Apuntes TCP - Asignatura SAR
TCP - SAR - Python 3

Laboratorio SGSSI - Minar bloques
Laboratorio 5 SGSSI

Librería python - SHA 256
GitHub - Python - SHA 256

Khan Academy - Conocimientos de la tecnología blockchain
Khan Academy - YouTube