IPFS:

Almacenamiento Distribuido de Archivos con Permisos de Usuario

Gabriel Fernando Martín Fernández y Aarón Rial Gonzalez

1 *Objetivo del Proyecto*

El proyecto presentado en esta memoria consiste en un sistema de almacenamiento distribuido de archivos que implementará permisos de edición y visualización para los usuarios mediante el uso de IPFS.

IPFS (InterPlanetary File System o Sistema de Archivos InterPlanetario) es un protocolo y una red de almacenamiento y distribución de archivos mediante un sistema de archivos distribuidos. Las posibles aplicaciones de IPFS son múltiples y variadas, pero la cual interesa a nuestro proyecto es la de el almacenaje y acceso de archivos por varios usuarios.
Sin embargo, el simple almacenaje y acceso a un grupo de archivos de manera distribuida no resultaría suficiente para asegurar la seguridad de los archivos, por lo que un control de acceso debería ser establecido.

Éste es el objetivo de nuestro proyecto: crear un control de acceso basado en los usuarios para un sistema de almacenaje de archivos distribuido.

2 *Concepto del Proyecto*

Este proyecto fue concebido como la conclusión a la implementación de un sistema de archivos distribuidos a un concepto previamente desarrollado: el de un sistema de compromiso-revelación.

Ésta implementación de dicho sistema consiste en que cada usuario participante que suba un archivo a la aplicación IPFS queda ligado a su archivo. A pesar de que el compromiso y la revelación previamente se utilizaron con otros fines, en éste proyecto tienen la finalidad de poder identificar al propietario de cada archivo, otorgándole así control sobre el mismo.

La lógica del funcionamiento del proyecto será explicada a continuación, mientras que la implementación se tratará más adelante.

1. En primer lugar un usuario de la aplicación sube un archivo a IPFS mediante la misma.

2. Los usuarios de la aplicación pueden interactuar con los archivos subidos a IPFS ejecutando distintas tareas.

3. Cuando un usuario ejecuta una de las tareas (leer, eliminar, etc...) IPFS contacta al smart contract para comprobar los permisos asociados al archivo.

4. El contrato entonces ejecuta la tarea y responde a la aplicación IPFS, o envia un error si el usuario no tiene permisos para ejecutar esa acción.

5. Si la aplicación recibe la respuesta esperada por el contrato, termina la tarea (e.g. borra el archivo de IPFS). En el caso de que reciba un error del contrato la tarea no se ejecuta.

A pesar de que el propósito de éste proyecto ya ha sido implementado con anterioridad sin el uso de IPFS existen ciertas ventajas con respecto a éstas implementaciones.

Principalmente, el uso de IPFS garantiza un sistema distribuido de archivos, consiguiendo una reducción en la centralización del sistema y en la dependencia de pocos servidores. IPFS también proporciona una mayor velocidad y eficiencia en el procesado de los datos, además de una mayor permanencia de los archivos.

3 *Posibles Casos de Uso*

El propósito de éste proyecto es simple, pero ampliamente adaptable, ya que aunque existan varias opciones para subir y compartir archivos en Internet nuestro proyecto propone una solución descentralizada y controlada (ya que el sistema de permisos ofrece la seguridad de que solo el propietario de un archivo puede borrarlo o editarlo), por lo que podría ser utilizado en multitud de escenarios que requieran a varios usuarios enviando un archivo a un mismo sitio conjunto.

4 *Diseño del Código*

Una vez explicado el proyecto y sus objetivos, se explicará el desarrollo del código en Solidity y las ediciones realizadas en IPFS.

4.1 *Inicio del Código*

El código comienza especificando la versión de Solidity en la que será compilado y ejecutado. En el caso de éste proyecto ésta versión será la 0.8.3 o superior. En ésta sección se define también el smart contract para almacenar los archivos en IPFS.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract IpfsStorage{
```

4.2 *Estructuras*

El primer paso tras la creación del contrato será definir las escructuras. Éste proyecto sólo contará con un struct, el cual será dedicado para almacenar información de manera ordenada sobre los archivos. El struct almacenará la dirección de su propietario, su propia dirección IPFS y los permisos de usuario (en forma de mappings), así como su estado en la aplicación de IPFS (privado, público o borrado).

```
1     // STRUCTS
2     struct File {
3         address owner;
4         string  addr;
5         mapping (address => bool) userHasReadAccess;
6         mapping (address => bool) userHasWriteAccess;
7         bool    isPublic;
8         bool    Erased;
9     }
```

4.3 *Variables Globales*

El siguiente paso será establecer variables globales. Estas variables no son parametrizables, si no que serán utilizadas para almacenar información durante el código.

```
1     //GlOBAL VARIABLES
2     File[] private files;
3     uint private fileCount;
4     mapping (address => uint[]) private userOwnedFiles;
```

4.4 *Funciones Parametrizadoras*

En ésta sección del código se establecen las funciones parametrizadoras.
Éstas funciones serán utilizadas para actualizar los parámetros del struct "File" durante la ejecución de las distintas tareas.

```
1     //SET PARAMETERS
2     function setWriteAccess(address user, uint fileIndex, bool value)
         public validIndex(fileIndex) onlyOwner(fileIndex) {
3         files[fileIndex].userHasWriteAccess[user] = value;
4     }
5
6     function setReadAccess(address user, uint fileIndex, bool value)
         public validIndex(fileIndex) onlyOwner(fileIndex) {
7         files[fileIndex].userHasReadAccess[user] = value;
8     }
9
10    function setPublicFlag(uint fileIndex, bool value) public validIndex
         (fileIndex) onlyOwner(fileIndex) {
11        files[fileIndex].isPublic = value;
12    }
```

4.5 *Modificadores*

La siguiente sección define los distintos modificadores para las funciones del contrato. Éstos modificadores se encargarán de comprobar si un archivo (o un usuario) tiene los permisos adecuados para una tarea.

```
1     //MODIFIERS
2      modifier validRead(uint fileIndex) {
3         require(files[fileIndex].userHasReadAccess[msg.sender] || files[
             fileIndex].isPublic);
4         _;
5     }
6
7     modifier validWrite(uint fileIndex) {
8         require(files[fileIndex].userHasWriteAccess[msg.sender]);
9         _;
10    }
11
12    modifier onlyOwner(uint fileIndex){
13        require(files[fileIndex].owner==msg.sender);
14        _;
15    }
16
17    modifier validIndex(uint fileIndex) {
```

```
18          require (fileIndex >= 0 && fileIndex < fileCount);
19          require(!files[fileIndex].Erased);
20          _;
21      }
```

4.6 *Métodos Principales*

Finalmente, se definen las funciones que comprenderán los métodos principales durante la ejecución del contrato.
En un primer lugar se define la función encargada de añadir los archivos al IPFS y de actualizar las variables (así como crear un nuevo struct que almacene la información del archivo).

```
1       //MAIN METHODS
2       function addFileIPFS(string memory file) external returns (uint
            index){
3           File storage newFile = files.push();
4           newFile.owner = msg.sender;
5           newFile.addr = file;
6           userOwnedFiles[msg.sender].push(fileCount);
7           newFile.isPublic = true;
8           fileCount++;
9           setReadAccess(msg.sender, fileCount - 1, true);
10          setWriteAccess(msg.sender, fileCount - 1, true);
11          return fileCount - 1;
12      }
```

Las siguientes funciones ejecutan las distintas tareas posibles, haciendo uso de los modificadores para limitar el acceso a las mismas dependiendo de los permisos de cada usuario.

```
1       function readFileIPFS(uint fileIndex) external view validIndex(
            fileIndex) validRead(fileIndex) returns (string memory file) {
2           return files[fileIndex].addr;
3       }
4
5       function editFileIPFS(uint oldFileIndex, string memory newFile)
            external validIndex(oldFileIndex) validWrite(oldFileIndex) {
6           files[oldFileIndex].addr = newFile;
7       }
8
9       function deleteFileIPFS(uint fileIndex) external validIndex(
            fileIndex) validWrite(fileIndex) returns (string memory _file) {
10          files[fileIndex].Erased = true;
11          return files[fileIndex].addr;
12      }
```

La siguientes dos funciones definen las dos últimas tareas disponibles para los usuarios. La primera función define la tarea de crear una lista con todos los archivos públicos. El resultado de ésta será una lista con los índices de cada archivo subido a la aplicación (que tenga el estado Público).

```
1       function getPublicFilesIPFS() external view returns (uint[] memory
            filesIndex) {
2           bool [] memory defaultAns = new bool[](fileCount);
3           uint publicFilesCount = 0;
4           for (uint i = 0; i < fileCount; i++){
5               if (files[i].isPublic && !files[i].Erased) {
```

```
                defaultAns[i] = true;
                publicFilesCount++;
            }
        }
        uint [] memory ans = new uint[](publicFilesCount);
        uint j = 0;
        for (uint i = 0; i < fileCount; i++){
            if(defaultAns[i]){
                ans[j] = i;
                j++;
            }
        }
        return ans;
    }
```

La segunda y última función define la creación de una lista de los archivos que son propiedad del usuario que ejecuta la tarea.

```
    function getFilesOwnedIPFS() external view returns (uint[] memory
       filesIndex) {
        uint _fileCount = userOwnedFiles[msg.sender].length;
        bool [] memory defaultAns = new bool[](_fileCount);
        uint ownedFilesCount = 0;
        for (uint i = 0; i < fileCount; i++){
            uint index = userOwnedFiles[msg.sender][i];
            if (!files[index].Erased) {
                defaultAns[i] = true;
                ownedFilesCount++;
            }
        }
        uint [] memory ans = new uint[](ownedFilesCount);
        uint j = 0;
        for (uint i = 0; i < fileCount; i++){
            if(defaultAns[i]){
                ans[j] = i;
                j++;
            }
        }
        return ans;
    }

}
```

4.7 *IPFS*

Los ficheros de la aplicación IPFS usados fueron los creados y obtenidos durante la primera parte de la práctica, pero a dichos ficheros se les añadieron ciertas secciones de código.

La inclusión más relevante es la de la opción de borrar de IPFS los archivos, la cual tuvo que ser incluida manualmente. Este código añade la opción de eliminar no solo un archivo de IPFS, si no que también su copia local.

```
    else if (option === "Delete") {
      const cid = await readFileIPFS(input)
      const result = await deleteFileIPFS(input);
      strResult = cid.toString();
```

```
5       if (strResult.length > 0){
6         const client = await create('/ip4/127.0.0.1/tcp/5001');
7         const _cid = CID.parse(strResult);
8         const _result = await client.pin.rm(_cid, {recursive: true,
              force: true});
9         await client.files.rm(`/${cid}`);
10        console.log(_result);
11        setTextOperationResult(strResult);
12
13      }
14      else {
15        setTextOperationResult("There is no file to delete!");
16      }
```

Además de ésta inclusión, cabe mencionar la redacción del código encargado de crear la interfaz de la aplicación, el cual también tuvo que ser añadido manualmente, y resultó en la interfaz a continuación:

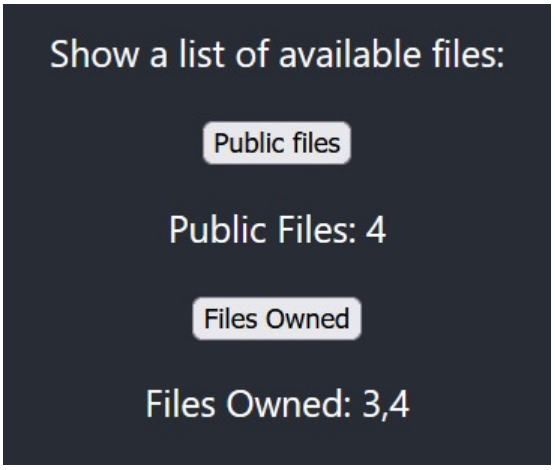

Figura 1: Botones para generar una lista de archivos públicos o otra lista de archivos propios.

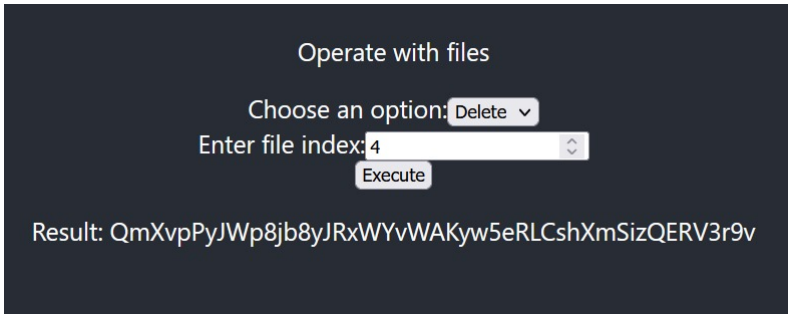

Figura 2: Ejemplo del borrado de un archivo de la aplicación.

Figura 3: Ejemplo del cambio de visibilidad de un archivo de público a privado.

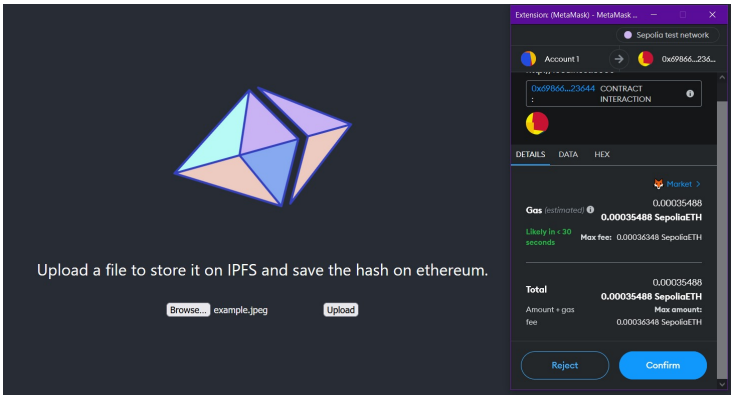

Figura 4: Interfaz para el subido de archivos y ejemplo de una transacción.

5 *Compatibilidad con Windows*

Para la instalación y uso de la aplicación en un sistema operativo Windows 10 / 11 es necesario tener WSL. La versión 2 de WSL requiere habilitar Virtual Machine Plataform en Windows Features y permite instalar una distribución de Linux dentro del propio sistema operativo de Windows, por defecto se instala Ubuntu. Luego de esto se instala Docker y este reconoce el subsistema de Windows instalado previamente, permitiendo acceder a los ficheros de los contenedores con el explorador.

Los comandos empleados para habilitar el contenedor de IPFS en Windows se muestran a continuación:

```
docker run -d --name ipfs_host -v ${PWD}:/export
-v ${PWD}:/data/ipfs -p 4001:4001  -p  4001:4001/udp
-p  127.0.0.1:8080:8080  -p 127.0.0.1:5001:5001 ipfs/kubo

docker exec ipfs_host ipfs config --json
API.HTTPHeaders.Access-Control-Allow-Origin
'[\"http://0.0.0.0:5001\", \"http://localhost:3000\",
\"http://127.0.0.1:5001\", \"https://webui.ipfs.io\"]'

docker exec ipfs_host ipfs config --json
```

```
API.HTTPHeaders.Access-Control-Allow-Methods
'[\"PUT\", \"POST\", \"GET\"]'
```

Para que el proyecto funcionara debidamente fue necesario modificar el fichero App.js ubicado en emph/dapp_ipfs/src/App.js concretamente cambiando la referencia al IP *0.0.0.0* por *127.0.0.1*. En sentido general el uso de *0.0.0.0* provoca ciertos errores por ejemplo las imágenes no las carga cuando se referencia con este IP pero si con localhost.

6 *Conclusiones*

IPFS es una herramienta tecnológica que, si bien no tiene muchas aplicaciones novedosas, es un gran complemento a blockchain y, en general, al movimiento basado en la descentralización del Internet, ya que no solo provee una alternativa descentralizada, si no que también aporta mejoras con respecto a otras opciones.

Además de esto, el desarrollo de esta sencilla aplicación nos ha demostrado la capacidad de adaptación que tienen los conceptos en la creación de software, ya que la creación de este proyecto vino de la implementación de IPFS a un proyecto anterior, lo cual terminó por causar la creación de una aplicación con un propósito completamente nuevo.

De manera general se mostró como se puede hacer una sencilla aplicación para manejar la subida y descarga de archivos usando IPFS y controlando los permisos usando un contrato de Solidity. Sobre esto se puede ampliar al incluir funcionalidades más complejas que requieran ser ejecutadas por la blockchain y debido a su alto coste en cadena necesiten un servicio paralelo para guardar archivos. IPFS en este sentido es ideal pues comparte la naturaleza distribuida de blockchain y se ajusta a varios tipos de problemas que pueden solucionarse empleando ambas tecnologías en conjunto.