Applicazione Decentralizzata per la Sicurezza e Autenticazione di Contenuti Digitali

Uso di Smart Contract e Blockchain Contro la Violazione dei Diritti d'Autore e i Deepfake

Dipartimento di Informatica
Corso di Laurea in Informatica
Tesi Compilativa di Lorenzo Cavallaro
Anno Accademico 2023-2024

Relatore:
Prof./Dott. Sara Migliorini

INTRODUZIONE

LAVORO SVOLTO:

Il progetto nasce dal mio tirocinio universitario e ha l'obiettivo di creare una soluzione basata su **blockchain** per la gestione dei **diritti d'autore** e la protezione dei contenuti digitali. Ho sviluppato uno **smart contract** su **Ethereum** che consente la registrazione e la tracciabilità dei contenuti, garantendo agli autori il controllo sui propri diritti e la possibilità di autorizzare nuove creazioni derivate. In seguito, ho creato un'applicazione frontend con **Ionic** e **React** per facilitare l'interazione degli utenti con il contratto, testando il sistema sulla **Binance Smart Chain Testnet**.

PROBLEMA AFFRONTATO:

Il problema principale affrontato riguarda la crescente violazione dei diritti d’autore e la minaccia dei **deepfake**, ovvero contenuti manipolati tramite **intelligenza artificiale**. Queste problematiche minano l’autenticità dei contenuti e la fiducia nel digitale. La blockchain, grazie alla sua **immutabilità**, offre una soluzione per garantire la protezione e l’autenticazione dei contenuti, contrastando le manomissioni e migliorando la gestione dei diritti digitali.

DEEPFAKE: COSA SONO E SOLUZIONE PROPOSTA

Cosa Sono:

I **deepfake** rappresentano una tecnologia emergente che utilizza **l'intelligenza artificiale** per creare contenuti multimediali manipolati, come video e audio, in cui le immagini e le voci di persone reali vengono imitate in modo estremamente realistico. I deepfake rappresentano una delle sfide più pericolose nell'era digitale, poiché possono essere utilizzati per manipolare l'opinione pubblica e danneggiare la reputazione di individui e organizzazioni.

Soluzione Proposta:

La tesi suggerisce l'uso di **blockchain** e **smart contracts** per proteggere gli artisti dai deepfake. Queste tecnologie garantiscono l'autenticità dei contenuti e permettono agli artisti di mantenere il controllo sulle loro opere, riducendo il rischio di sfruttamento non autorizzato.

Deepfake – Significato, Storia, evoluzione (ICT Security Megazine, 2024)

PROTEZIONE CONTRO I DEEPFAKE

L'applicazione mira a proteggere gli utenti dai **deepfake** attraverso l'uso di **contratti intelligenti** su blockchain e tecnologie di **archiviazione decentralizzata**. Ogni contenuto digitale pubblicato viene associato in modo univoco ad un **smart contract** che ne garantisce l'autenticità e la provenienza, registrando dati chiave come l'autore, l'hash e la data di pubblicazione su blockchain.

Inoltre, l'applicazione implementa un sistema di **permessi** attraverso contratti **Child**, che consente agli artisti di autorizzare modifiche o remix dei loro contenuti, riducendo il rischio di manipolazioni non autorizzate o falsificazioni. I file caricati vengono salvati su IPFS, una rete sicura e decentralizzata, fornendo **CIDs** univoci che verificano l'integrità del contenuto originale.

Questa combinazione di blockchain e IPFS rende più difficile diffondere **contenuti falsificati** o manipolati, offrendo un livello di protezione aggiuntivo contro i deepfake, promuovendo un ambiente digitale più sicuro e trasparente.

PANORAMICA DELLA SOLUZIONE TECNICA

SMART CONTRACT:

Il cuore del progetto è uno **smart contract** che ho programmato utilizzando **Solidity** e la piattaforma **Truffle**. Truffle mi ha permesso di testare il contratto su una rete di sviluppo locale, verificando il corretto funzionamento dei metodi e delle transazioni eseguite. Dopo questi test iniziali, sono passato alla **BSC Testnet** (Binance Smart Chain Testnet) per simulare l'operatività del contratto in un ambiente blockchain reale.

DECENTRALIZED APPLICATION (DApp):

Una volta confermato il corretto funzionamento dello smart contract, ho sviluppato una **dApp** (applicazione decentralizzata) utilizzando **React**, una libreria open-source basata su JavaScript per creare interfacce utente reattive e modulari. Grazie all'integrazione con il framework **Ionic**, l'applicazione è eseguibile sia su browser che su dispositivi mobili Android.

ARCHITETTURA DELLA SOLUZIONE

ARCHITETTURA DELLO SMART CONTRACT:

Lo smart contract è stato progettato utilizzando il pattern **factory**, dove un unico contratto principale viene inizialmente pubblicato sulla blockchain per gestire e tracciare tutte le operazioni all'interno dell'applicazione. Questo contratto factory tiene traccia degli utenti registrati come artisti, consentendo loro di pubblicare nuovi contratti che rappresentano i contenuti digitali che desiderano proteggere.

Tutti gli utenti, sia artisti che non, possono inviare richieste ai contratti dei contenuti per richiederne il riutilizzo. Sarà compito dell'artista, proprietario del contenuto, accettare o rifiutare la richiesta. In caso di accettazione, viene creato un contratto figlio del contratto originale del contenuto, che rappresenta una copia gestita dal richiedente. Questo contratto figlio mantiene un riferimento al contratto padre, garantendo che i permessi concessi siano tracciabili.

Il contratto factory gestisce e memorizza tutte le informazioni necessarie per costruire e aggiornare la User Interface della dApp, permettendo agli utenti di visualizzare lo stato e le interazioni tra i contenuti digitali.

ARCHITETTURA DELLA SOLUZIONE

ARCHITETTURA DELLA DAPP:

CONNESSIONE AL WALLET:

L'applicazione è stata sviluppata in TypeScript utilizzando React e si divide in due componenti principali. Il primo componente gestisce la **connessione con il wallet**, essenziale per identificare il profilo utente e per eseguire transazioni sulla blockchain, consentendo l'interazione con lo smart contract.

Per garantire il funzionamento sia su mobile che su browser, è stata integrata la libreria **WalletConnect**, che si occupa di riconoscere il dispositivo dell'utente e stabilire la connessione appropriata. L'utente, tramite un semplice pulsante, può scegliere tra diversi wallet supportati, nel progetto, il wallet **MetaMask** è stato il più utilizzato e viene raccomandato.

Dopo aver selezionato il wallet, se l'applicazione è in esecuzione su browser verrà aperta l'estensione corrispondente, mentre su mobile si aprirà direttamente l'app del wallet. Durante questa fase, all'utente verrà richiesto di selezionare gli account da connettere. Una volta confermata la scelta, la connessione sarà stabilita, permettendo all'utente di accedere a tutte le funzionalità della dApp.

ARCHITETTURA DELLA SOLUZIONE

ARCHITETTURA DELLA DAPP:

USER INTERFACE – PARTE 1:

Il secondo componente principale della dApp gestisce l'intera **User Interface** per gli utenti connessi tramite wallet. Le informazioni visualizzate sono estratte dallo **smart contract factory** tramite metodi di lettura che non richiedono transazioni.

La UI cambia in base al ruolo dell'utente:

Utente non artista: visualizza i contratti pubblicati dagli artisti, le richieste inviate e i contratti figlio ottenuti da richieste accettate.

Artista: oltre alle funzionalità dell'utente non artista, visualizza i propri contratti, le richieste ricevute e i contratti figlio creati. Gli artisti possono inoltre caricare file sulla rete **IPFS** e creare contratti unici per proteggerli, memorizzando l'hash del file nel contratto.

ARCHITETTURA DELLA SOLUZIONE

ARCHITETTURA DELLA DAPP:

USER INTERFACE – PARTE 2:

Per aggiornare le interfacce utente in tempo reale, viene utilizzato un **eventListener**, un metodo capace di intercettare gli eventi emessi dallo smart contract factory. Questo permette di riconoscere gli eventi che si verificano, come la creazione di un nuovo contratto o una richiesta di utilizzo accettata, e gestirli in modo personalizzato.

Gli utenti possono interagire direttamente con lo smart contract tramite bottoni che eseguono metodi di scrittura sullo smart contract. Queste azioni richiedono una transazione sulla blockchain, che comporta il pagamento delle **gas fees** in criptovalute.

Per garantire la compatibilità dell'applicazione sia su mobile che su browser è stata utilizzata la libreria **Wagmi**, ovvero una libreria integrata con **WalletConnect** progettata per semplificare l'interazione tra le applicazioni decentralizzate (dApp) e gli smart contract su diverse piattaforme tramite gli **hook**.

FUNZIONALITÀ PRINCIPALI DELLO SMART CONTRACT

Struttura dei contratti

```
contract Content{
    // il deploy di uno smart contract corrisponde con la pubblicazione di un video/immagine/canzone/...
    // ogni SC si occupa di autenticare l'originalità del econtent e di gestire richieste eseguite da altri artisti/utenti di riutilizzare tale contenuto
    address private _owner;  // proprietario dello SC
    EContent private thisEContent;   // informazioni sul contenuto digitale
    Factory private factory; // contratto factory che ha eseguito il deploy di questo SC
    bool private freeze;    // se vale true il contratto non potrà ricevere nuove richieste

    struct EContent {
        address owner;  // proprietario del econtent
        bytes32 IPFS_Hash;    // IPFS hash del econtent
        address SC_address;   // indirizzo SC associato all' econtent
        uint256 timestamp;    // data pubblicazione
    }
```

Il contratto **Content** rappresenta univocamente un contenuto digitale e memorizza informazioni fondamentali su di esso: il proprietario, l'Hash del contenuto, l'indirizzo dello smart contract e il timestamp di creazione del blocco sulla blockchain.

Il contratto **ChildContent**, invece, rappresenta univocamente un permesso concesso per modificare un contenuto digitale precedentemente pubblicato da un artista. Oltre a memorizzare gli stessi dati del contratto Content, include anche un riferimento al contenuto originale (padre), garantendo così la tracciabilità della relazione tra i contenuti e i permessi concessi per le modifiche.

```
contract ChildContent {
    // il contratto figlio non può ricevere ed elaborare richieste
    address private _owner;  // proprietario dello SC
    ChildEContent private childContent;  // informazioni del contratto figlio

    struct ChildEContent {
        address owner;  // proprietario del econtent
        bytes32 IPFS_Hash;    // IPFS hash del econtent
        address SC_address;   // indirizzo SC associato all' econtent
        uint256 timestamp;    // data pubblicazione
        Content.EContent parent_EC;    // informazioni del contratto genitore
    }
```

Registrazione Artisti

```
function registerArtist() public {
    require(!isArtist[msg.sender], "Artista presente in lista");    // per registrare un nuovo artista questo non può essere già registrato

    isArtist[msg.sender] = true; // imposto il valore dell'artista sulla mappa a true
    emit ArtistAdded("Artista aggiunto in lista: (artistAddress)", msg.sender); // artista aggiunto con successo
}
```

La funzione **registerArtist()** permette a un utente di registrarsi come artista. Utilizza un **mapping** (address => bool), che funziona come una hash map, dove la chiave è l'indirizzo dell'utente e il valore è un booleano. Solo gli artisti registrati avranno **true** come valore. Il metodo verifica che l'utente non sia già registrato come artista prima di procedere. Alla fine dell'esecuzione, viene emesso un **evento** che registra il cambiamento sulla blockchain. Essendo una funzione public, può essere chiamata da chiunque.

Creazione del contratto

```
// esegue il deploy di uno smart contract associandolo all'artista che ha chiamato il metodo
function createContract(bytes32 IPFSHash) public {
    require(isArtist[msg.sender], "Artista non presente in lista");    // solo artisti registrati possono eseguire il deploy di contratti
    require(artistContracts[msg.sender].length < MAX_LENGTH, "Non possibile creare un contratto, limite raggiunto"); // limito la quantità di contratti possedibili da un artista

    address newContract = address(new Content(msg.sender, IPFSHash), address(this));    // creazione contratto con proprietario il chiamante del metodo e l'hash del contenuto
    artistContracts[msg.sender].push(newContract);    // aggiungo il contratto alla lista dei contratti posseduti dall'artista
    contracts.push(newContract);    // aggiungo il contratto alla lista di tutti i contratti
    authorizedContracts[newContract] = true;  // autorizzo il contratto a poter interagire con il factory
    emit DeployedContracts("Contratto pubblicato: (owner, contractAddress)", msg.sender, newContract);    // deploy del contratto completato
}
```

La funzione **createContract(bytes32)** consente agli artisti di pubblicare un nuovo smart contract, rappresentante un contenuto digitale di loro proprietà che desiderano registrare. Il contratto viene creato utilizzando l'indirizzo dell'artista, che diventa il proprietario del contratto e del contenuto, l'hash del contenuto memorizzato sulla rete IPFS, e l'indirizzo del contratto factory che lo ha generato. L'indirizzo del nuovo contratto viene salvato in un mapping che memorizza gli array dei contratti posseduti da ogni artista, oltre a essere aggiunto a una lista generale di tutti i contratti pubblicati. Per eseguire il metodo, l'utente deve essere registrato come artista, e il numero di contratti pubblicati deve essere inferiore a un limite prestabilito, introdotto per evitare che la crescita degli array sovraccarichi il sistema. Alla fine dell'esecuzione, viene emesso un evento che notifica la pubblicazione del contratto da parte dell'artista.

Richiesta permesso

```
// richiesta permesso di riutilizzo di un EContent - il proprietario non può fare richiesta per un suo contenuto
function request() public notOwner{
    require(requests[msg.sender].state == artistState.NoReq, "Ammessa una sola richiesta per utente");    // un utente può fare una sola richiesta per contenuto
    require(freeze == false, "Contratto congelato");   // esegue solo le il contratto non è stato bloccato

    requests[msg.sender].state = artistState.ReqInviata; // aggiorno lo stato della richiesta e ne salvo le informazioni
    requests[msg.sender].EA = msg.sender;
    requests[msg.sender].thisContract = address(this);
    requests[msg.sender].hash = thisEContent.IPFS_Hash;
    requests[msg.sender].result = false;

    bytes32 requestKey = keccak256(abi.encodePacked(msg.sender, address(this), _owner));
    factory.addArtistRequestsReceived(_owner, requests[msg.sender], requestKey);    // aggiungo la richiesta alla lista di quelle ricevute dal proprietario
    factory.addArtistRequestsSent(msg.sender, requests[msg.sender], requestKey);    // aggiungo la richiesta alla lista di quelle inviate dall'artista
    factory.RegisteredReqEvent(msg.sender, address(this), _owner); // comunico l'avvenuta registrazione della richiesta
    emit Request("Richiesta registrata: ", requestKey);
}
```

La funzione **request()** permette a un utente di richiedere il permesso per riutilizzare un contenuto direttamente al contratto associato. Vengono salvati lo stato della richiesta (inizialmente "inviata"), l'indirizzo del richiedente, l'indirizzo del contratto, l'hash del contenuto e un flag che indica se la richiesta è stata accettata. Le richieste vengono archiviate in due mapping del contratto factory che salvano per ogni artista un array, uno per le richieste inviate e uno per quelle ricevute. Viene calcolato anche un hash della richiesta che viene usato per salvare la posizione nell'array, utile per quando bisognerà rispondere. Il metodo può essere chiamato solo da utenti che non sono proprietari del contratto e ciascun utente può fare una sola richiesta per contenuto. Inoltre, il contratto deve essere attivo, ovvero non congelato.

Risposta alla richiesta

```
// il proprietario dello SC può rispondere alle richieste, accettandole o rifiutandole - richiamabile sono dal proprietario
function reply(bool result, address senderAddress) public onlyOwner{
    require(requests[senderAddress].state == artistState.ReqInviata, "Richiesta non in attesa di risposta");   // valuto solo richieste che necessitano di risposta

    if(result){ // richiesta accettata
        requests[senderAddress].state = artistState.ReqAccettata;  // aggiorno lo stato della richiesta nella lista
        requests[senderAddress].result = result;
        factory.GrantedPermEvent(senderAddress, address(this), _owner); // comunico l'esito
        factory.createChildContract(senderAddress, msg.sender, thisEContent.IPFS_Hash, thisEContent.SC_address, thisEContent.timestamp); // richiamo il metodo del contratto factory che crea il figlio
    }
    else {  // richiesta rifiutata
        requests[senderAddress].state = artistState.ReqNegata;   // aggiorno lo stato della richiesta generale
        factory.DeniedPermEvent(senderAddress, address(this), _owner); // comunico l'esito
    }
    emit Reply("Risposta inviata: ", keccak256(abi.encodePacked(senderAddress, address(this), _owner)));
}
```

La funzione **reply(bool, address)** permette al proprietario del contratto di rispondere a una richiesta di utilizzo di contenuto. L'input include una variabile booleana, che indica se la richiesta viene accettata o rifiutata, e l'indirizzo dell'utente che l'ha inviata. A seconda della decisione, lo stato e il flag della richiesta vengono aggiornati, e viene emesso un evento per notificare l'esito (accettato o rifiutato). Se la richiesta è accettata, viene invocato un metodo nel contratto factory per creare un contratto figlio. Il metodo può essere eseguito solo dal proprietario del contratto e richiede che la richiesta sia in attesa di risposta.

```
function GrantedPermEvent(address artist, address contractAddress, address owner) external {
    require(authorizedContracts[msg.sender], "Contratto non autorizzato per concedere permessi");    // solo contratti autorizzati possono concedere permessi
    bytes32 requestKey = keccak256(abi.encodePacked(artist, contractAddress, owner));
    artistRequestsReceived[owner][indexArtistRequestsReceived[requestKey]].result = true;    // aggiorno la richiesta ricevuta con l'esito positivo
    artistRequestsReceived[owner][indexArtistRequestsReceived[requestKey]].state = artistState.ReqAccettata;    // aggiorno lo stato della richiesta ricevuta
    artistRequestsSent[artist][indexArtistRequestsSent[requestKey]].result = true;    // aggiorno la richiesta inviata con l'esito positivo
    artistRequestsSent[artist][indexArtistRequestsSent[requestKey]].state = artistState.ReqAccettata;    // aggiorno lo stato della richiesta inviata
    emit GrantedPerm("Permesso concesso (senderAddress, contractAddress, owner)", artist, contractAddress, owner);
}

function DeniedPermEvent(address artist, address contractAddress, address owner) external {
    require(authorizedContracts[msg.sender], "Contratto non autorizzato per negare permessi");    // solo contratti autorizzati possono negare permessi
    bytes32 requestKey = keccak256(abi.encodePacked(artist, contractAddress, owner));
    artistRequestsReceived[owner][indexArtistRequestsReceived[requestKey]].state = artistState.ReqNegata;    // aggiorno la richiesta ricevuta con l'esito negativo
    artistRequestsSent[artist][indexArtistRequestsSent[requestKey]].state = artistState.ReqNegata;    // aggiorno la richiesta inviata con l'esito negativo
    emit DeniedPerm("Permesso negato (senderAddress, contractAddress, owner)", artist, contractAddress, owner);
}
```

Creazione contratto figlio

```
function createChildContract(address senderAddress, address parentArtist, bytes32 IPFShash, address parentContract, uint256 timestamp) external {
    require(authorizedContracts[msg.sender], "Contratto non autorizzato per creare un figlio");    // solo contratti autorizzati possono creare contratti figli
    ChildContent child = new ChildContent(senderAddress, parentArtist, IPFShash, parentContract, timestamp);   // creo il contratto figlio
    childContractsObtained[senderAddress].push(Child({
        childOwner: senderAddress,
        childAddress: address(child),
        parentArtist: parentArtist,
        parentContract: parentContract
    }));  // aggiunge l'indirizzo del figlio ottentuto alla mappa dell'utente
    childContractsGiven[parentArtist].push(Child({
        childOwner: senderAddress,
        childAddress: address(child),
        parentArtist: parentArtist,
        parentContract: parentContract
    }));  // aggiunge l'indirizzo del figlio concesso alla mappa del padre
    emit ChildCreated("Contratto figlio pubblicato: (childOwner, childAddress, parentArtist, parentContract)", senderAddress, address(child), parentArtist, parentContract); // contratto figlio creato
}
```

La funzione **createChildContract(address, address, bytes32, address, uint256)** crea un nuovo contratto figlio dopo che una richiesta di riutilizzo è stata accettata. Questo contratto figlio viene assegnato all'utente la cui richiesta è stata approvata, memorizzando anche l'indirizzo del contratto padre, ovvero il contratto a cui era stata inviata la richiesta originale. Le informazioni sul contratto figlio vengono salvate in due mapping: uno per i contratti figli ricevuti e uno per quelli concessi. Al termine, viene emesso un evento che conferma la pubblicazione del contratto figlio. Il metodo è external, quindi può essere richiamato solo da account o contratti esterni, ma non da altre funzioni dello stesso contratto. Per rendere il metodo eseguibile solo se chimato dai contratti Content, è stato inserito un controllo se il chiamante è presente nel mapping dei contratti pubblicati.

Congelamento del contratto

```
function freezeContract() public onlyOwner() {  // congela contratto - non potrà ricevere nuove richieste
    require(freeze == false);   // non posso congelare un contratto già congelato

    freeze = true;
    factory.addFreezedContracts(address(this), msg.sender);    // aggiungo il contratto alla lista di quelli bloccati
}
```

```
function addFreezedContracts(address contractAddress, address owner) external {
    require(authorizedContracts[msg.sender], "Contratto non autorizzato per congelare contratti");    // solo contratti autorizzati possono congelare contratti
    freezedContracts.push(contractAddress);
    artistFreezedContracts[owner].push(contractAddress);
    emit FreezedContract("Contratto congelato (owner, contractAddress)", owner, contractAddress);
}
```

La funzione **freezeContract()** consente al proprietario del contratto di congelarlo, impedendo la ricezione di nuove richieste relative a quel contenuto. Una volta che un contratto è stato congelato, non può più essere scongelato in alcun modo. Per tracciare gli indirizzi dei contratti congelati, viene richiamata la funzione **addFreezedContracts(address, address)** del contratto Factory. Questa funzione aggiunge l'indirizzo del contratto congelato a una lista generale e a un array all'interno di un mapping, così da poter associare anche il proprietario del contratto. Inoltre, essendo un metodo di tipo external, può essere richiamato solo esternamente dal contratto Factory. Per garantire che solo i contratti Content possano invocare questo metodo, viene eseguito un controllo per verificare che il chiamante appartenga alla lista di questi contratti.

INTERFACCIA UTENTE (UI)

Event Listener

```
}else if(log.topics[0] == "0xfe85a440eeb955db02cb6a979383619bf804fb3a73ecadfc54b024a40294a12a"){  // gestisco l'evento Request
  const senderAddress = (listenerProv as any).utils.toChecksumAddress(extractAddress(log.data, 1));
  const contractAddress = (listenerProv as any).utils.toChecksumAddress(extractAddress(log.data, 2));
  const ownerAddress = (listenerProv as any).utils.toChecksumAddress(extractAddress(log.data, 3));
  console.log("New request:\nSender address:", senderAddress + "\nContract address:", contractAddress + "\nOwner address:", ownerAddress);
  if(senderAddress == address){ // se il mittente della richiesta è l'utente attuale
    setAlert({ isOpen: true, backdropDismiss: true, header: 'Request sent', message: 'Request sent to contract with address:\n' + contractAddress });
    setState((prevState) => {
      const exists = prevState.requestSent.some(request => request.thisContract === contractAddress && request.EA === senderAddress && request.state === '1');
      if(!exists){  // se la richiesta non è già presente nella lista
        return {
          ...prevState,
          requestSent: [...prevState.requestSent, {EA: senderAddress, thisContract: contractAddress, state: '1'}],  // aggiungo la richiesta alla lista delle richieste inviate dall'utente attuale
        };
      } else {
        return prevState; // altrimenti restituisco lo stato precedente
      }
    });
  }else if (ownerAddress == address){ // se il proprietario del contratto è l'utente attuale
    setAlert({ isOpen: true, backdropDismiss: true, header: 'Request received', message: 'Request received from:\n' + senderAddress + '\nFor contract:\n' + contractAddress });
    setState((prevState) => {
      const exists = prevState.requestReceived.some(request => request.EA === senderAddress && request.thisContract === contractAddress && request.state === '1');
      if(!exists){  // se la richiesta non è già presente nella lista
        return {
          ...prevState,
          requestReceived: [...prevState.requestReceived, {EA: senderAddress, thisContract: contractAddress, state: '1'}],  // aggiungo la richiesta alla lista delle richieste ricevute dall'utente attuale
        };
      } else {
        return prevState; // altrimenti restituisco lo stato precedente
      }
    });
  }
```

Questa parte di codice è interna al metodo handleLogs, ovvero una funzione che viene richiamata quando **l'eventListener** intercetta un nuovo log emesso dallo smart contract. I log degli eventi sono identificati da un **hash**, conoscendolo è possibile intercettare gli eventi e gestirli in modo personalizzato. In questo caso gestisco l'evento di avvenuta richiesta, aggiornando in tempo reale la UI degli utenti coinvolti, ovvero del sender e del receiver. Inoltre, per segnalare l'invio o la ricezione viene usato un **alert** a schermo.

User Interface

```
<IonItem style={{ fontSize: `${fontSize}px` }}>
  <IonGrid>
    <IonRow>
      <IonCol>
        <IonLabel style={{ fontWeight: 'bold', fontSize: '1.2em' }}>Contratti a cui poter fare richiesta</IonLabel>
      </IonCol>
    </IonRow>
    <IonRow>
      <IonCol>
        <IonCheckbox justify="start" checked={showHidden} onIonChange={handleCheckboxChange}>Visualizza nascosti</IonCheckbox>
      </IonCol>
    </IonRow>
  </IonGrid>
</IonItem>
{filteredContractAddresses.length === 0 ? (
  <IonItem style={{ fontSize: `${fontSize}px` }}><IonLabel>Lista vuota</IonLabel></IonItem>
) : (
  <IonList>
    {filteredContractAddresses.map((address, index) => (
      <IonItem key={index} style={{ fontSize: `${fontSize}px` }}>
        {state.requestSent.some(request => request.thisContract === address) ? (
          <IonLabel>
            <a href="">{address}</a> <br /> Richiesta già inviata per questo contratto
          </IonLabel>
        ) : state.freezedContracts.includes(address) ? (
          <IonLabel>
            <a href="">{address}</a> <br /> Contratto congelato
          </IonLabel>
        ) : (
          <IonLabel>
            <a href="">{address}</a> <br />
            <IonButton onClick={() => requestPermission(address)}>Richiedi permesso</IonButton>
          </IonLabel>
        )}
      </IonItem>
    ))}
  </IonList>
)}
```

Questa componente **React** con **Ionic** fa parte del codice che si occupa della visualizzazione della **User Interface**. Questa specifica porzione si occupa della visualizzazione dei contratti a cui un determinato utente può fare richiesta, consentendogli di rimuovere i nascosti dalla visualizzazione, ovvero i contratti a cui non può inviare la richiesta perché congelati o perché ne ha già effettuata una verso di loro. Il codice visualizzato funziona sia per browser e sia per mobile Android.

Letture sul contratto

```
// funzioni per interagire con il contratto
const { data: isReg } = useReadContract({
  abi: FactoryAbi,
  address: factoryAddress,
  functionName: 'getIsArtist',  // funzione del contratto che restituisce se l'utente è registrato come artista
  args: [address],
});

const { data: contractAddressesList } = useReadContract({
  abi: FactoryAbi,
  address: factoryAddress,
  functionName: 'getContracts', // funzione del contratto che restituisce la lista degli indirizzi dei contratti pubblicati dagli artisti
});

const { data: myContracts } = useReadContract({
  abi: FactoryAbi,
  address: factoryAddress,
  functionName: 'getArtistContracts', // funzione del contratto che restituisce la lista degli indirizzi dei contratti pubblicati dall'utente attuale
  args: [address],
});

const filteredContractAddressesList = (contractAddressesList as string[])?.filter((address: string) => !(myContracts as string[])?.includes(address));
```

In questa parte del codice, viene gestita l'interazione in lettura con lo smart contract. Tramite delle funzioni della libreria **wagmi**, l'applicazione effettua chiamate per ottenere informazioni memorizzate sul contratto, come ad esempio lo stato dell'utente o liste di indirizzi. Queste chiamate permettono di accedere ai dati registrati sul contratto in modo da poterli visualizzare nella **UI**. Poiché i metodi sono solo in lettura **non richiedono l'uso di transazioni** e quindi non costano gas fees. L'utilizzo di wagmi è stato l'unico modo possibile per far funzionare l'interazione con lo smart contract sia su browser che su mobile Android.

Scritture sul contratto

```
const registerArtist = async () => {
  try {
    sendTransaction({ ...config, to: factoryAddress, data: factory.methods.registerArtist().encodeABI() as '0x${string}' });
  } catch (error) {
    console.error("Error registering artist: ", error);
    setAlert({ isOpen: true, backdropDismiss: true, header: 'Error', message: 'Error registering artist' });
  }
};

const requestPermission = async (contentAddress: string) => {
  if (!listenerProv) {
    console.error('Web3 is not available.');
    return;
  }
  try {
    const contentContract = new listenerProv.eth.Contract(ContentAbi as any, contentAddress);
    sendTransaction({ ...config, to: contentAddress as '0x${string}', data: contentContract.methods.request().encodeABI() as '0x${string}' });
  } catch (error) {
    console.error("Error sending request: ", error);
    setAlert({ isOpen: true, backdropDismiss: true, header: 'Error', message: 'Error sending request' });
  }
};
```

In questa parte di codice sono presenti le funzioni dedicate alla scrittura sullo smart contract tramite l'esecuzione di transazioni. Queste funzioni sono di tipo **async** poiché l'esecuzione della transazione non è istantanea, dunque è necessario attendere il ritorno asincrono del metodo sendTransaction. Questo metodo sfrutta la libreria **wagmi**, utile per rendere il codice funzionante sia su mobile Android che su browser. Nel primo caso, viene inviata una transazione al contratto **Factory**, di cui si conosce l'indirizzo (factoryAddress), per registrare un artista. Nel secondo caso, la transazione è inviata a un contratto **Content**, per richiedere un permesso di modifica. Prima di effettuare la transazione, è necessario recuperare l'indirizzo del contratto Content dalla blockchain (nella rete BSCTestnet) per ottenerne l'istanza.

Caricamento file su IPFS

```
// funzioni per l'upload su IPFS
async function createHeliaNode() {
  const blockstore = new MemoryBlockstore();  // crea un blockstore in memoria

  const helia = await createHelia({ blockstore });  // crea un nodo Helia con il blockstore in memoria

  const fs = unixfs(helia); // crea un filesystem UnixFS

  return { helia, fs }; // restituisci il nodo Helia e il filesystem UnixFS
}

async function uploadToIPFS(file: File) {
  const { fs } = await createHeliaNode(); // crea un nodo Helia e un filesystem UnixFS

  const fileArrayBuffer = await file.arrayBuffer(); // leggi il file come ArrayBuffer
  const fileBytes = new Uint8Array(fileArrayBuffer);  // converti l'ArrayBuffer in Uint8Array

  const cid = await fs.addBytes(fileBytes); // aggiungi il file al filesystem UnixFS e ottieni il CID

  console.log('File caricato con successo. CID:', cid.toString());

  return cid.toString();  // restituisci il CID in formato stringa
}

const handleUpload = async () => {
  if (!file) {  // se non è stato selezionato alcun file
    setAlert({ isOpen: true, backdropDismiss: true, header: 'Error', message: 'You need to select a file' });
    return;
  }

  console.log(file.name);
  const cid = await uploadToIPFS(file); // carica il file su IPFS e ottieni il CID
  if (cid) {  // se il CID è stato ottenuto
    setState((prevState) => ({
      ...prevState,
      IPFSHash: cid,  // aggiorna lo stato con il CID del file
    }));
    createContract(cid);  // crea un contratto con il CID del file
    setAlert({ isOpen: true, backdropDismiss: true, header: 'File uploaded', message: 'CID: '+ cid });
  }
};
```

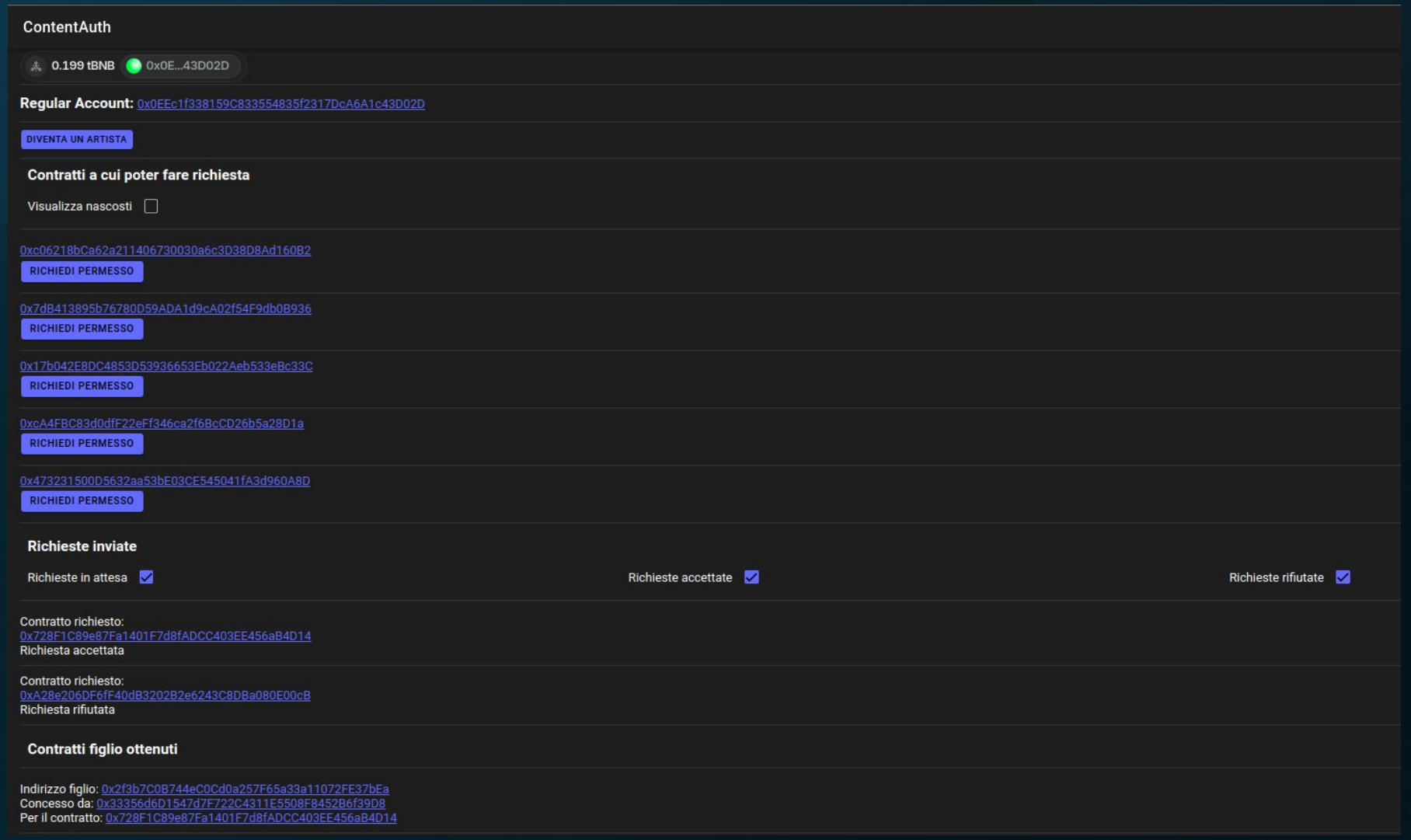
Questo codice gestisce il caricamento di file su **IPFS** (InterPlanetary File System) per garantire un'archiviazione decentralizzata e sicura dei contenuti digitali. IPFS viene usato perché permette di conservare file in modo distribuito, fornendo un **CID** (Content Identifier) unico che garantisce l'integrità del file e permette di identificarlo ovunque sulla rete.

Per interagire con IPFS, viene utilizzata **Helia**, una libreria che facilita la connessione e la gestione dei nodi IPFS. La funzione createHeliaNode crea un nodo Helia con un "**blockstore**" in memoria, necessario per mantenere temporaneamente i blocchi di dati da caricare su IPFS.

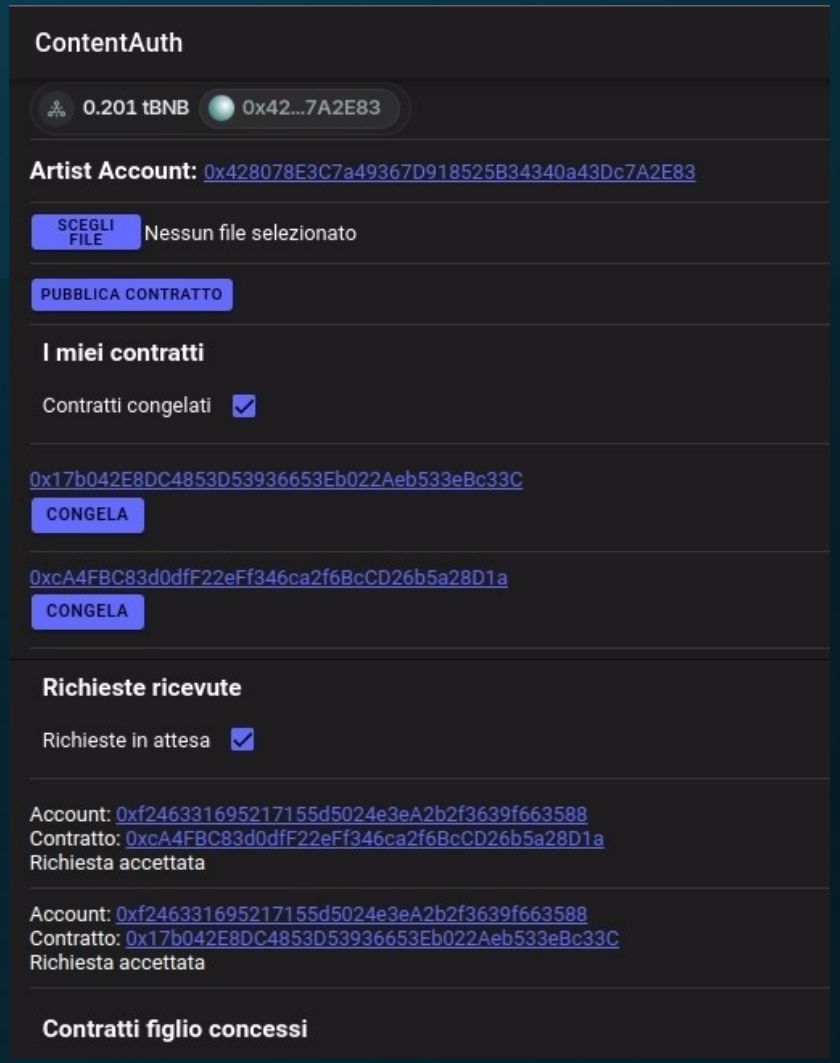
Il file system **UnixFS** viene usato per gestire i file da caricare, consentendo di lavorare con i dati in formato compatibile con IPFS. La funzione uploadToIPFS carica effettivamente il file, lo converte in byte, e lo aggiunge al file system di IPFS, ottenendo il CID.

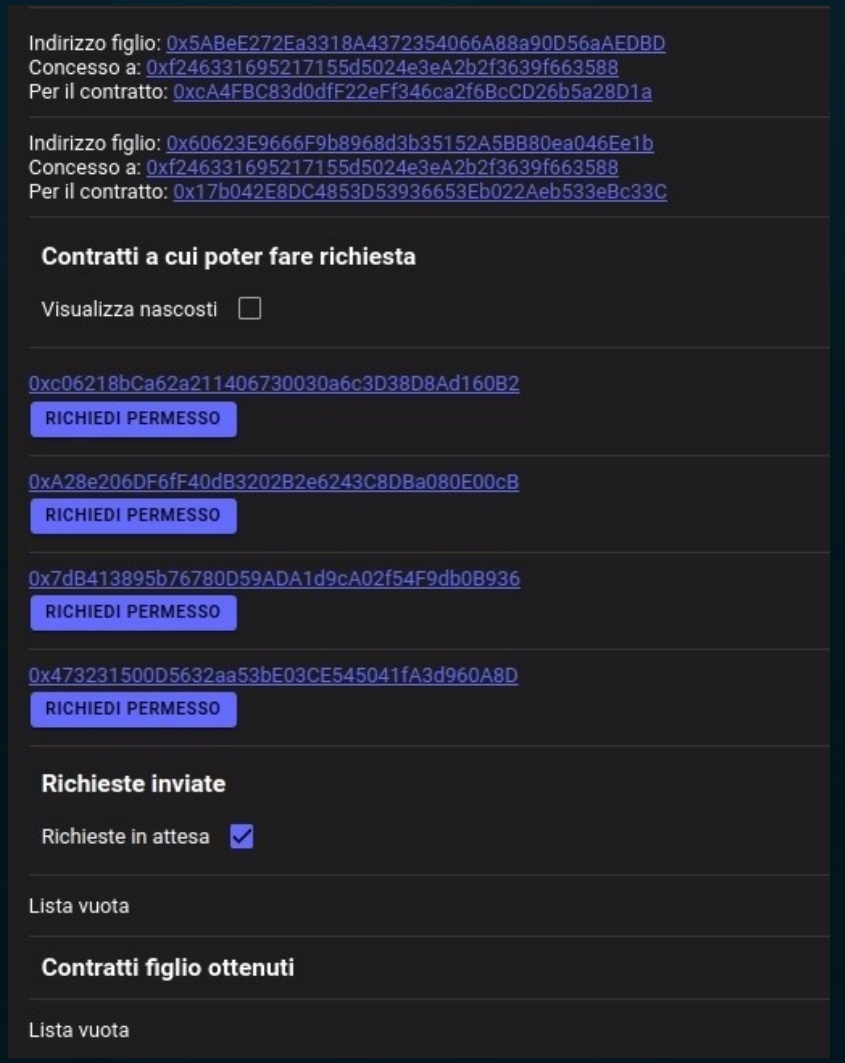
Questo CID viene poi passato al metodo **createContract** per poter creare un contratto associato al file caricato su IPFS.

FLUSSO DI LAVORO
Utenti non registrati



FLUSSO DI LAVORO
Utenti registrati come Artisti





CONCLUSIONI E FUTURI SVILUPPI

I futuri sviluppi che ho pensato per l'applicazione si concentrano sull'espansione delle funzionalità e sul miglioramento dell'esperienza dell'utente. In particolare:

- **Miglioramento dell'interfaccia utente**: semplificare l'interfaccia per renderla più intuitiva e accessibile, così da facilitare l'utilizzo da parte di utenti meno esperti. Associare agli indirizzi degli artisti dei nomi chiari e riconoscibili, e agli address dei contratti un link diretto per visualizzare il contenuto su IPFS. Migliorare la visualizzazione dei contratti pubblicati, consentire di effettuare ricerche e ordinamenti su questi. Visualizzare informazioni aggiuntive riguardo i contenuti e gli utenti.
- **Maggiore controllo del padre,** consentire all'artista che ha accettato una richiesta di poter revocare e gestire l'autorizzazione concessa.
- **Aggiungere un sistema di verifica basato su reputazione**: ad ogni utente e ad ogni artista verrà associato un punteggio di affidabilità basato sui feedback ricevuti dagli altri con cui ha interagito. Gli utenti potranno facilmente identificare contenuti affidabili grazie a questi punteggi, migliorando la fiducia nel sistema e aiutando gli artisti a proteggersi da deepfake e violazioni dei diritti d'autore.
- **Espandere l'uso della tecnologia cross-chain**: integrare altre blockchain per rendere i contenuti accessibili e verificabili su più reti, migliorando la scalabilità dell'applicazione.

DEMO TECNICA