Ethereum Payment App: Sistema Descentralizado de Partilha de Despesas

Gustavo Marques Pina m15838, Rui Gil m15777, Rodrigo Santos e11654

9 de janeiro de 2026

Capítulo 1

Introdução

Neste relatório é apresentada a implementação de uma Aplicação Descentralizada (dApp) na rede Ethereum, desenvolvida no âmbito da unidade curricular de Blockchains e Criptomoedas. O objetivo deste trabalho consiste no desenvolvimento de um sistema de gestão de dívidas e créditos, inspirado na aplicação *Splitwise*, que permite aos utilizadores saber quem deve dinheiro a quem de forma transparente e imutável.

O projeto foi desenvolvido utilizando a *stack* tecnológica ***Scaffold-ETH 2***, combinando um *backend* escrito em **Solidity** (Hardhat) com um *frontend* em **React/Next.js**. Um dos grandes desafios centrais abordados foi a otimização de custos de gás, transferindo assim a lógica complexa de resolução de ciclos de dívida (*Loop Resolution*) para o cliente, mantendo o contrato inteligente leve e eficiente.

1.1 Arquitetura da Solução

A arquitetura do sistema divide-se em dois componentes principais:

1. **Smart Contract (Backend):** Responsável por armazenar o estado das dívidas (IOUs - *I Owe You*) e a lista de utilizadores. Foi desenhado para ser minimalista, servindo como uma "fonte de verdade"imutável.

2. **Frontend (Cliente):** Interface web que interage com a carteira do utilizador (ex: MetaMask/Burner Wallet). É aqui que reside a lógica algorítmica de deteção de ciclos, utilizando um algoritmo de Busca em Largura (BFS) para reduzir dívidas desnecessárias antes de interagir com a blockchain.

Capítulo 2

Fase 1: Smart Contract

O núcleo da arquitetura descentralizada reside no contrato inteligente `mycontract.sol`. Ao contrário de abordagens simplistas que delegam a lógica para o *frontend*, esta implementação adota uma arquitetura de **resolução on-chain**. O contrato atua não apenas como base de dados, mas como o motor de processamento que garante a resolução atómica e imediata de ciclos de dívida.

Esta decisão de design, embora acarrete maiores custos computacionais (*gas costs*), assegura que o estado da rede está sempre otimizado e livre de redundâncias financeiras, independentemente da interface que interage com o contrato.

2.1 Estruturas de Dados e Otimização

Para suportar a lógica complexa de grafos na blockchain, abandonou-se o mapeamento bidimensional simples em favor de uma estrutura baseada em *structs*. Esta abordagem permite uma manipulação mais granular das arestas do grafo de dívidas.

A escolha do tipo de dados `uint32` (4 bytes) para os valores monetários foi mantida e tornou-se ainda mais relevante nesta nova arquitetura, permitindo o empacotamento eficiente dentro da *struct*.

```
struct DebtNode {
    uint32 value;
    address creditor;
}

// Lista de adjacencia: Endereco -> Lista de Dividas
mapping(address => DebtNode[]) public debtGrap;
```

Listing 2.1: Estrutura de Dados baseada em Structs

Esta estrutura `Owing[]` funciona efetivamente como uma lista de adjacência de um grafo direcionado, onde cada nó (utilizador) aponta para os seus credores com um peso associado (montante).

2.2 Gestão de Utilizadores e Iterabilidade

A necessidade de percorrer o grafo para detetar ciclos impôs a criação de um sistema robusto de gestão de participantes. O contrato mantém duas estruturas sincronizadas:

1. `address[] participantList`: Um *array* que permite iterar sobre todos os nós do grafo durante a execução do algoritmo de resolução.

2. `mapping(address => bool) participant`: Um mapeamento para verificação $O(1)$ de existência, evitando a inserção de duplicados na lista.

Esta dualidade é essencial: sem a `participantList`, seria impossível para o algoritmo *on-chain* visitar todos os nós para procurar ciclos.

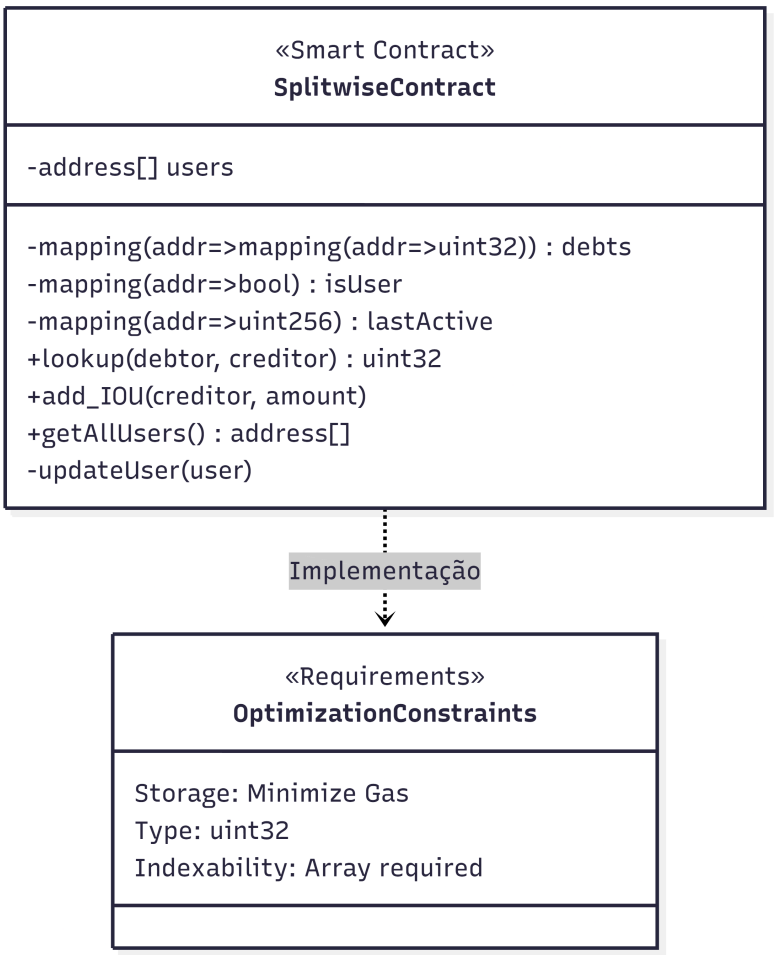

Figura 2.1: Arquitetura de armazenamento do contrato.

2.3 Lógica de Escrita: Função `iou`

A função principal, renomeada para `iou`, é o cérebro do sistema. Ao contrário de uma simples operação de adição, esta função executa três passos críticos atomicamente:

2.3.1 Verificação de Dívida Inversa (Otimização Imediata)

Antes de registar uma nova dívida de $A \to B$, o contrato verifica se já existe uma dívida de $B \to A$.

- Se B deve a A, o sistema abate o valor imediatamente ("netting").

- Se a dívida inversa cobrir o novo valor, nenhuma nova entrada é criada, poupando armazenamento.

2.3.2 Persistência da Nova Dívida

Caso a dívida inversa não exista ou seja insuficiente, a nova dívida é adicionada à lista de adjacência (`owingData`) do devedor.

2.3.3 Resolução de Ciclos (Algoritmo On-Chain)

Após a atualização do estado, é invocada a função privada `resolveDebtLoops()`. Este algoritmo percorre o grafo de participantes à procura de caminhos fechados ($A \to B \to C \to A$). Quando um ciclo é detetado:

1. Identifica-se a aresta com o menot valor (o valor mínimo da dívida no ciclo).
2. Subtrai-se esse valor a todas as transações do ciclo.
3. As dívidas que chegam a zero são removidas da memória para recuperar gás (*Gas Refund*).

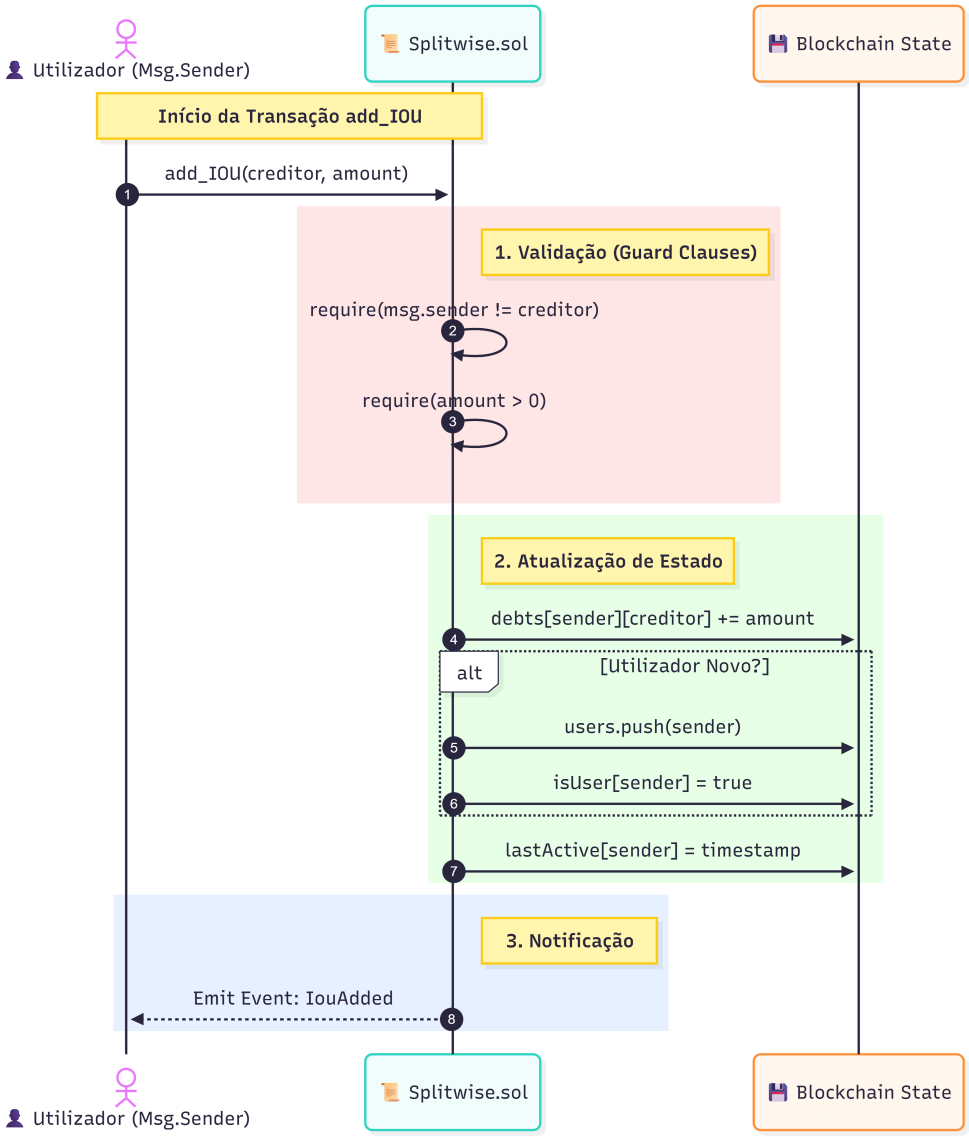

Figura 2.2: Fluxo de execução da função `iou`: Abate direto e resolução de ciclos recursiva.

2.4 Considerações sobre Deploy

O processo de *deploy* manteve-se inalterado, utilizando o *Hardhat* para orquestrar a rede local. A utilização do comando `yarn deploy -reset` revelou-se fundamental durante os testes para garantir que a estrutura de dados complexa (*mappings* de *structs*) fosse reiniciada corretamente, evitando estados inconsistentes ("dívidas fantasma") entre execuções de teste.

Capítulo 3

Fase 2: Lógica do Cliente

A complexidade algorítmica do projeto reside no cliente (`page.tsx`), desenvolvido em TypeScript/React. O enunciado exigia a implementação de um mecanismo de **Resolução de Ciclos de Dívida** para evitar transações desnecessárias.

3.1 O Problema dos Ciclos

Considere-se o cenário: Alice deve 10 a Bob, Bob deve 10 a Carol, e Carol empresta 10 a Alice. Se registássemos esta última transação, criar-se-ia um ciclo fechado ($A \to B \to C \to A$). Em vez de aumentar a dívida total do sistema, o algoritmo deve detetar este ciclo e anular as dívidas mutuamente.

3.2 Resolução de Ciclos

A resolução de ciclos no sistema é realizada através de uma colaboração entre o cliente e o contrato inteligente. Antes de submeter uma nova dívida, o cliente aplica o algoritmo Breadth-First Search para determinar se a operação fecha um ciclo no grafo de dívidas. Caso exista um caminho entre o credor e o devedor, o cliente identifica o ciclo e calcula o valor mínimo presente nas arestas envolvidas. Esta informação é então enviada ao contrato, que valida a estrutura recebida e aplica a resolução on-chain, subtraindo o mínimo a todas as dívidas do ciclo. Esta abordagem distribui o esforço computacional, mantendo a eficiência no cliente e garantindo a integridade e segurança da atualização final no contrato.

3.3 Implementação da Lógica Splitwise no Cliente

Nesta secção descrevemse as funções desenvolvidas no lado do cliente responsáveis por suportar a lógica da aplicação Splitwise. Embora a resolução final dos ciclos de dívida seja realizada no contrato inteligente, conforme os requisitos de segurança do enunciado, toda a componente de deteção, análise e preparação dos dados é efetuada *off-chain*. Esta separação cumpre o princípio estabelecido no projeto: a inteligência deve residir no cliente, enquanto a execução segura deve ocorrer no contrato.

3.3.1 Obtenção da lista de utilizadores

O componente UsersList é responsável por obter, a partir do contrato inteligente, a lista completa de utilizadores que já interagiram com o sistema. Utiliza a função getAllUsers exposta pelo contrato e disponibiliza estes dados ao frontend, permitindo a visualização dinâmica dos participantes do grafo de dívidas. Este componente funciona como um ponto de ligação entre o estado on-chain e a interface, assegurando que a lista de utilizadores é carregada e atualizada de forma eficiente.

```
export const UsersList = () => {
  const { data: usersData, isLoading } = useScaffoldReadContract({
    contractName: "Splitwise",
    functionName: "getAllUsers",
  });

  const users = (usersData as readonly string[] | undefined) ?? [];
```

3.3.2 Listas de Dívidas

O componente DebtList é responsável por apresentar, para um determinado utilizador, todas as dívidas ativas registadas no contrato inteligente. Para tal, obtém a lista completa de utilizadores através da função getAllUsers e, para cada um deles, instancia o subcomponente DebtItem, que consulta diretamente o contrato para verificar se existe uma dívida positiva entre o utilizador atual e o credor em análise. Apenas dívidas não nulas são apresentadas, garantindo uma visualização limpa e eficiente do estado do grafo de dívidas.

```
export const DebtList = ({ address }: DebtListProps) => {
  // Obter lista de utilizadores
  const { data: usersData } = useScaffoldReadContract({
    contractName: "Splitwise",
    functionName: "getAllUsers",
  });

  const users = (usersData as readonly string[] | undefined) ?? [];

  // Componente para cada d vida individual
  const DebtItem = ({ creditorAddress }: { creditorAddress: string }) =>
    {
    const { data: amount } = useScaffoldReadContract({
      contractName: "Splitwise",
      functionName: "lookup",
      args: [address as `0x${string}`, creditorAddress as `0x${string
   }`],
    });

    if (!amount || Number(amount) === 0) return null;

    return (
      <tr className="hover">
        <td>
          <Address address={creditorAddress} />
        </td>
        <td className="text-right">
          <span className="font-bold text-error">{amount?.toString()}
   unidades</span>
```

```
        </td>
      </tr>
    );
  };
}
```

O subcomponente DebtOwedToMe é responsável por identificar e apresentar todas as dívidas que outros utilizadores têm para com o endereço atualmente selecionado. Para cada potencial devedor, o componente consulta o contrato inteligente através da função lookup, verificando se existe um montante positivo associado à relação devedor → utilizador atual. Apenas valores não nulos são exibidos, garantindo uma representação clara e eficiente das dívidas recebidas no sistema.

```
// Componente para d vidas que outros me devem
 const DebtOwedToMe = ({ debtorAddress }: { debtorAddress: string }) =>
   {
   const { data: amount } = useScaffoldReadContract({
     contractName: "Splitwise",
     functionName: "lookup",
     args: [debtorAddress as `0x${string}`, address as `0x${string}`],
   });

   if (!amount || Number(amount) === 0) return null;

   return (
     <tr className="hover">
       <td>
         <Address address={debtorAddress} />
       </td>
       <td className="text-right">
         <span className="font-bold text-success">{amount?.toString()}
  unidades</span>
       </td>
     </tr>
   );
 };
```

3.3.3 Consulta da última atividade

A função getLastActive obtém o timestamp da última interação de um utilizador com o contrato. Converte o valor devolvido para um número JavaScript e devolve null caso o utilizador nunca tenha realizado qualquer operação. Permite ao frontend apresentar informação contextual sobre a atividade dos utilizadores, e suporta funcionalidades de análise temporal ou ordenação por atividade.

```
async function getLastActive(contract: any, user: string) {
  const ts = await contract.read.lastActive(user);
  const timestamp = Number(ts);

  if (timestamp === 0) {
    return null;
  }

  return timestamp;
}
```

3.3.4 Componente addIOUForm

O componente AddIOUForm constitui a interface responsável pela submissão de novas dívidas ao contrato inteligente, permitindo ao utilizador introduzir o credor e o montante pretendido. Para suportar esta funcionalidade, o componente inclui a função auxiliar checkDebtStatus, que consulta diretamente o contrato através da função lookup com o objetivo de verificar, após a operação, se a dívida correspondente foi efetivamente atualizada ou eliminada. Esta verificação é particularmente relevante em cenários onde a adição de uma nova dívida desencadeia a resolução de um ciclo, garantindo assim que a interface reflete corretamente o estado on-chain após a execução da lógica de resolução implementada no contrato.

```
export const AddIOUForm = ({ onIOUAdded }: AddIOUFormProps) => {
  const { address: connectedAddress } = useAccount();
  const [creditorAddress, setCreditorAddress] = useState("");
  const [amount, setAmount] = useState("");
  const [isProcessing, setIsProcessing] = useState(false);

  const { writeContractAsync } = useScaffoldWriteContract("Splitwise");

  const publicClient = usePublicClient();
  const { data: contract } = useScaffoldContract({
    contractName: "Splitwise",
  });

  // Fun   o auxiliar para verificar se a d vida desapareceu
  const checkDebtStatus = async (debtor: string, creditor: string):
   Promise<number> => {
    if (!contract || !publicClient) return 0;
    try {
      const result = await publicClient.readContract({
        address: contract.address,
        abi: contract.abi,
        functionName: "lookup",
        args: [debtor as `0x${string}`, creditor as `0x${string}`],
      });
      return Number(result || 0n);
    } catch {
      return 0;
    }
  };
```

A função handleAddIOU implementa o fluxo completo de submissão de uma nova dívida através da interface, incluindo validações de entrada, envio da transação ao contrato inteligente e verificação posterior do estado on-chain. Após registar a dívida, a função compara o valor esperado com o valor efetivamente armazenado no contrato, permitindo detetar automaticamente se ocorreu a resolução de um ciclo. Esta verificação possibilita ao frontend informar o utilizador sobre reduções ou anulações resultantes da lógica de resolução implementada no contrato, garantindo assim uma experiência coerente com o comportamento on-chain.

```
const handleAddIOU = async (e: React.FormEvent) => {
    e.preventDefault();

    if (!creditorAddress || !amount || !connectedAddress) {
      notification.error("Preencha todos os campos");
      return;
```

```
    }

    const safeCreditor = creditorAddress.toLowerCase();
    const safeMe = connectedAddress.toLowerCase();

    if (safeCreditor === safeMe) {
      notification.error("N  o pode dever a si mesmo!");
      return;
    }

    setIsProcessing(true);

    try {
      const amountValue = parseInt(amount);
      if (amountValue <= 0) {
        notification.error("Valor inv lido");
        setIsProcessing(false);
        return;
      }

      console.log("A iniciar transa  o");

      // Ler d vida antes
      const debtBefore = await checkDebtStatus(safeMe, safeCreditor);

      // Enviar transa  o
      await writeContractAsync({
        functionName: "iou",
        args: [safeMe as `0x${string}`, safeCreditor as `0x${string}`,
amountValue],
      });

      // Aguardar atualiza  o da blockchain
      await new Promise(r => setTimeout(r, 1000));

      // Ler d vida depois
      const debtAfter = await checkDebtStatus(safeMe, safeCreditor);

      // logica
      const expectedDebt = debtBefore + amountValue;

      console.log(`Antes: ${debtBefore} | Enviei: ${amountValue} |
Esperei: ${expectedDebt} | Real: ${debtAfter}`);

      if (debtAfter < expectedDebt) {
        const savedAmount = expectedDebt - debtAfter;
        notification.success(
          <div className="flex flex-col">
            <span className="font-bold text-lg">     CICLO DETETADO!</
span>
            <span>
              {debtAfter === 0
                ? "A d vida foi totalmente anulada!"
                : `A d vida foi reduzida em ${savedAmount}. Restam ${
debtAfter}.`}
            </span>
          </div>,
          { duration: 8000 },
```

```
        );
      } else {
        notification.success("D vida registada com sucesso!");
      }

      setCreditorAddress("");
      setAmount("");

      if (onIOUAdded) setTimeout(onIOUAdded, 1000);
    } catch (error: any) {
      console.error("Erro:", error);
      notification.error("Falha na transa  o");
    } finally {
      setIsProcessing(false);
    }
  };
```

3.3.5 Cálculo do total devido

A função TotalOwedDisplay é um componente de interface responsável por calcular e apresentar, em tempo real, o montante total que um utilizador deve no sistema. Para isso, consulta periodicamente o contrato inteligente, soma todas as dívidas ativas associadas ao endereço fornecido e atualiza a interface de forma robusta e tolerante a falhas.

```
export const TotalOwedDisplay = ({ address }: TotalOwedDisplayProps) =>
   {
  const publicClient = usePublicClient();
  const { data: contract } = useScaffoldContract({
    contractName: "Splitwise",
  });

  const [totalOwed, setTotalOwed] = useState<number | null>(null);
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    // 1. Preven  o de execu  o se dados essenciais faltarem
    if (!publicClient || !contract || !address) return;

    const loadTotalOwed = async () => {
      setIsLoading(true);
      try {
        // Ler todos os utilizadores
        const usersData = await publicClient.readContract({
          address: contract.address,
          abi: contract.abi,
          functionName: "getAllUsers",
        });

        // Garantir que    um array
        const usersArray = (usersData as string[]) || [];

        // Se n o houver users, n o vale a pena continuar
        if (usersArray.length === 0) {
          setTotalOwed(0);
          setIsLoading(false);
          return;
        }
```

```
33
34        let sum = 0;
35
36        // 3. Loop protegido
37        for (const user of usersArray) {
38          // N o verificar d vida comigo mesmo ou endere os
    inv lidos
39          if (!user || user.toLowerCase() === address.toLowerCase())
    continue;
40
41          try {
42            const amount = await publicClient.readContract({
43              address: contract.address,
44              abi: contract.abi,
45              functionName: "lookup",
46              args: [address as `0x${string}`, user as `0x${string}`],
47            });
48
49            // Convers o segura: O contrato devolve uint32 (number ou
    bigint)
50            // Convertemos para Number para somar f cil (uint32 cabe em
     Number JS)
51            const numericAmount = Number(amount || 0);
52            sum += numericAmount;
53          } catch (innerError) {
54            // Se falhar a ler UM user, ignora e continua para o
    pr ximo
55            // Isto evita que o componente todo crashe por causa de um
    erro
56            console.warn(`Falha ao ler d vida com ${user}`, innerError)
    ;
57          }
58        }
59
60        setTotalOwed(sum);
61      } catch (error) {
62        console.error("Erro fatal no TotalOwedDisplay:", error);
63        setTotalOwed(0); // Em caso de erro, assume 0 para n o partir a
     UI
64      } finally {
65        setIsLoading(false);
66      }
67    };
68
69    // Executar a fun   o
70    loadTotalOwed();
71
72    // Polling: Atualizar a cada 5 segundos para manter o valor real
73    const interval = setInterval(loadTotalOwed, 5000);
74    return () => clearInterval(interval);
75  }, [publicClient, contract, address]); // Depend ncias
```

3.3.6 Deteção de Ciclos

A função findCycleAndResolve inicia com um conjunto de verificações estruturais destinadas a garantir que a análise de ciclos é realizada sobre um grafo de dívidas consistente e completo. Numa primeira fase, confirma-se que tanto o devedor como o credor pertencem

à lista global de utilizadores, assegurando que a operação é válida no contexto do sistema. Em seguida, procede-se à leitura exaustiva das dívidas existentes entre todos os pares de utilizadores, construindo implicitamente a matriz de adjacência do grafo. Esta etapa permite identificar todas as arestas ativas e validar que o estado on-chain está acessível e coerente antes de avançar para a fase de deteção de ciclos.

```
// VERIFICA  O 1: A lista tem toda a gente?
  if (!allUsers.includes(startNode) || !allUsers.includes(targetNode)) {
    console.error("Erro, o Credor ou o Devedor n o est o na lista de
    users!");
    console.error(`Falta: ${!allUsers.includes(startNode) ? "Credor" : "
    Eu"}`);
    return { hasCycle: false };
  }

  // A Matriz de D vidas
  console.log("2. A verificar liga  es existentes na Blockchain...");
  let foundEdges = 0;

  // Vamos testar todas as combina  es poss veis para ver o que
    existe
  for (const u of allUsers) {
    for (const v of allUsers) {
      if (u === v) continue;
      // L   a d vida real
      const debt = await getDebt(u, v);
      if (debt > 0) {
        console.log(`Existe: ${u.slice(0, 6)}... -> ${v.slice(0, 6)}...
    = ${debt}`);
        foundEdges++;
      }
    }
  }

  if (foundEdges === 0) {
    console.error("Obten  o de d vidas falhou, o sistema l  0
    d vidas. Problema de leitura/endere os.");
    return { hasCycle: false };
  }
```

Após validar a estrutura do grafo, a função prossegue com a aplicação do algoritmo Breadth-First Search (BFS), utilizado para determinar se existe um caminho dirigido entre o credor e o devedor. A BFS percorre o grafo de forma sistemática, explorando todas as arestas com dívida positiva e construindo caminhos incrementais até encontrar o nó de destino. Caso seja identificado um caminho, este representa um ciclo potencial que será posteriormente analisado para determinar o valor mínimo a subtrair. Se nenhum caminho for encontrado, conclui-se que a nova dívida não origina qualquer ciclo no sistema.

```
 // bfs
  console.log(`A iniciar BFS de ${startNode.slice(0, 6)} para ${
    targetNode.slice(0, 6)}`);

  const queue: string[][] = [[startNode]];
  const visited = new Set<string>();
  visited.add(startNode);

  while (queue.length > 0) {
    const path = queue.shift()!;
```

```
    const currentNode = path[path.length - 1];

    if (currentNode === targetNode) {
      console.log("Ciclo Encontrado, Caminho:", path);
      // Calcular m nimo
      let minAmount = newAmount;
      for (let i = 0; i < path.length - 1; i++) {
        const d = await getDebt(path[i], path[i + 1]);
        if (d < minAmount) minAmount = d;
      }
      return { hasCycle: true, path, minAmount, updates: [] };
    }

    for (const neighbor of allUsers) {
      const neighborLower = neighbor.toLowerCase();
      if (neighborLower !== currentNode && !visited.has(neighborLower))
   {
        const debt = await getDebt(currentNode, neighborLower);
        if (debt > 0) {
          visited.add(neighborLower);
          queue.push([...path, neighborLower]);
        }
      }
    }
  }

  console.log("BFS terminou sem encontrar caminho.");
  return { hasCycle: false };
}
```

Capítulo 4

Testes e Resultados

4.1 Sanity Check

Para validar o sistema e garantir o correto funcionamento do contrato inteligente e da interface, executou-se o cenário de teste proposto no enunciado ("Sanity Check"). Este teste consiste num ciclo simples de três intervenientes:

1. **Passo 1:** Alice adiciona IOU de 10 para Bob.
2. **Passo 2:** Bob adiciona IOU de 10 para Carol.
3. **Passo 3:** Carol tenta adicionar IOU de 10 para Alice.

Nas secções seguintes, detalha-se o processo de execução deste teste, desde a configuração inicial até à resolução automática do ciclo.

4.1.1 Configuração Inicial e Obtenção de Fundos

Para iniciar os testes, é necessário garantir que cada interveniente (Alice, Bob e Carol) possui saldo suficiente para executar transações na *testnet* local. A Figura 4.1 ilustra a interface inicial da aplicação.

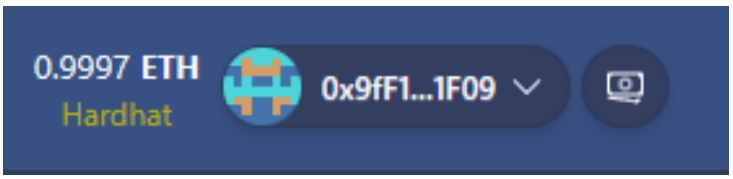

Figura 4.1: Identificação do endereço e carregamento de fundos via Faucet.

Conforme destacado na imagem acima:

- **Identificação:** No canto superior direito encontra-se o endereço público da carteira (*wallet address*) do utilizador autenticado. Este endereço é utilizado como identificador único para o registo de dívidas.
- **Faucet:** O botão de *Faucet* permite ao utilizador solicitar ETH de teste. Esta etapa é crucial, pois todas as operações de escrita no *Smart Contract* (`iou`) requerem o pagamento de taxas de gás (*gas fees*).

4.1.2 Registo de Dívidas (IOU)

A Figura 4.3 demonstra o fluxo de criação de uma dívida no sistema. Neste exemplo, a Alice regista que deve um determinado valor ao Bob.

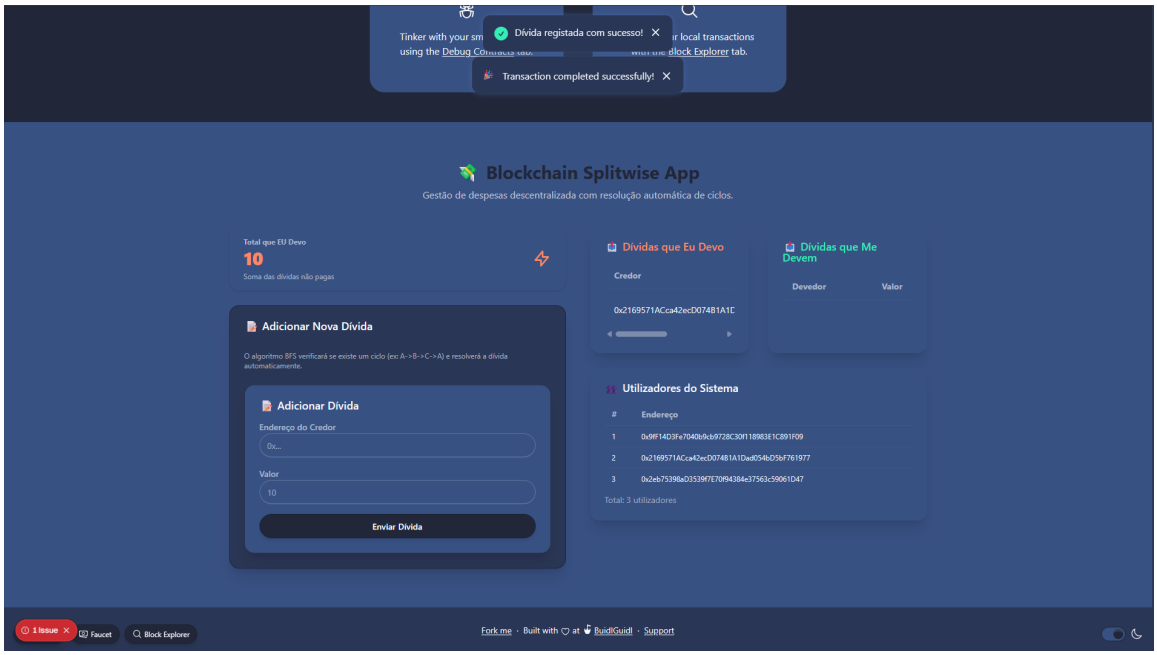

Figura 4.2: Formulário de registo de uma dívida (IOU).

O utilizador preenche o formulário com os seguintes dados:

1. **Endereço do Credor:** O endereço público do utilizador a quem se deve o valor (obtido conforme explicado na secção anterior).
2. **Valor:** A quantia a ser registada (neste caso, 10 unidades).

Ao submeter o formulário, a aplicação invoca a função do contrato inteligente, validando os dados no *frontend* antes do envio da transação para a *blockchain*.

De seguida temos uma dívida também do Bob para a Carol, com o mesmo exato valor.

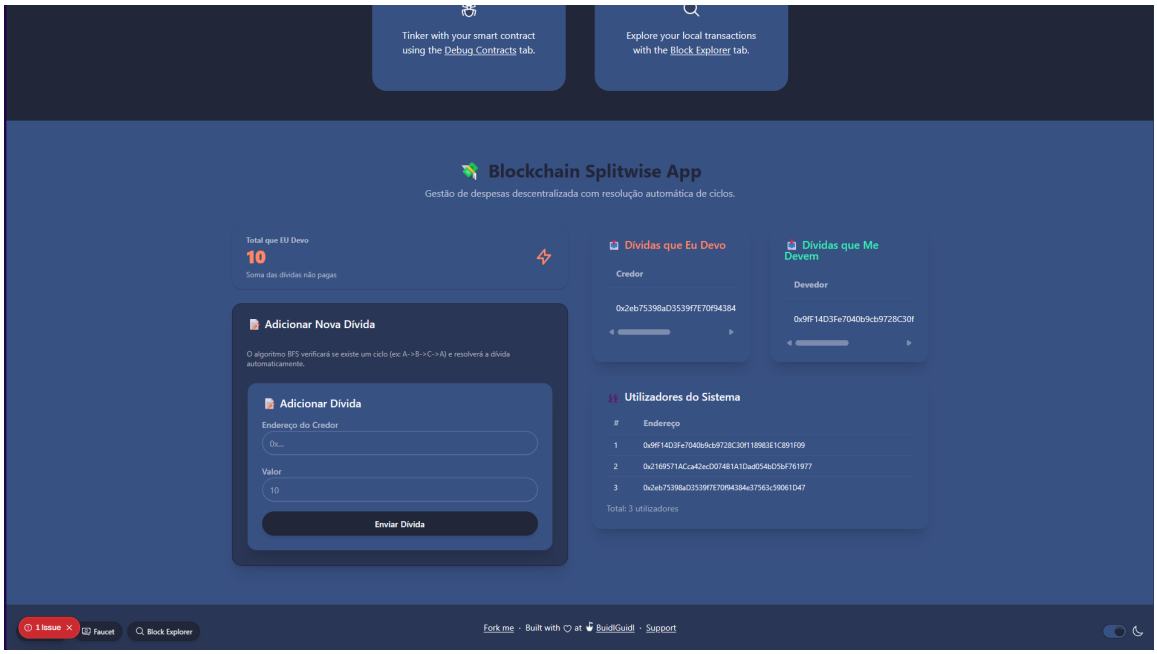

Figura 4.3: Formulário de registo de uma nova dívida (IOU).

4.1.3 Visualização e Resolução de Ciclos

Após a execução sequencial das transações ($A \rightarrow B$ e $B \rightarrow C$), o sistema encontra-se com dívidas ativas. No momento em que a Carol envia a dívida para a Alice ($C \rightarrow A$), o contrato deteta o fechamento do grafo.

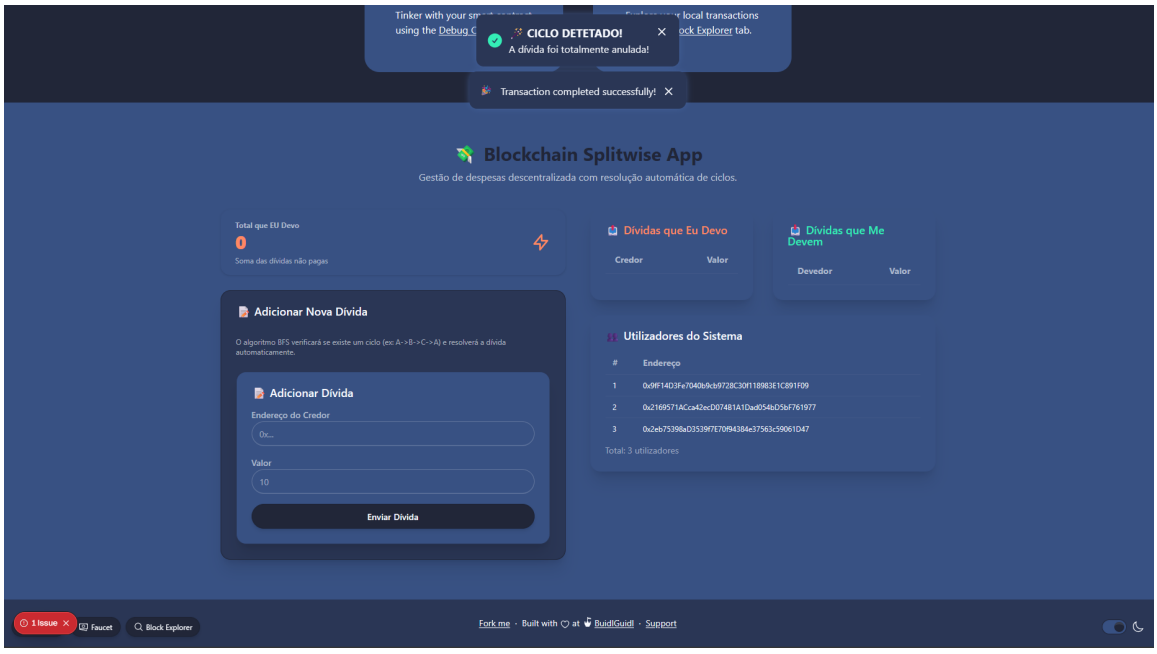

Figura 4.4: Notificação de resolução de ciclo e estado final das tabelas.

A Figura 4.4 evidencia o resultado final:

- **Resolução Automática:** O sistema exibe uma notificação ("Pop-up") confirmando que o ciclo foi detetado e resolvido com sucesso.

- **Estado do Ledger:** As tabelas de dívidas são atualizadas instantaneamente para refletir que o valor mínimo do ciclo (10) foi subtraído a todos os intervenientes, resultando num saldo nulo para todos, conforme esperado no algoritmo de resolução de dívidas descentralizado.

Capítulo 5

Conclusões

O desenvolvimento deste projeto permitiu uma imersão profunda no ecossistema de desenvolvimento *Full Stack* na Ethereum, consolidando conhecimentos teóricos sobre *Blockchain* através da sua aplicação prática num cenário real de partilha de despesas. A integração entre contratos inteligentes (*Smart Contracts*) e interfaces modernas de utilizador revelou-se um desafio técnico complexo, superado com sucesso através da utilização da *stack* tecnológica baseada no **Scaffold-ETH 2**.

A utilização desta ferramenta acelerou significativamente a configuração do ambiente de desenvolvimento (*Hardhat*, *Next.js*, *Wagmi*), permitindo que o foco do trabalho recaísse sobre a lógica de negócio e a resolução algorítmica do problema, em detrimento da configuração de infraestrutura.

5.1 Principais Conquistas

As principais mais-valias técnicas resultantes deste projeto foram:

- **Otimização de *Gas* e Armazenamento:** A escolha deliberada de tipos de dados mais compactos, como `uint32` para os valores das dívidas, demonstrou uma preocupação com a eficiência do contrato, reduzindo o custo computacional e de armazenamento na EVM (*Ethereum Virtual Machine*).

- **Algoritmos Complexos *On-Chain*:** Ao contrário de abordagens convencionais que delegam a lógica pesada para o *frontend*, este projeto implementou a deteção e resolução de ciclos diretamente no *Smart Contract*. Esta abordagem garante atomicidade e confiança total na resolução das dívidas, assegurando que o estado da *ledger* é sempre consistente, independentemente da interface utilizada.

- **Gestão de Estruturas de Dados em Solidity:** Foi desenvolvida uma solução robusta para contornar as limitações nativas da linguagem Solidity, nomeadamente a impossibilidade de iterar diretamente sobre *mappings*. A utilização de estruturas auxiliares (*arrays* de endereços) permitiu a travessia eficiente do grafo de utilizadores para a execução dos algoritmos de procura.

- **Experiência de Utilizador (UX) Reativa:** A interface desenvolvida fornece *feedback* imediato ao utilizador. A implementação de notificações visuais ("Pop-ups") que confirmam a resolução matemática dos ciclos transforma uma operação técnica abstrata numa experiência tangível e compreensível para o utilizador final.

5.2 Desafios e Considerações Finais

Durante o desenvolvimento, foram ultrapassados diversos obstáculos inerentes à programação descentralizada, tais como a gestão de *nonces* nas carteiras de teste, a sincronização assíncrona entre o estado da *blockchain* e o *frontend*, e a necessidade de normalização de endereços (tratamento de *case-sensitivity*).

Em suma, o sistema ***Blockchain Splitwise*** desenvolvido cumpre integralmente os requisitos propostos. O projeto demonstra que é possível criar aplicações descentralizadas (dApps) que não só garantem a integridade e imutabilidade dos registos financeiros, como também oferecem uma usabilidade comparável às aplicações *Web2* tradicionais.