Simplifying Current Decentralized Micropayment Channels

Ruoyu Hou rhou@ucsd.edu
Jing Wen Lin jwl002@ucsd.edu
Stanley Park stpark@ucsd.edu
Zhekai Wang zhw040@ucsd.edu

Sheffield Nolan Sheffield.Nolan@franklintempleton.com

Abstract

Ethereum has allowed people to make payments of Ether on a decentralized network. However, one of the main drawbacks of sending payments through Ethereum are gas fees. As adoption of Ethereum grows, the need to minimize gas fees is important for the user's experience. One solution is to create a payment channel, which will allow a sender to make continuous payments to a receiver within a predefined time period but only requiring a single transaction fee from both parties. One current approach to setting up and using this requires five separate applications: MetaMask, Solidity code, Remix IDE, Sepolia Etherscan, and Javascript. However, this overly complicates the process. In this project, we condense the functionalities of four of the five applications, namely Solidity code, Remix IDE, Sepolia Etherscan, and JavaScript, required to create and use a payment channel into a single decentralized application, which reduces the number of applications required to just two: MetaMask and this decentralized application.

Website: https://stpark01.github.io/about-micropayment-channel-dapp/
Code: https://github.com/Ryhouu/capstone-web-dashboard

1 Introduction

Ethereum has allowed people to make payments of Ether on a decentralized network. However, one of the main drawbacks of sending payments through Ethereum are gas fees. As adoption of Ethereum grows, the need to minimize gas fees is important for the user's experience. One solution is to create a payment channel, which will allow a sender to make continuous payments to a receiver within a predefined time period but only requiring a single transaction fee from both parties: one for opening the payment channel by the sender and another for closing the payment channel by the receiver. One current approach to setting up and using this requires five separate applications: MetaMask, Solidity code, Remix IDE, Sepolia Etherscan, and Javascript. However, this overly complicates the process. In this project, we condense the functionalities of four of the five applications, namely Solidity code, Remix IDE, Sepolia Etherscan, and JavaScript, required to create and use a payment channel into a single decentralized application, which reduces the number of applications required to just two: MetaMask and this decentralized application. In other words, we successfully created a decentralized application that along with MetaMask can create and interact with payment channels on the Sepolia Testnet.

1.1 Literature Review

Bitcoin is the first decentralized cryptocurrency. It allows people to send and receive Bitcoin, but sending Bitcoin costs a transaction fee in Bitcoin to complete each time (Nakamoto 2008). This was implemented in order to keep the Bitcoin network stable. While transaction fees in decentralized cryptocurrencies like Bitcoin and Ethereum are necessary, it is also an annoyance for the user. The idea of payment channels was one potential solution to this problem. This idea can come into fruition through Ethereum smart contracts. A simple example of a payment channel smart contract has already been created by Soliditylang, which allows the creation of a one-way payment channel between a sender and a receiver. However, one current approach to setting up and using this requires five separate applications. The purpose of this project is to try and minimize the complexity of setting up this provided payment channel smart contract in the Sepolia Testnet by condensing four of the five applications into a single application.

1.2 Data Description

1. Sepolia Ether (decimal numbers): It is the native cryptocurrency used in the Sepolia Testnet. Importantly, Sepolia Ether has no real world value, and it is the currency used in the payment channels created in this application.

2. User's Externally Owned Account (42-character hexadecimal): Ethereum account address, which will be used to deploy to the Sepolia Testnet, interact with the Sepolia Testnet, and create/verify payment signatures through this application.

3. Recipient's Externally Owned Account (42-character hexadecimal): Ethereum account

address the sender wants to create a payment channel with.

3. Expiration (int): The number of time periods until a payment channel expires.

4. New Expiration (int): Unix timestamp the sender wants to extend a payment channel to (must be greater than the current expiration's Unix timestamp).

4. Contact Address: Payment channel's contract address in the Sepolia Testnet.

5. Signature (hexadecimal): Payment signature created by a sender for a specific payment channel that they own.

2 Methods

2.1 Sepolia Testnet

The Sepolia Testnet is similar to the Ethereum network, except it uses testnet Ether. Sepolia Ether has no real world value unlike Ether.

2.2 Micropayment Channel

The code for the micropayment channel was created by Soliditylang (et al. 2019).

2.3 Frontend

2.3.1 Overview

Our framework choices for the web dashboard are **Next.js** with **TypeScript** and **React**. This combination is not only a technical decision but a strategic one - aimed at enhancing performance, scalability, and developer efficiency.

First, React is a popular JavaScript library for building user interfaces. Its component-based architecture allows developers to create reusable UI elements, streamlining the development process and ensuring consistency across the web dashboard.

1. React's efficient updating mechanism, utilizing a virtual DOM, ensures that the user interface remains responsive and dynamic. This is particularly important for our dApp, where users expect real-time updates and interactive elements.
2. By encapsulating UI logic into components, React makes the codebase more manageable and modular. As developers, we can reuse components across different parts of the application in order to reduce development time and ensure a cohesive user experience.

Next.js is a React framework that brings plenty of benefits to web development, particularly in areas where performance and search engine optimization (SEO). The use of React within

Next.js enables the creation of a component-based architecture, making the user interface modular and easy to manage. React's efficient update and rendering system ensures a seamless user experience, essential for browsing and interacting with our dashboard and smart contracts.

1. Server-Side Rendering (SSR): Next.js allows for server-side rendering of web pages, which means that the server pre-renders each page into HTML before sending it to the client's browser. This significantly improves the loading time of the web dashboard. Particularly, Next.js supports Automatic Code Splitting, which splits the code into small bundles, loading only the necessary code for the page the user is visiting. This reduces the amount of data transferred over the network, leading to faster page loads and a smoother browsing experience. Moreover, SSR is beneficial for SEO, as it ensures that search engines can crawl and index the platform's content more efficiently, making it easier for potential users to discover SecureArt online.
2. Pathway to Progressive Web App (PWA): Next.js facilitates the development of Progressive Web Apps, enabling SecureArt to offer offline functionality and app-like experiences. PWAs are crucial for engaging users in areas with unreliable internet connections, ensuring that the platform remains accessible and functional.

And TypeScript is a superset of JavaScript. Integrating TypeScript with Next.js and React brings strong typing to the project, improving maintainability and developer productivity. This combination ensures robust code quality, reducing bugs and facilitating scalability.

2.3.2 Dependencies

The following dependencies are crucial for the frontend development, styling, interaction with Ethereum smart contracts, and compilation of Solidity code. First, we need to install Node.js and npm (Node Package Manager), and then these commands should be executed in the terminal or command prompt.

1. Next.js, React, and React-DOM:

   ```
   npm install next@latest react@latest react-dom@latest
   ```

2. UI Components and Styling: we use the MUI ecosystem. `@mui/material` offers pre-designed material UI components, while `@emotion/react` and `@emotion/styled` are used for styling these components. Also, `styled-components` allows us to write actual CSS code to style our components; `notistack` enhances snackbars provided by Material-UI, allowing for easier customization and management.

   ```
   npm install @mui/material @emotion/react @emotion/styled
   npm install @mui/icons-material
   npm install styled-components
   npm install @mui/x-date-pickers
   npm install @mui/x-date-pickers-pro
   npm install notistack
   ```

3. Smart Contract Interaction: `ethereumjs-abi` is used to encode and decode data according to the smart contract ABI (Application Binary Interface). `ethereumjs-util` is a utility library containing common functions for Ethereum, such as account generation, hashing functions, and signature validation. `ethers` is a library that enables developers to send transactions, interact with smart contracts, and handle wallets. They are crucial for interacting with smart contracts and handling Ethereum-specific operations in the project.

   ```
   npm install ethereumjs-abi
   npm i --save-dev @types/ethereumjs-abi
   npm install ethereumjs-util@latest
   npm install ethers
   ```

4. Solidity Compiler: `solc` is used for compiling Solidity source code (`.sol` files) into bytecode.

   ```
   npm install solc
   ```

5. Data Management: We use Supabase, which is based on PostgreSQL for our data management. `supabase-js` is the official JavaScript client that allows us to interact with Supabase, an open-source Firebase alternative, providing functionalities like authentication, database interactions, and real-time subscriptions.

   ```
   npm install @supabase/supabase-js
   ```

6. Other dependencies:

   ```
   npm install sonner
   npm install dayjs
   npm install fs
   ```

2.3.3 Run the Development Server

To run the development server locally:

```
npm run dev
# or
yarn dev
# or
pnpm dev
# or
bun dev
```

Then open `http://localhost:3005` with your browser to see the result.

2.3.4 Pages

Homepage Figure 1 shows the breakdowns of all the foundational services for both senders and receivers.

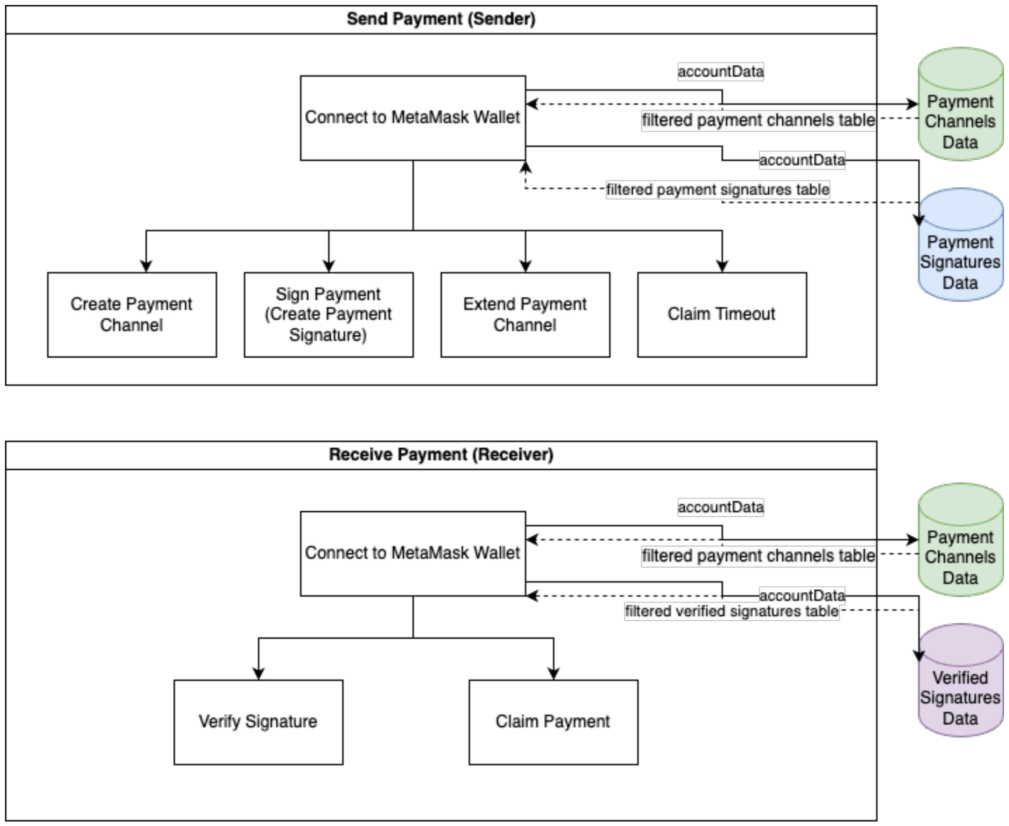

Send Payment (Sender)
Connect to MetaMask Wallet
accountData
filtered payment channels table
accountData
filtered payment signatures table
Payment Channels Data
Payment Signatures Data
Create Payment Channel
Sign Payment (Create Payment Signature)
Extend Payment Channel
Claim Timeout
Receive Payment (Receiver)
Connect to MetaMask Wallet
accountData
filtered payment channels table
accountData
filtered verified signatures table
Payment Channels Data
Verified Signatures Data
Verify Signature
Claim Payment

Figure 1: Services Workflow

Figure 2 shows the current layout of our homepage. It has a navigator bar with four buttons:

- Send (sender): Send SepoliaETH to another wallet account.
- Claim (receiver): Claim SepoliaETH sent from another account to you.
- Split a Bill (multiple senders): Mutually send SepoliaETH to another wallet account.
- Transactions: View your recent transactions.

When a user clicks **Send**, it will direct them to another page with buttons that continue sending them to different pages for each of the functionalities:

- Deploy Payment Channel
- Sign Payment
- Extend Expiration
- Claim Timeout

and when a user clicks **Receive (Claim)**, it will direct them to a page with buttons for

- Verify Payment
- Claim Payment

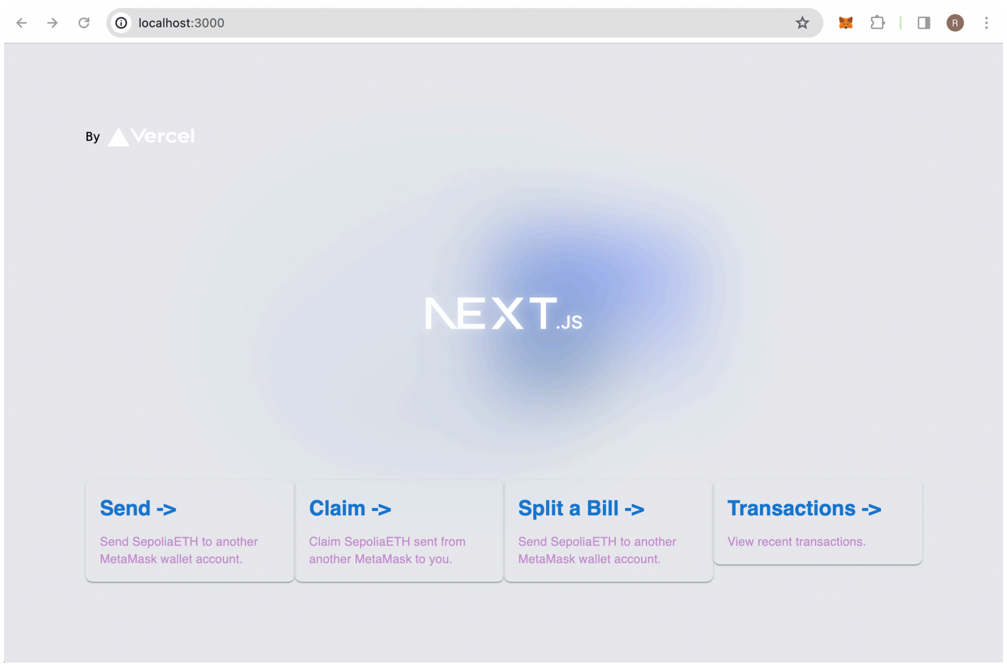

localhost:3000
By Vercel
NEXT.JS
Send ->
Send SepoliaETH to another MetaMask wallet account.
Claim ->
Claim SepoliaETH sent from another MetaMask to you.
Split a Bill ->
Send SepoliaETH to another MetaMask wallet account.
Transactions ->
View recent transactions.

Figure 2: Homepage

Account Menu Note that before the user proceeds to any of the above services, they should connect their MetaMask Wallet to the website first. To help with the log-in check, our dashboard will pop up an alert to inform them to connect their MetaMask wallet if they haven't when they attempt to start a function.

Figure 3 shows the panel for the Account Menu (the left side is a sender account and the right side is a receiver account). The menu items seem alike, but specifically, the "My Payment Channels" and "My Signatures" tabs on senders' side will toggle on the **Payment Channels** table and **Payment Signatures** table, whereas the tabs on receivers' side will toggle on the **Payment Channels** table and **Verified Signatures** table (both only show the data related to the connected MetaMask account).

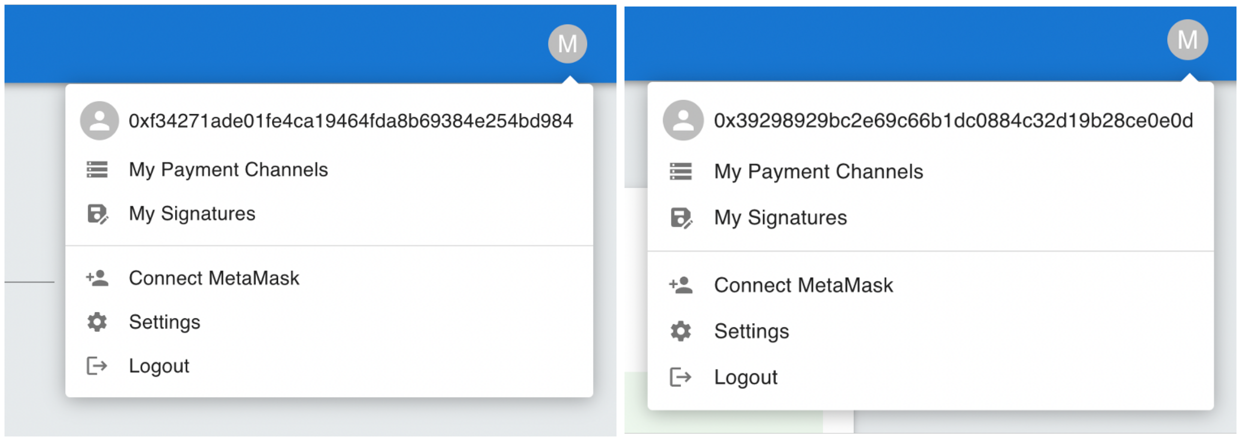


Figure 3: Account Menu - Sender (left) & Receiver (right)

The Account object data schema is below:

```
export interface AccountDataSchema {
    id: string,
    account: string,
    isSender: boolean,
    isReceiver: boolean
}
```

Send Payment Page As described before, the Send Payment Page has four actions:

- Create Payment Channel
- Sign Payment (Create Payment Signature)
- Extend Channel
- Claim Timeout

Each action is independent and holds temporary data in the cache respectively.

For the contract/signature data forms on the dashboard, we use the schema as follow:

```
    export interface PaymentChannelDataSchema {
    id: string,
    senderAddress: string,
    recipientAddress: string,
    escrowAmount: number, // eth
    escrowAmountWei: number,
    contractAddress: string,
    signature: string,
    message: string,
    isVerified: boolean,
    isDeployed: boolean
    claimAmount: number,
    claimAmountWei: number,
    closed: boolean,
    expirationDate: Date,
    transactionHash: string,
    balance: number // eth
}
```

1. Create Payment Channel. Figure 4 shows the sequence diagram.

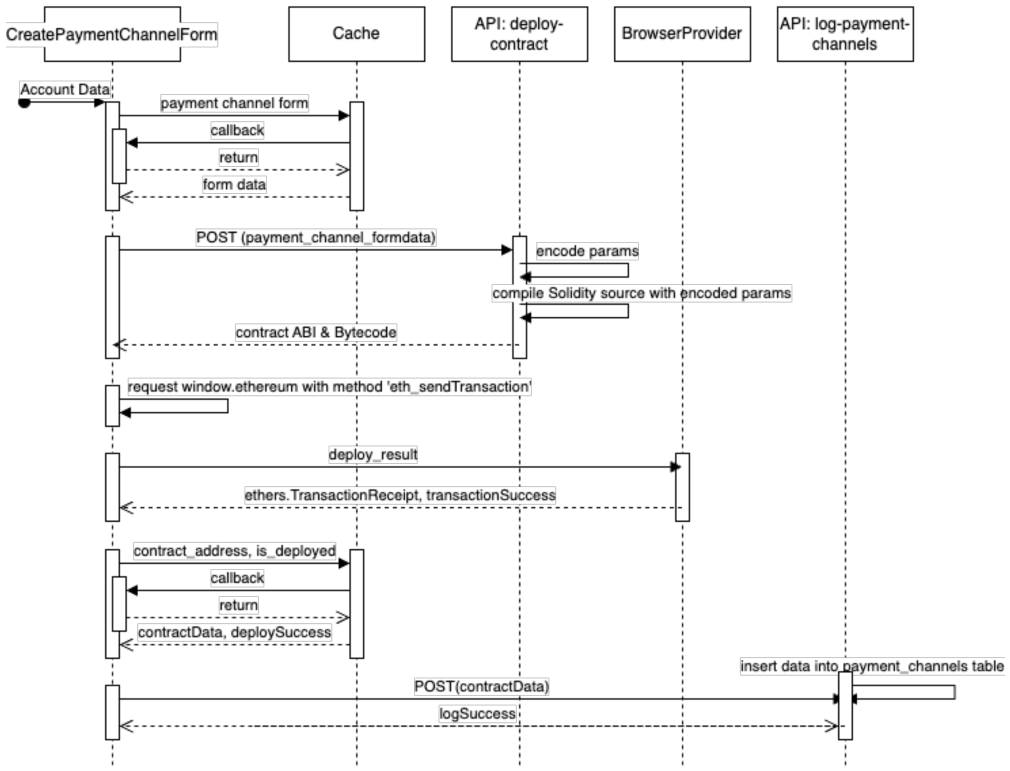
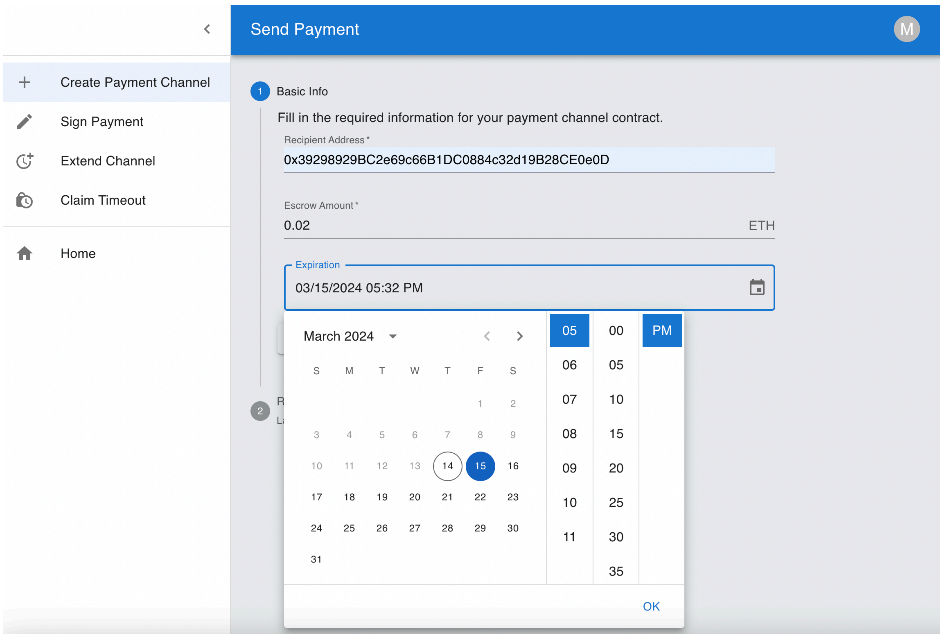

CreatePaymentChannelForm
Cache
API: deploy-contract
BrowserProvider
API: log-payment-channels
Account Data
payment channel form
callback
return
form data
POST (payment_channel_formdata)
encode params
compile Solidity source with encoded params
contract ABI & Bytecode
request window.ethereum with method 'eth_sendTransaction'
deploy_result
ethers.TransactionReceipt, transactionSuccess
contract_address, is_deployed
callback
return
contractData, deploySuccess
insert data into payment_channels table
POST(contractData)
logSuccess

Figure 4: Create Payment Channel, Sequence Diagram

Figure 5 shows the filled form in the first step. Here, `recipientAddress`, `escrowAmount` and `escrowAmountWei`, `expirationDate` are changed accordingly. Particularly, we use Material-UI's `LocalizationProvider` and `DateTimePicker` components to create a date and time picker for selecting an expiration date and time, where the selected value **cannot be in the past**, and communicates the selected value to the parent component or handles it as needed.
Then, Figure 6 shows the review page for the user to confirm the information they filled in is correct;
Figures 7 and 8 exemplify the progress: when the user clicks ”deploy”, it will trigger MetaMask to let the user confirm the transaction, and once the contract is deployed on Sepolia, our website will show the contract address with a ”copy” button, and allow the user to view the transaction and contract on the Sepolia Explorer externally for their convenience.
Then, Figure 9 shows that the contract just deployed by the user is logged in our database and can be viewed as in the table ”Account Menu - My Payment Channels”.



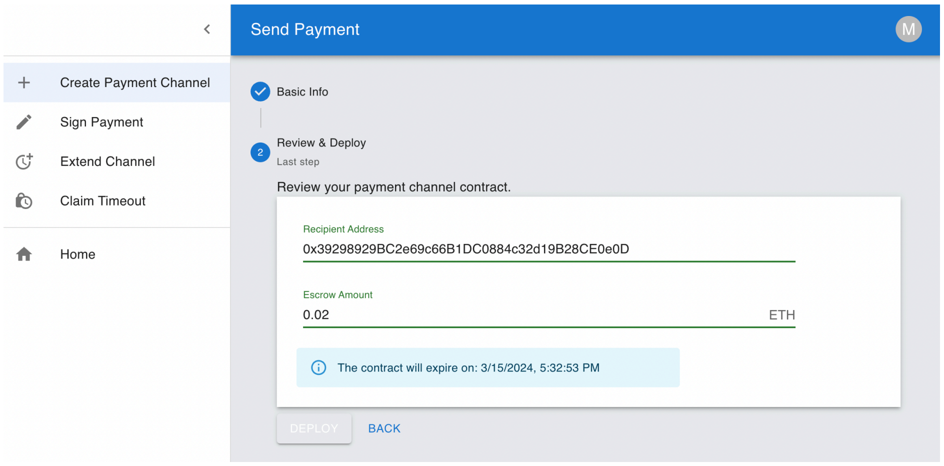
Fill in the required information for your payment channel contract.

Recipient Address *
0x39298929BC2e69c66B1DC0884c32d19B28CE0e0D

Escrow Amount *
0.02 ETH

Expiration
03/15/2024 05:32 PM

Figure 5: Create Payment Channel, Basic Info Form



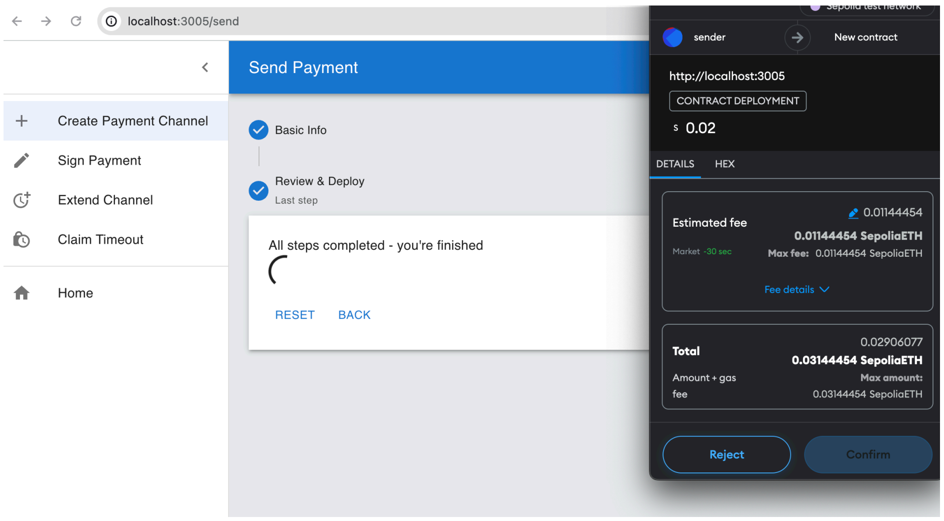
Review your payment channel contract.

Recipient Address
0x39298929BC2e69c66B1DC0884c32d19B28CE0e0D

Escrow Amount
0.02 ETH

The contract will expire on: 3/15/2024, 5:32:53 PM

Figure 6: Create Payment Channel, Review Form



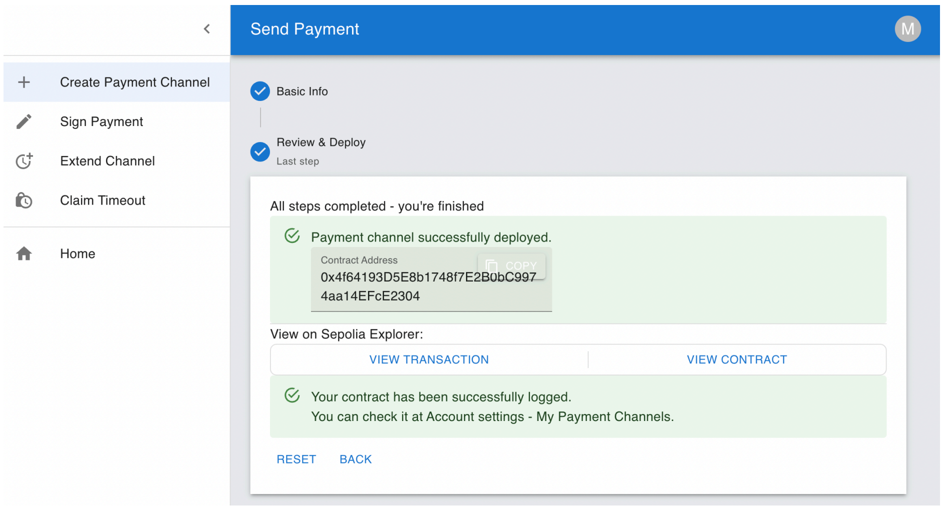
All steps completed - you're finished

http://localhost:3005
CONTRACT DEPLOYMENT
$ 0.02

Estimated fee 0.01144454
0.01144454 SepoliaETH
Market ~30 sec
Max fee: 0.01144454 SepoliaETH

Total 0.02906077
0.03144454 SepoliaETH
Amount + gas fee
Max amount: 0.03144454 SepoliaETH

Figure 7: Create Payment Channel, Deploy



All steps completed - you're finished

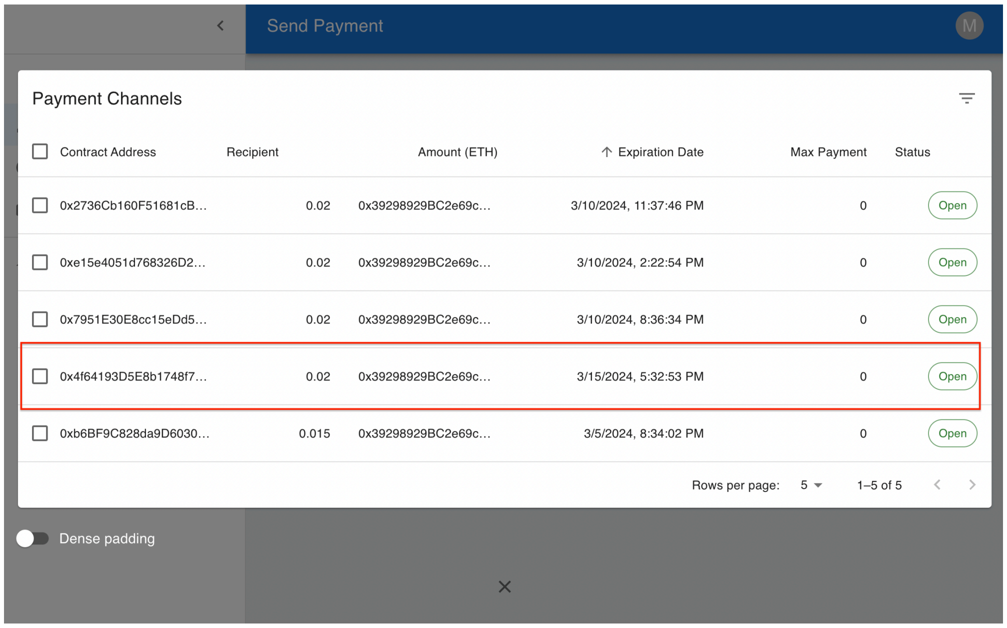
Payment channel successfully deployed.

Contract Address
0x4f64193D5E8b1748f7E2B0bC997 4aa14EFcE2304

View on Sepolia Explorer:

Your contract has been successfully logged.
You can check it at Account settings - My Payment Channels.

Figure 8: Create Payment Channel, Deploy Result



Payment Channels

Contract Address	Recipient	Amount (ETH)	Expiration Date	Max Payment	Status
0x2736Cb160F51681cB…	0.02	0x39298929BC2e69c…	3/10/2024, 11:37:46 PM	0	Open
0xe15e4051d768326D2…	0.02	0x39298929BC2e69c…	3/10/2024, 2:22:54 PM	0	Open
0x7951E30E8cc15eDd5…	0.02	0x39298929BC2e69c…	3/10/2024, 8:36:34 PM	0	Open
0x4f64193D5E8b1748f7…	0.02	0x39298929BC2e69c…	3/15/2024, 5:32:53 PM	0	Open
0xb6BF9C828da9D6030…	0.015	0x39298929BC2e69c…	3/5/2024, 8:34:02 PM	0	Open

Rows per page: 5 1–5 of 5

Dense padding

Figure 9: Create Payment Channel, Log

2. Create Payment Signature. Figure 10 shows the sequence diagram.

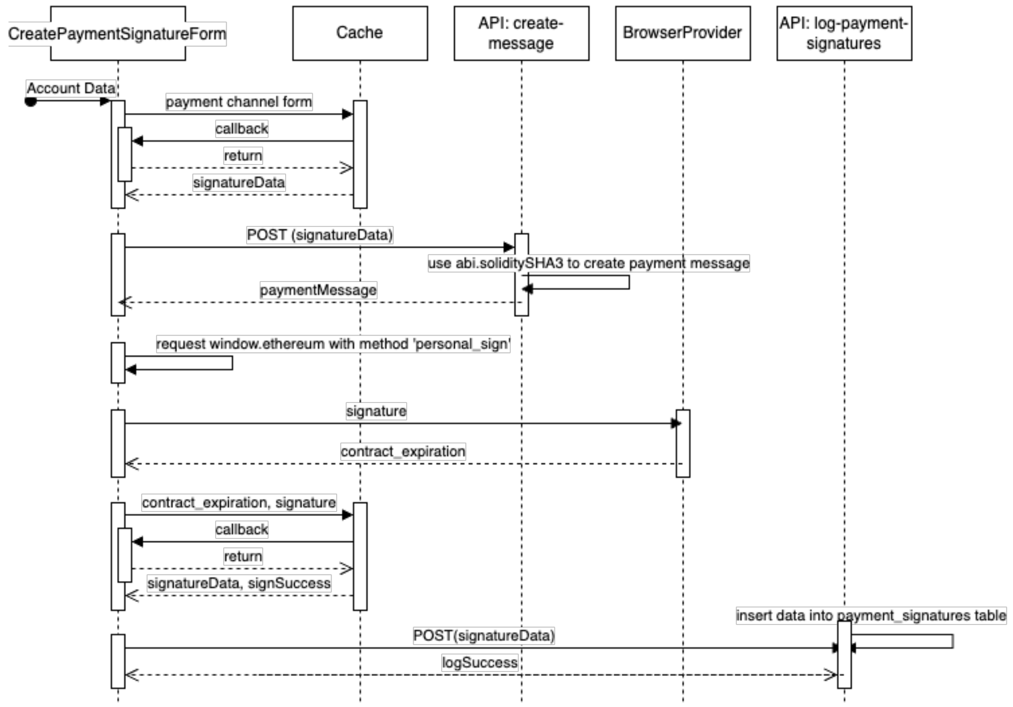
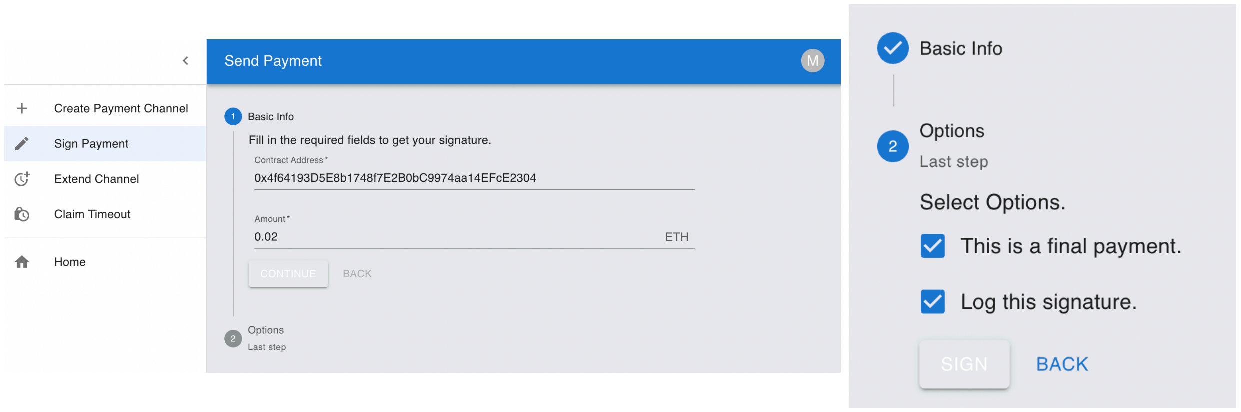

CreatePaymentSignatureForm
Cache
API: create-message
BrowserProvider
API: log-payment-signatures
Account Data
payment channel form
callback
return
signatureData
POST (signatureData)
use abi.soliditySHA3 to create payment message
paymentMessage
request window.ethereum with method 'personal_sign'
signature
contract_expiration
contract_expiration, signature
callback
return
signatureData, signSuccess
insert data into payment_signatures table
POST(signatureData)
logSuccess

Figure 10: Create Payment Signature, Sequence Diagram

Figure 11 shows the filled form with `Partial<PaymentChannelDataSchema>` and the options checkbox. The user can copy the contract address easily either from their previous step ”Create Payment Channel”, or from ”Account Menu - My Payment Channels”.

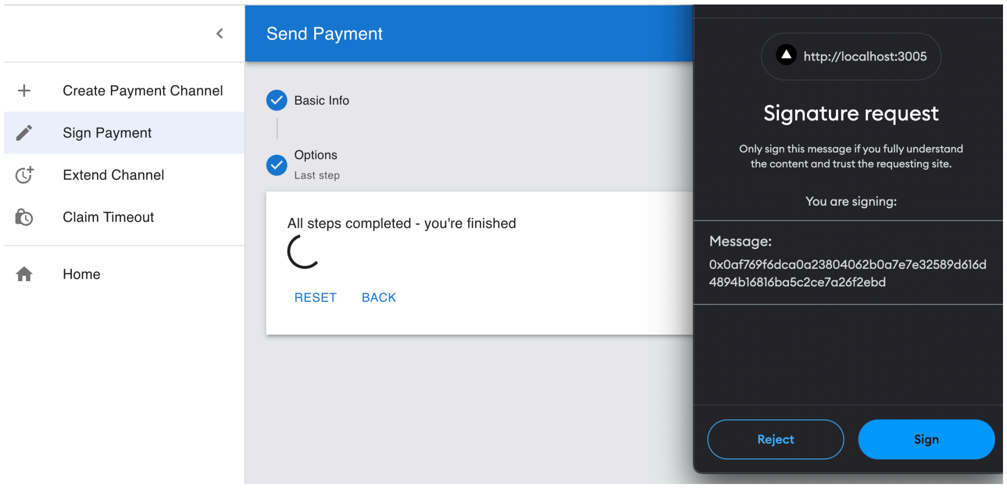


Fill in the required fields to get your signature.
Contract Address *
0x4f64193D5E8b1748f7E2B0bC9974aa14EFcE2304
Amount *
0.02 ETH

Select Options.
☑ This is a final payment.
☑ Log this signature.

Figure 11: Create Payment Signature, Basic Info and Options

Then, Figures 12 and 13 exemplify the transaction of signing the payment and the result panel, where the user can copy the message and signature easily.
Figure 14 shows that the signature just created by the user is logged in our database and can be viewed as in the table ”Account Menu - My Signatures” and these addresses can be accessed and copied for later uses.



All steps completed - you're finished

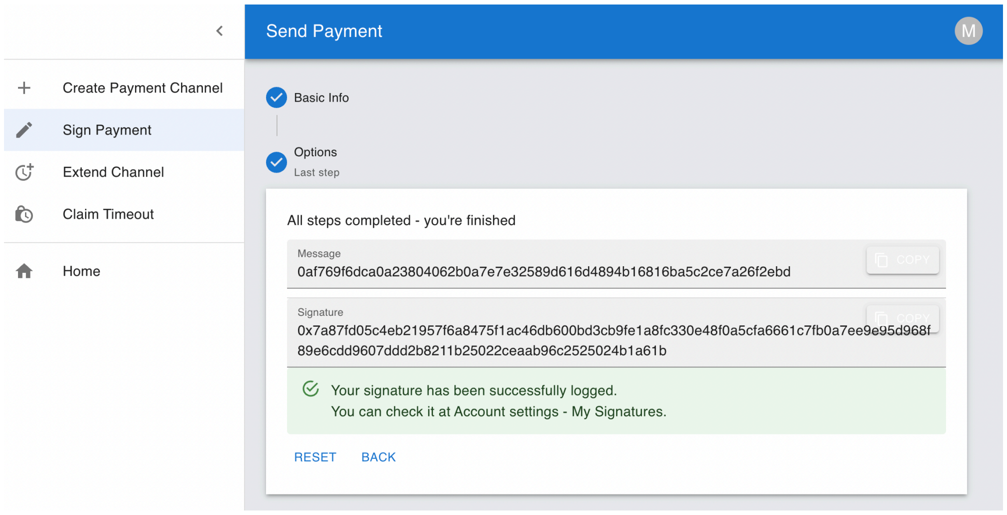
Signature request

Only sign this message if you fully understand the content and trust the requesting site.

You are signing:

Message:
0x0af769f6dca0a23804062b0a7e7e32589d616d4894b16816ba5c2ce7a26f2ebd

Figure 12: Create Payment Signature, Transaction



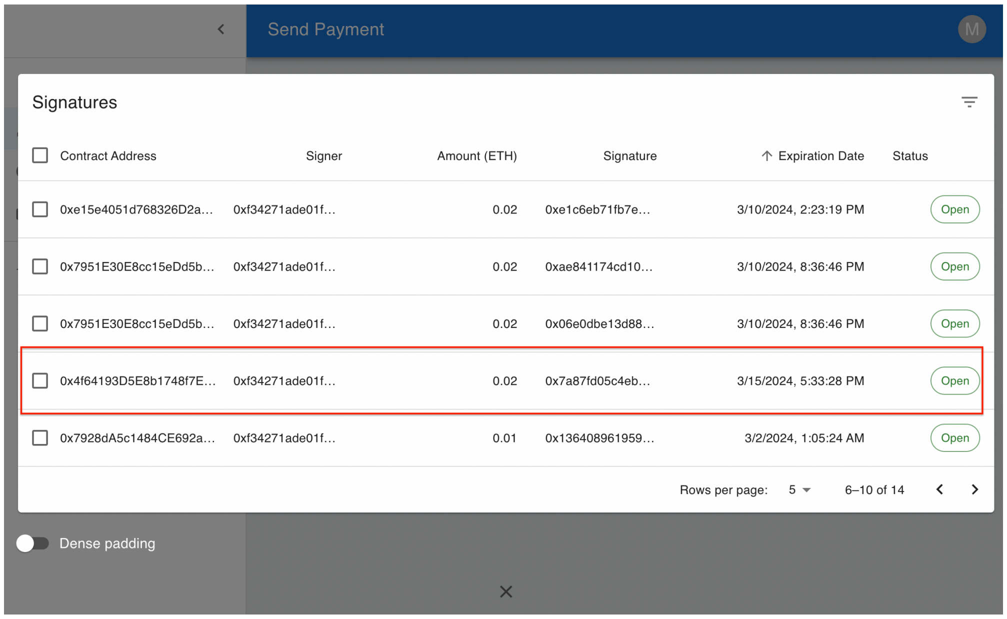
All steps completed - you're finished

Message
0af769f6dca0a23804062b0a7e7e32589d616d4894b16816ba5c2ce7a26f2ebd

Signature
0x7a87fd05c4eb21957f6a8475f1ac46db600bd3cb9fe1a8fc330e48f0a5cfa6661c7fb0a7ee9e95d968f89e6cdd9607ddd2b8211b25022ceaab96c2525024b1a61b

Your signature has been successfully logged.
You can check it at Account settings - My Signatures.

Figure 13: Create Payment Signature, Result



Send Payment

Signatures

Contract Address	Signer	Amount (ETH)	Signature	Expiration Date	Status
0xe15e4051d768326D2a…	0xf34271ade01f…	0.02	0xe1c6eb71fb7e…	3/10/2024, 2:23:19 PM	Open
0x7951E30E8cc15eDd5b…	0xf34271ade01f…	0.02	0xae841174cd10…	3/10/2024, 8:36:46 PM	Open
0x7951E30E8cc15eDd5b…	0xf34271ade01f…	0.02	0x06e0dbe13d88…	3/10/2024, 8:36:46 PM	Open
0x4f64193D5E8b1748f7E…	0xf34271ade01f…	0.02	0x7a87fd05c4eb…	3/15/2024, 5:33:28 PM	Open
0x7928dA5c1484CE692a…	0xf34271ade01f…	0.01	0x136408961959…	3/2/2024, 1:05:24 AM	Open

Rows per page: 5 6–10 of 14

Dense padding

Figure 14: Create Payment Signature, Log

3. Extend Channel. Figure 15 shows the sequence diagram.

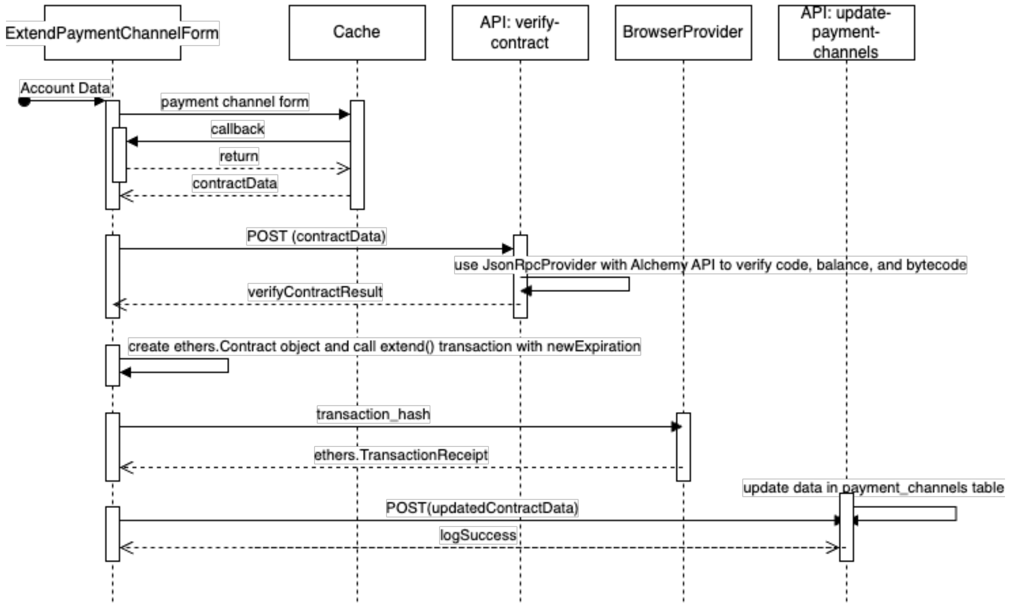
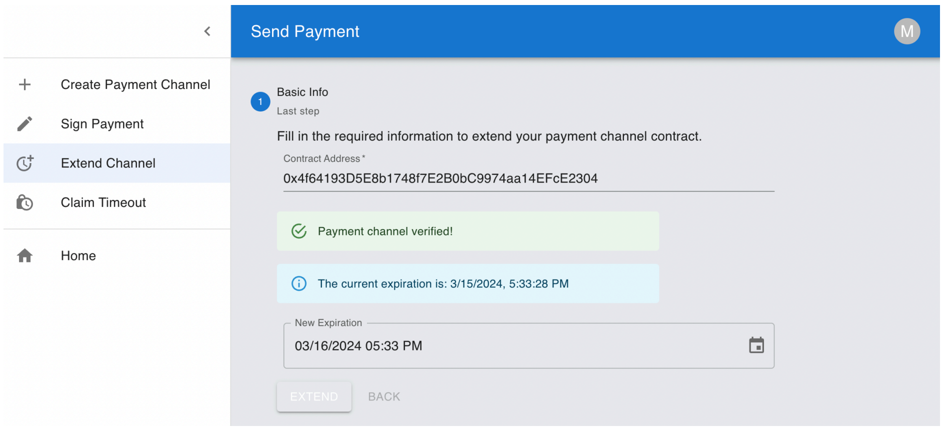

ExtendPaymentChannelForm
Cache
API: verify-contract
BrowserProvider
API: update-payment-channels
Account Data
payment channel form
callback
return
contractData
POST (contractData)
use JsonRpcProvider with Alchemy API to verify code, balance, and bytecode
verifyContractResult
create ethers.Contract object and call extend() transaction with newExpiration
transaction_hash
ethers.TransactionReceipt
update data in payment_channels table
POST(updatedContractData)
logSuccess

Figure 15: Extend Payment Channel, Sequence Diagram

Figure 16 shows that once the user has filled in the contract address to extend, our dashboard would automatically verify the contract address exists and is valid, and use `ethers` to help get the contract expiration and inform the user with an info alert. Figures 17 and 18 show the transaction and its result.



Fill in the required information to extend your payment channel contract.

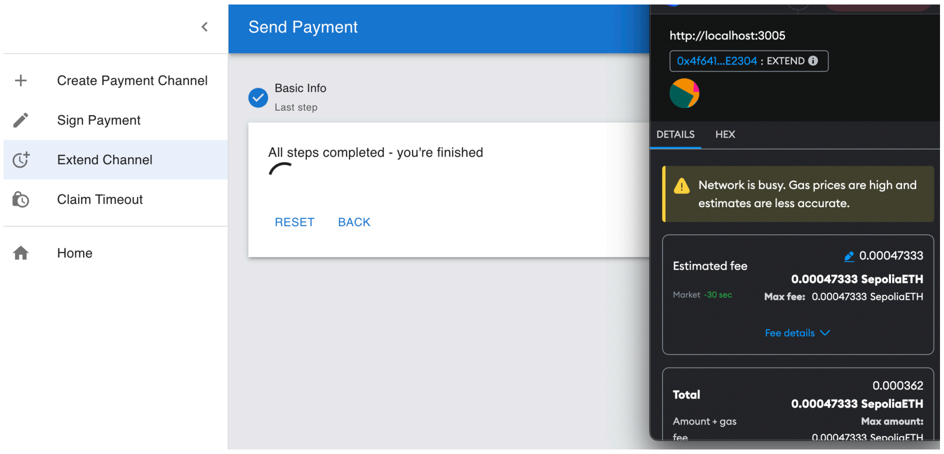
Contract Address *
0x4f64193D5E8b1748f7E2B0bC9974aa14EFcE2304

Payment channel verified!

The current expiration is: 3/15/2024, 5:33:28 PM

New Expiration
03/16/2024 05:33 PM

Figure 16: Extend Channel, Basic Info



All steps completed - you're finished

http://localhost:3005

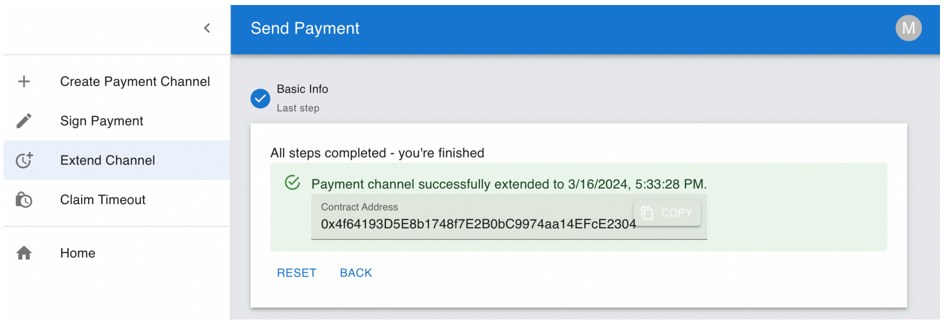
0x4f641...E2304 : EXTEND

Network is busy. Gas prices are high and estimates are less accurate.

Estimated fee 0.00047333
0.00047333 SepoliaETH
Market ~30 sec
Max fee: 0.00047333 SepoliaETH

Total 0.000362
0.00047333 SepoliaETH
Amount + gas fee
Max amount: 0.00047333 SepoliaETH

Figure 17: Extend Channel, Transaction



All steps completed - you're finished

Payment channel successfully extended to 3/16/2024, 5:33:28 PM.

Contract Address
0x4f64193D5E8b1748f7E2B0bC9974aa14EFcE2304

Figure 18: Extend Channel, Result

4. Claim Timeout. Figure 19 shows the sequence diagram.

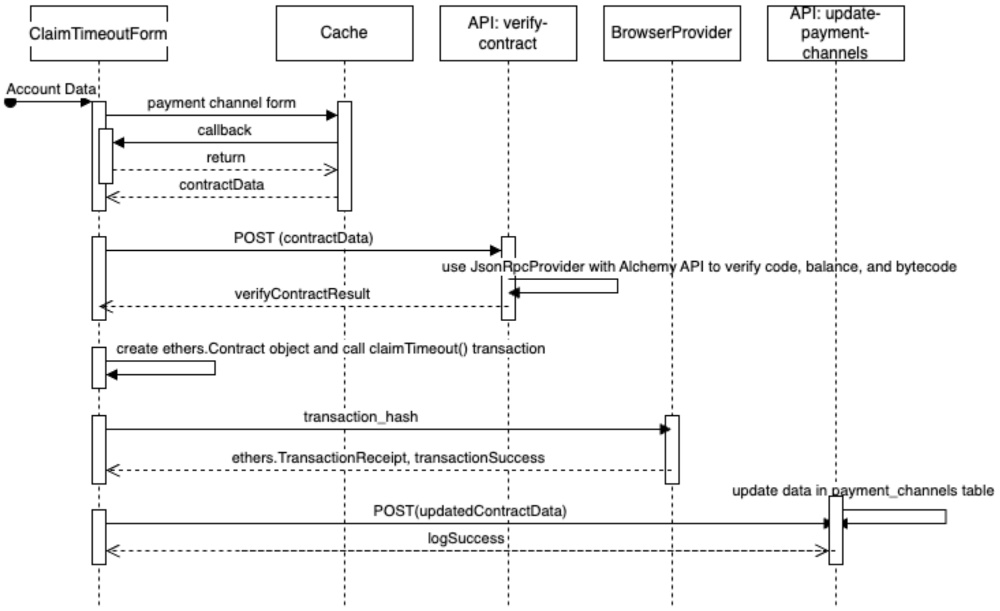
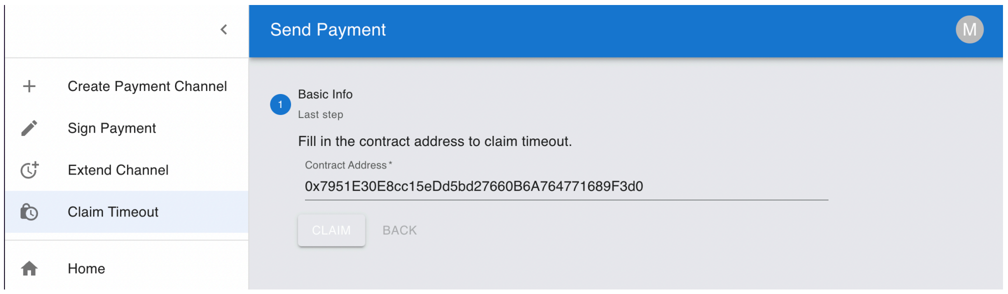

ClaimTimeoutForm
Cache
API: verify-contract
BrowserProvider
API: update-payment-channels
Account Data
payment channel form
callback
return
contractData
POST (contractData)
use JsonRpcProvider with Alchemy API to verify code, balance, and bytecode
verifyContractResult
create ethers.Contract object and call claimTimeout() transaction
transaction_hash
ethers.TransactionReceipt, transactionSuccess
update data in payment_channels table
POST(updatedContractData)
logSuccess

Figure 19: Claim Timeout, Sequence Diagram

Figures 20, 21, and 22 exemplify the user claiming an expired contract and the escrowed amount would be returned to his account.

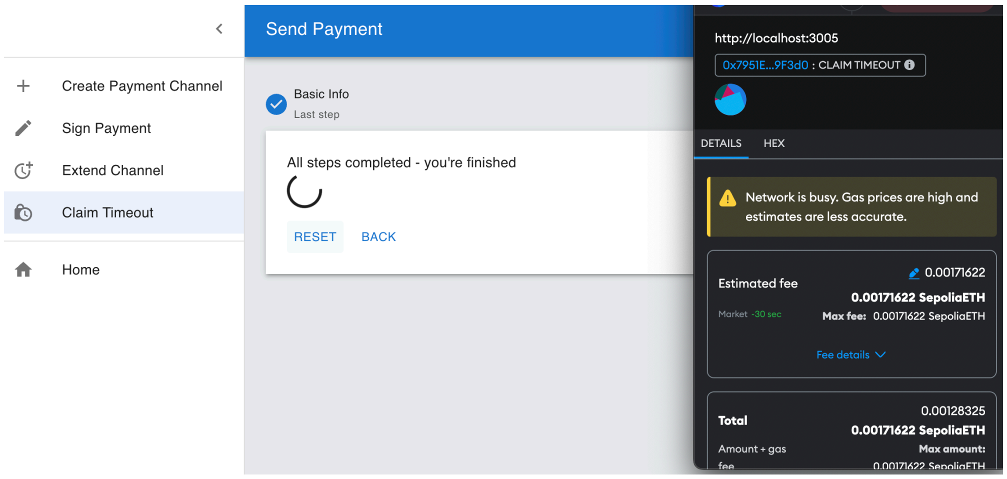


Fill in the contract address to claim timeout.

Contract Address *

0x7951E30E8cc15eDd5bd27660B6A764771689F3d0

Figure 20: Claim Timeout, Basic Info



All steps completed - you're finished

http://localhost:3005

0x7951E...9F3d0 : CLAIM TIMEOUT

Network is busy. Gas prices are high and estimates are less accurate.

Estimated fee 0.00171622

0.00171622 SepoliaETH

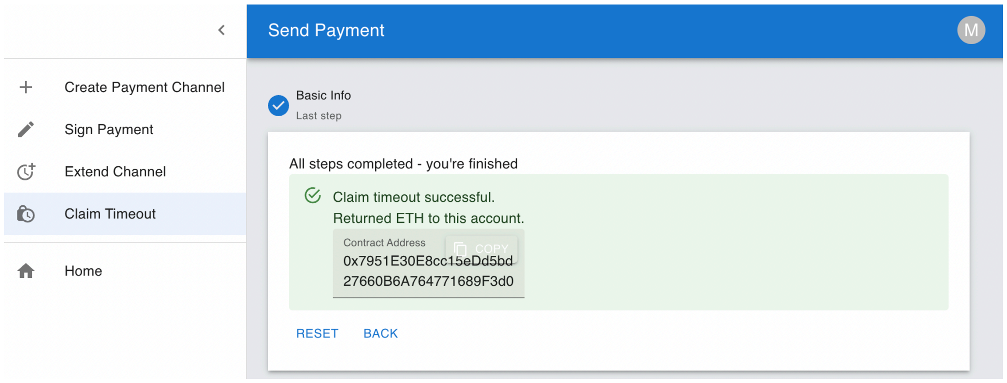
Market ~30 sec Max fee: 0.00171622 SepoliaETH

Total 0.00128325

0.00171622 SepoliaETH

Amount + gas fee Max amount: 0.00171622 SepoliaETH

Figure 21: Claim Timeout, Transaction



All steps completed - you're finished

Claim timeout successful.
Returned ETH to this account.

Contract Address
0x7951E30E8cc15eDd5bd27660B6A764771689F3d0

Figure 22: Claim Timeout, Result

Receive Payment Page As described before, the Receive Payment Page has two foundational actions:

- Verify Payment Signature
- Claim Payment (Close Payment Channel)

Similar to the Send Payment Page, each action is independent and holds temporary data in the cache respectively.

Once the sender has sent the signature and contract address to the receiver through offline methods like email, the receiver can verify if the signature is valid and effective, and claim the payment (close the payment channel) after it is verified.

1. Verify Signature. Figure 23 shows the sequence diagram.

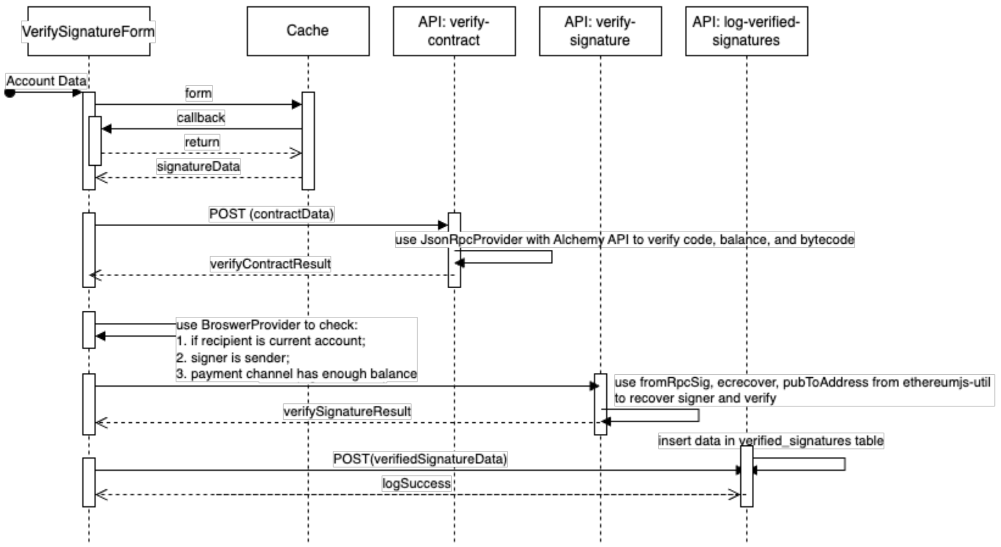
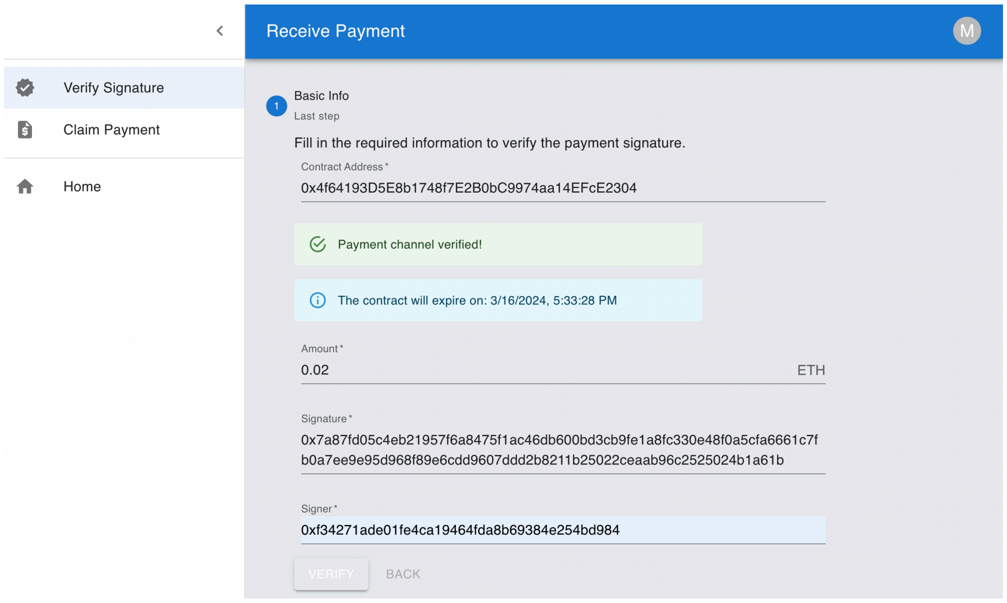

VerifySignatureForm
Cache
API: verify-contract
API: verify-signature
API: log-verified-signatures
Account Data
form
callback
return
signatureData
POST (contractData)
use JsonRpcProvider with Alchemy API to verify code, balance, and bytecode
verifyContractResult
use BroswerProvider to check:
1. if recipient is current account;
2. signer is sender;
3. payment channel has enough balance
use fromRpcSig, ecrecover, pubToAddress from ethereumjs-util
to recover signer and verify
verifySignatureResult
insert data in verified_signatures table
POST(verifiedSignatureData)
logSuccess

Figure 23: Verify Signature, Sequence Diagram

Figures 24 and 25 exemplify the receiver verifying a valid signature.
Figure 26 shows the receiver can access to the verified signatures from ”Account Menu - My Signatures”.



Fill in the required information to verify the payment signature.

Contract Address *
0x4f64193D5E8b1748f7E2B0bC9974aa14EFcE2304

Payment channel verified!

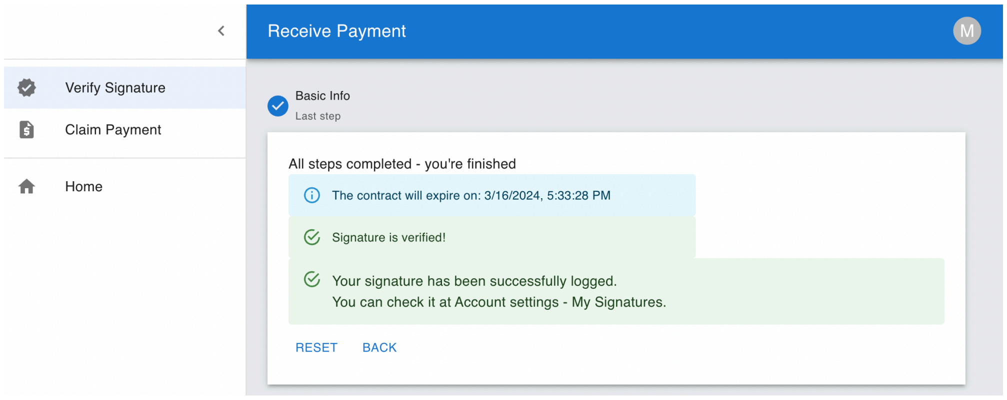
The contract will expire on: 3/16/2024, 5:33:28 PM

Amount *
0.02 ETH

Signature *
0x7a87fd05c4eb21957f6a8475f1ac46db600bd3cb9fe1a8fc330e48f0a5cfa6661c7f
b0a7ee9e95d968f89e6cdd9607ddd2b8211b25022ceaab96c2525024b1a61b

Signer *
0xf34271ade01fe4ca19464fda8b69384e254bd984

Figure 24: Verify Signature, Basic Info



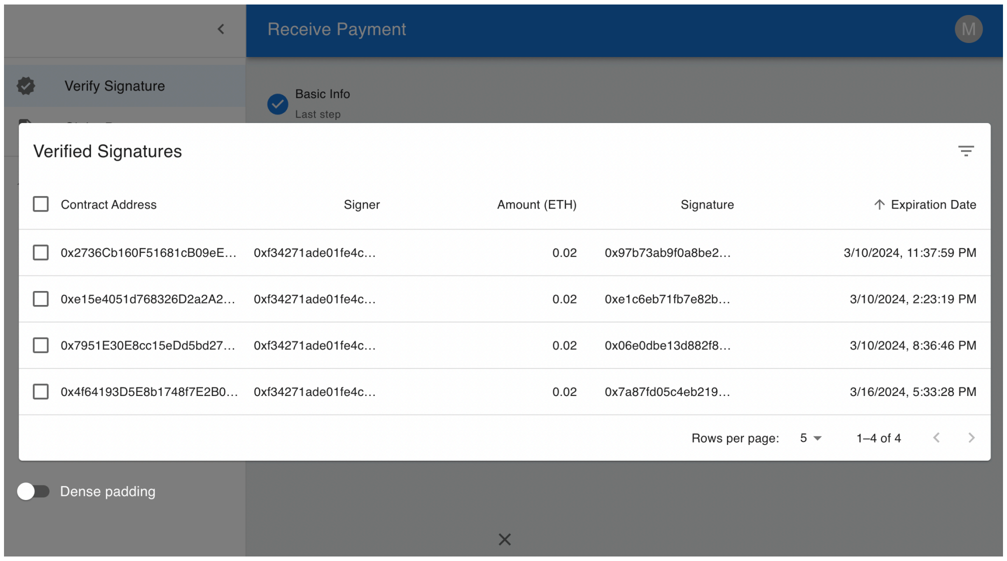
All steps completed - you're finished

The contract will expire on: 3/16/2024, 5:33:28 PM

Signature is verified!

Your signature has been successfully logged.
You can check it at Account settings - My Signatures.

Figure 25: Verify Signature, Result



Verified Signatures

Contract Address	Signer	Amount (ETH)	Signature	Expiration Date
0x2736Cb160F51681cB09eE…	0xf34271ade01fe4c…	0.02	0x97b73ab9f0a8be2…	3/10/2024, 11:37:59 PM
0xe15e4051d768326D2a2A2…	0xf34271ade01fe4c…	0.02	0xe1c6eb71fb7e82b…	3/10/2024, 2:23:19 PM
0x7951E30E8cc15eDd5bd27…	0xf34271ade01fe4c…	0.02	0x06e0dbe13d882f8…	3/10/2024, 8:36:46 PM
0x4f64193D5E8b1748f7E2B0…	0xf34271ade01fe4c…	0.02	0x7a87fd05c4eb219…	3/16/2024, 5:33:28 PM

Rows per page: 5 1–4 of 4

Figure 26: Verify Signature, Log

2. Claim Payment. Figure 27 shows the sequence diagram.

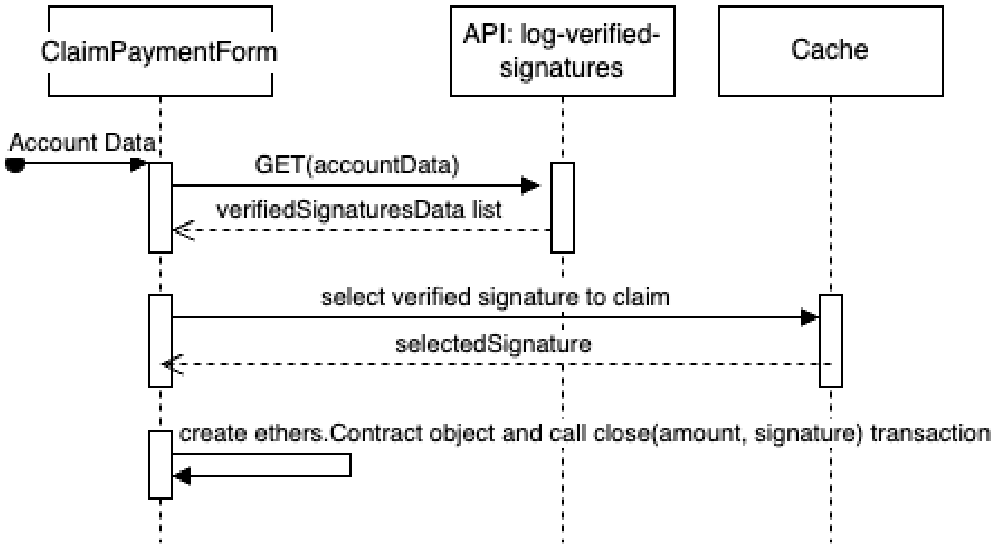
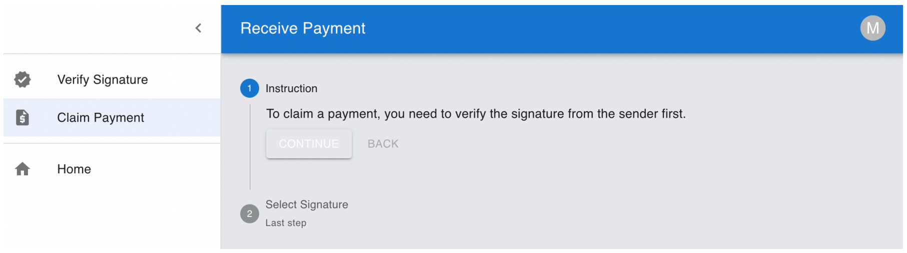

ClaimPaymentForm
API: log-verified-signatures
Cache
Account Data
GET(accountData)
verifiedSignaturesData list
select verified signature to claim
selectedSignature
create ethers.Contract object and call close(amount, signature) transaction

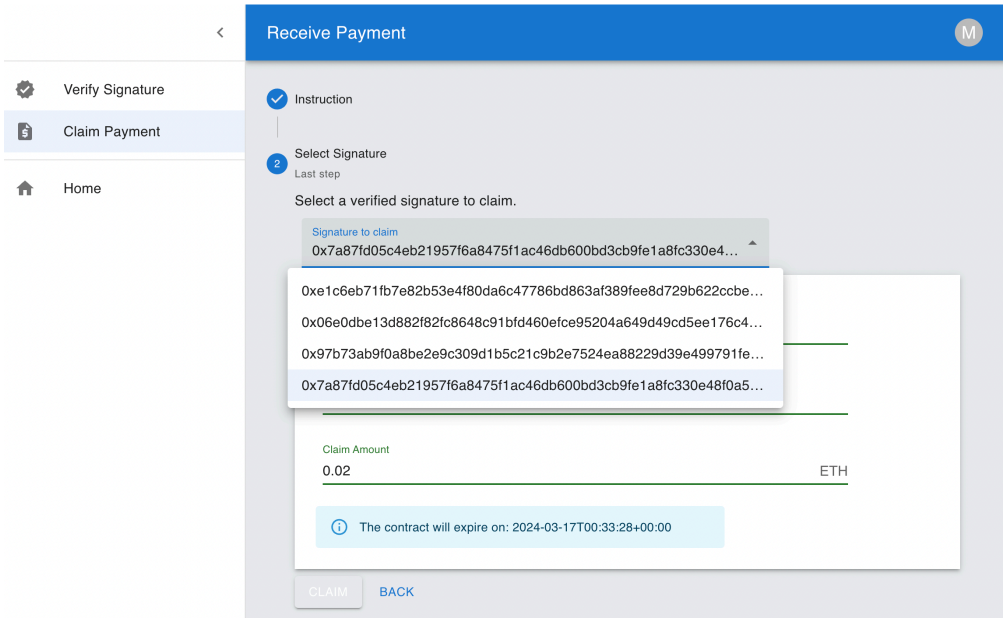
Figure 27: Claim Payment, Sequence Diagram

Figures 28, 29, 30, 31 exemplify the receiver closing a payment channel and receiving the escrowed amount of ETH.



To claim a payment, you need to verify the signature from the sender first.

Figure 28: Claim Payment, Instruction


Receive Payment

M

Verify Signature

Claim Payment

Home

Instruction

Select Signature

Last step

Select a verified signature to claim.

Signature to claim

0x7a87fd05c4eb21957f6a8475f1ac46db600bd3cb9fe1a8fc330e4…

0xe1c6eb71fb7e82b53e4f80da6c47786bd863af389fee8d729b622ccbe…

0x06e0dbe13d882f82fc8648c91bfd460efce95204a649d49cd5ee176c4…

0x97b73ab9f0a8be2e9c309d1b5c21c9b2e7524ea88229d39e499791fe…

0x7a87fd05c4eb21957f6a8475f1ac46db600bd3cb9fe1a8fc330e48f0a5…

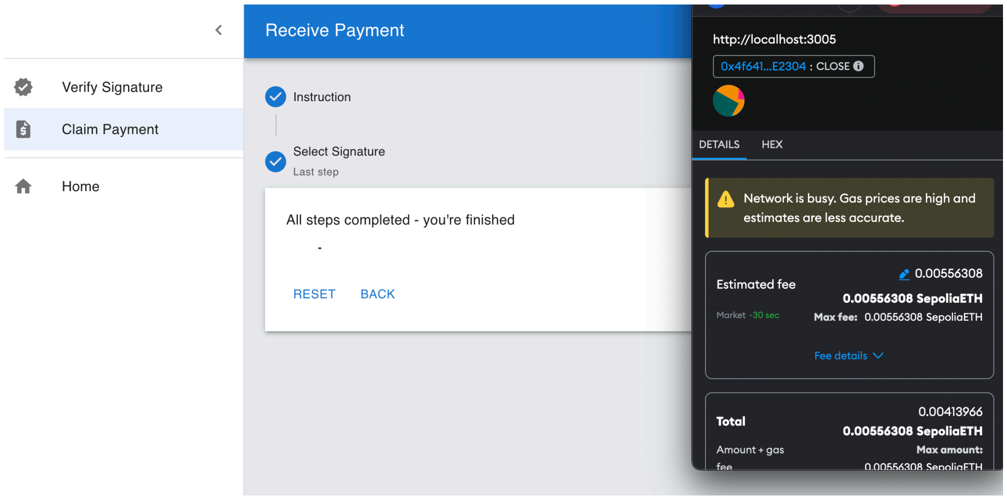
Claim Amount

0.02 ETH

The contract will expire on: 2024-03-17T00:33:28+00:00

CLAIM BACK

Figure 29: Claim Payment, Select


Receive Payment

Verify Signature

Claim Payment

Home

Instruction

Select Signature

Last step

All steps completed - you're finished

-

RESET BACK

http://localhost:3005

0x4f641...E2304 : CLOSE

DETAILS HEX

Network is busy. Gas prices are high and estimates are less accurate.

Estimated fee 0.00556308

0.00556308 SepoliaETH

Market ~30 sec Max fee: 0.00556308 SepoliaETH

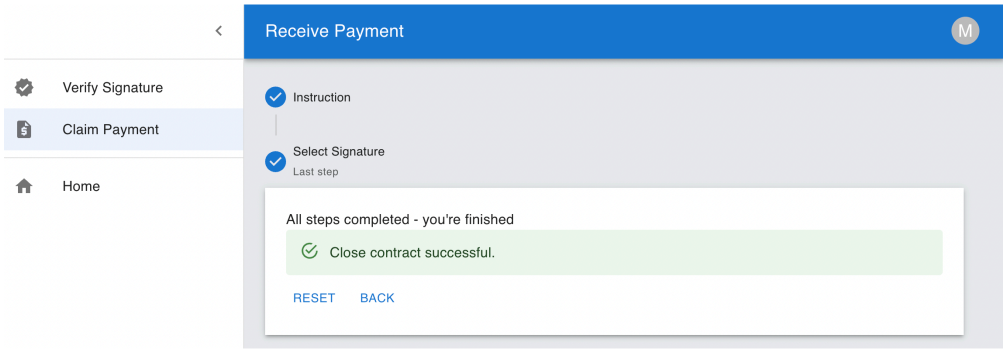
Fee details

Total 0.00413966

0.00556308 SepoliaETH

Amount + gas fee Max amount: 0.00556308 SepoliaETH

Figure 30: Claim Payment, Transaction


Receive Payment

M

Verify Signature

Claim Payment

Home

Instruction

Select Signature

Last step

All steps completed - you're finished

Close contract successful.

RESET BACK

Figure 31: Claim Payment, Result

2.4 Backend

2.4.1 API

Now we support a sequence of POST/GET methods in our API. We are going to use API to help with deploying contracts, processing signatures, contract utilities, logging data, and connecting to our database.

Specifically, we have the following POST/GET/UPDATE API methods:

1. Send Payment
   (a) Deploy-contract: POST, GET
   (b) Create-message: POST
2. Contract Functions
   (a) Get-expiration: POST
   (b) Verify-contract: POST
   (c) Verify-signature: POST
3. Log Data
   (a) Payment-channels
   (b) Payment-signatures
   (c) Verified-signatures

For example, the POST method `verify-contract` has the structure:

```
import { NextRequest, NextResponse } from ’next/server’;
import fs from ’fs’;
import path from ’path’;
import { ethers } from ’ethers’;


interface RequestBody {
    contractAddress: string
}

const alchemyUrl = ’https://eth-sepolia.g.alchemy.com/v2/N6jbViZYGzI-M8RUF
const provider = new ethers.JsonRpcProvider(alchemyUrl);

export async function POST (req: NextRequest) {
    try {
        const bodyJson: RequestBody = await req.json();
        const getCode = await provider
            .getCode(bodyJson.contractAddress);
        const getBalance = await provider
            .getBalance(bodyJson.contractAddress);

        const emptyCode = ’0x’;
        const bytecodePath = path.resolve(
            process.cwd(),
            'contracts',
            'PaymentChannel.bytecode'
        );
        const bytecode = fs.readFileSync(bytecodePath, 'utf8');

        if (getCode === emptyCode) {
            return NextResponse.json(
                { message: "Payment channel does not exist." },
                { status: 404 }
            )
        } else if (getBalance === BigInt(0)) {
            return NextResponse.json(
                { message: "Payment channel escrowed 0 ETH or closed." },
                { status: 400 }
            )
        } else if (getCode !== bytecode) {
            return NextResponse.json(
                { message: "Payment channel has been altered." },
                { status: 405 }
            )
        }
        return NextResponse.json(
            { message: "Payment channel verified!" },
            { status: 200 }
        )
    } catch (error) {
        console.log(error);
        return NextResponse.json(
            { message: "Internal server error." },
            { status: 500 }
        )
    }
}
```

And the POST/GET methods for updating and fetching payment channels data for a given user account are

```
import { NextRequest, NextResponse } from 'next/server';
import { supabase } from '@/app/src/utils/supabaseClient'
import { PaymentChannelLogDataSchema } from '@/app/src/components/schema/Pa

export async function POST (req: NextRequest) {
    try {
        const bodyJson: PaymentChannelLogDataSchema = await req.json();
        const { data, error } = await supabase
```

```
          .from('payment_channels')
          .insert(bodyJson)

        if (error) {
          console.log("insert error", error)
          return NextResponse.json(
            { message: "Bad Request" },
            { status: 400 }
          )
        }
        return NextResponse.json(
          { message: "Success" },
          { status: 200 }
        )
    } catch (error) {
        console.error("Error in POST function:", error);
        return NextResponse.json(
          { message: "Internal Server Error" },
          { status: 500 }
        )
    }
}

export async function GET (req: NextRequest) {
  try {
    console.log("GET - Received request");
    const url = new URL(req.url);
    const account = url.searchParams.get('account');

    const { data: payment_channels } = await supabase.from("payment_channel
      .select('*')
      .eq('account', account);

    return NextResponse.json(
      {
        message: "Success",
        paymentChannelData: payment_channels
      },
      { status: 200 }
    )
  } catch (error) {
    console.error("Error in GET function:", error);
    return NextResponse.json(
      { message: "Method Not Allowed : (" },
      { status: 405 }
```

```
        )
    }
}
```

2.4.2 Data Management

In the development phase, we use JSON files, which can store any payment channels opened and signatures redeemed.

For the web dashboard, we link our project to Supabase, which offers scalable, PostgreSQL-based databases that provide real-time capabilities, user authentication, and easy integration with modern web and mobile applications, allowing us to quickly build and deploy full-fledged applications.

Figure 32 shows the Supabse Table editors, where we have three tables in our web-dashboard organization:

1. `payment_channels`
2. `payment_signatures`
3. `verified_signatures`

The schema for `payment_channels` and `verified_channels` are

```
create table
  public.payment_channels (
    id bigint generated by default as identity,
    created_at timestamp with time zone not null default now(),
    account text null,
    contract_address text null,
    escrow_amount double precision null,
    recipient text null,
    expiration timestamp with time zone null,
    max_payment double precision null,
    status public.channel_status null,
    constraint payment_channels_pkey primary key (id)
  ) tablespace pg_default;

create table
  public.verified_signatures (
    id bigint generated by default as identity,
    created_at timestamp with time zone not null default now(),
    contract_address text null,
    amount double precision null,
    account text null,
    signer text null,
    signature text null,
    expiration timestamp with time zone null,
    constraint verified_signatures_pkey1 primary key (id)
) tablespace pg_default;
```

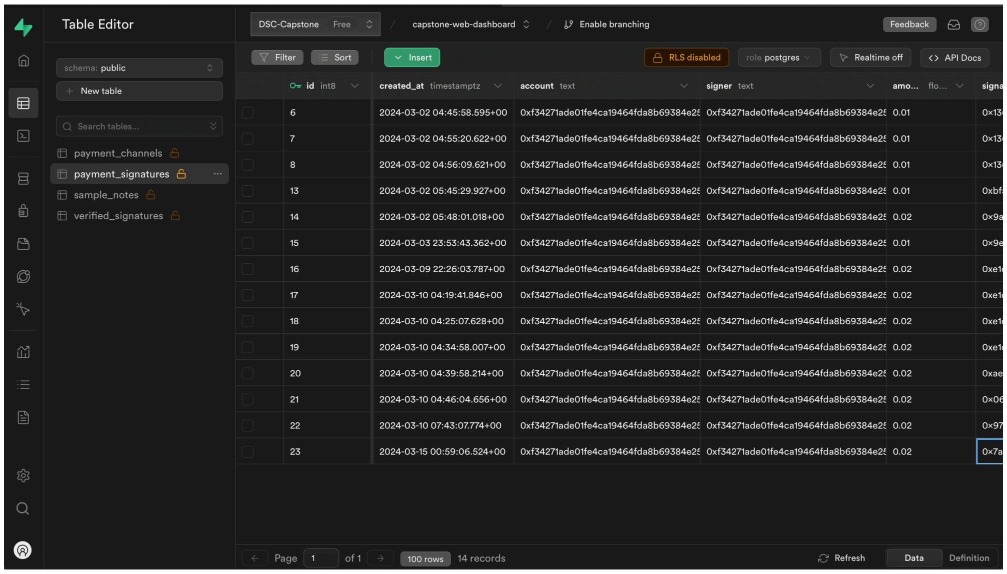


Table Editor

DSC-Capstone Free / capstone-web-dashboard / Enable branching

schema: public

New table

payment_channels
payment_signatures
sample_notes
verified_signatures

Filter Sort Insert RLS disabled role postgres Realtime off API Docs

id int8	created_at timestamptz	account text	signer text	amo… flo…	signa
6	2024-03-02 04:45:58.595+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.01	0x13
7	2024-03-02 04:55:20.622+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.01	0x13
8	2024-03-02 04:56:09.621+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.01	0x13
13	2024-03-02 05:45:29.927+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.01	0xbf
14	2024-03-02 05:48:01.018+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0x9a
15	2024-03-03 23:53:43.362+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.01	0x9e
16	2024-03-09 22:26:03.787+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0xe1
17	2024-03-10 04:19:41.846+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0xe1
18	2024-03-10 04:25:07.628+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0xe1
19	2024-03-10 04:34:58.007+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0xe1
20	2024-03-10 04:39:58.214+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0xae
21	2024-03-10 04:46:04.656+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0x06
22	2024-03-10 07:43:07.774+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0x97
23	2024-03-15 00:59:06.524+00	0xf34271ade01fe4ca19464fda8b69384e25	0xf34271ade01fe4ca19464fda8b69384e25	0.02	0x7a

Page 1 of 1 100 rows 14 records Refresh Data Definition

Figure 32: Supabase Table Editor

3 Results

In this project, we successfully created a decentralized application that along with MetaMask can

- Connect to MetaMask
- Deploy payment channels to the Sepolia Testnet
- Interact with deployed payment channels in the Sepolia Testnet
- Create and verifying payment signatures
- View databases with relevant information about ongoing payment channels and payment signatures

all in a single decentralized application.

4 Discussion

In this project, we successfully condensed the functionalities of four of the five applications, namely Solidity code, Remix IDE, Sepolia Etherscan, required to create and use a payment channel into a single decentralized application, which reduces the number of applications required to just two: MetaMask and this decentralized application. Furthermore, we added the extra functionality of databases that keep track of ongoing payment channels and payment signatures.

However, it is currently limited as a one-way payment channel, where a sender can only send but not receive and a receiver can only receive but not send. Future improvements to this payment channel might create a two-way payment channel, where both people can send and receive. If this is possible, the user-interface would have to be updated accordingly.

This application is also limited by using the Sepolia Testnet. Future implementations require more rigorous thinking about potential bugs and testing along with subsequent fixes if required before transitioning this application from the Sepolia Testnet to the Ethereum mainnet. This statement also extends to the Solidity code used for the payment channel smart contract. This code is from Solidity by Example's section on micropayment channels. This code requires more rigorous thinking about potential bugs and testing along with subsequent fixes if required before transitioning this application from the Sepolia Testnet to the Ethereum mainnet.

References

et al. 2019. "Solidity by Example."
Nakamoto, Satoshi. 2008. "Bitcoin: A peer-to-peer electronic cash system."