Spark - Bitcoin L2

#BitcoinL2

*Spark

https://www.lightspark.com/

会社名は "lightspark" で、Lightning Network ソリューションの会社。

Spark は L2ソリューション。

Spark

https://www.spark.money/

https://github.com/buildonspark/spark

https://docs.spark.money/home/welcome　: タブが3つあるので注意

追加情報

Grok: Lightspark SDKとConnect、 Sparkの関係

Grok2 : SparkとStatechain方式のL2比較(長い)

Grok3 : Wallet of Satoshi: Spark Integration Explained

ChatGPT

Wallet of SatoshiとLightsparkの提携 - Gemini

Sparkプロトコルは一般的にセルフカストディと認められているか？ - Gemini

プロダクト

- Lightspark Connect
  - Lightsparkが提供する開発者向けのLightning Network接続プラットフォーム
  - Spark とは直接関係ない？
- Spark
  - L2
- UMA
  - Bitcoinアドレスっぽい感じで送受金するそうだ。詳しく見てない。

2025年10月現在、mainnet で運用している(連動して動いている)

- 2025年7月は Magic Eden の AMM 関連がうまく動かなかったらしい
- 7月以降に Unilateral Exit が正式実装されたらしい

2025年7月1日

Wallet of Satoshi Integrates Spark: Self-custody lightning is so back

2026/04/22 追加調査

AIにいろいろ質問

https://docs.spark.money/start/overview

- ユーザとSpark間もFROST(2-of-2)だが簡易的なFROSTを使っている
  - 現在はSpark社とFlashnet社の2つだけSpark Operator(SO)で、彼らがSpark側の鍵管理をしている
    - SOはSparkの状態機械を運営する主体。鍵シェアの保持、FROST署名、リーフの状態遷移（デポジット・転送・協調exitのトランザクション署名）など、プロトコルの中核処理を担う
    - SOはパーミッション型で自由参加はできない
  - それとは別にSpark Service Provider(SSP)は自由に参加できる
    - デポジット／出金、Lightning送受信などをユーザー向けサービスとして提供する事業者。ユーザーがSparkを使いやすくするレイヤー
    - SSPの役割
      - Sparkへの入金／Sparkからの出金を円滑化するサービスを提供
      - オプションでLightningプロバイダーも務め、Sparkユーザー向けにLightning送受信を可能にする
      - 1つのSpark内に複数のSSPが共存可能
      - SSPは**必須ではなく**サービスレイヤーという位置づけ
        - おそらくSSPがないとLNは使えないのだろうが、SparkなりFlashnetなりがSSPもやればよいだけなのでSSPをやる人がいなくても困りはしないだろう。
- デポジットしたUTXOはSpark内ではleafと呼ばれる単位で管理される
  - leafは分割できる
  - 希望した額に分割して別のユーザにtransferできる
    - leafの所有者が変わるときに**statechain**を使ってオフチェーンで別のユーザとの2-of-2にするということができる
  - Spark内でのtransferは、なんと手数料ゼロ！ しかも即座に完了！
  - leafはSOがそれぞれ管理している。すなわち現在はSpark社とFlashnet社が管理している。
- Cooperative Exitした場合には直接指定したアドレスに送金するトランザクションを作るので、デポジットした額より多くなっても払うことができるそうだ。
  - 所有者が移動していってもたどっていけばデポジットしたUTXOにたどり着く。
  - その状態でUTXOの所有者が協調exitするとデポジットしたUTXOは閉じられるそうだ。そして残ったleafがどう管理されるかはドキュメントに載ってないらしい。

Ark - Bitcoin L2と似ている気がする

UTXOをL2で扱うという点を参考にしているとかなんとか。プロトコルを参考にしているのではなく考え方か。

https://chatgpt.com/share/69e9cb3a-d4e8-8323-8fad-66f1d58e90f3

GitHubで公開されているが、これで全部かどうかわからない

statechain方式の弱点

SparkではL2でのお金を leaf という単位で扱っている。
leaf は所有するユーザとSpark運営するSOとの2-of-2のようなものである(実際は簡易FROSTを使った集約鍵)。
別のSparkユーザに送金するとき、leafを所有するユーザが別の人になるが、そのたびにオンチェーンに展開すると手数料などが馬鹿にならない。
Sparkでは statechain という方式を使って、オフチェーンのまま2-of-2のユーザ側だけ鍵を交換するということができる。

statechainの弱点は、前の所有者が鍵を破棄しないといけないということ。破棄したことを証明することは数学的にもできないので信用するしかない。しかし破棄しないまま前の所有者がUnilateral Exitで一方的に抜けるということができてしまう (Limitations / Attack)。

現在の正しい所有者は、一定の時間ウィンドウ内に正しいleafトランザクションを公開する必要があります。期限内に行わないと攻撃者がそのUTXOを請求できてしまいます。

SO（Spark Operators）全員が署名済みトランザクションのコピーを保持し、現在のleaf所有者の代わりにwatchtowerとして機能します。

watchtowerは有害なデータ（toxic data）を保持しないため、ユーザーは任意の第三者にwatchtower機能を委譲することも可能です。

攻撃者はブランチ全体を公開しCPFPで各ツリーノードを手数料負担する必要があるため、攻撃コストが高くなるよう設計されています。

つまり、これはLightning Networkで過去の取引のときに使った commitment transaction を展開するのと同じパターンだ。ただ LN の場合はそれを検知するとやらかした相手から全額奪い取れるが、SparkはUTXOを守るだけになる。その代わりに攻撃するコスト自体が高いそうだ。

2026/04/22 オープンソース調査

quickstart/create-walletにセットアップ手順が書いてあるが、初っ端から動かない。

直接呼び出すことにした

https://docs.spark.money/tools/cli

https://www.npmjs.com/package/@buildonspark/cli

https://github.com/buildonspark/spark/tree/main/sdks/js/apps/spark-cli

setup

```
$ mkdir work
$ cd work
$ npm i @buildonspark/cli
$ node node_modules/@buildonspark/cli/dist/cli.mjs

 Available commands:
...中略...
⚠️  WARNING: This is an example CLI implementation and is not intended for
production use. Usee at your own risk. The official package is available at
https://www.npmjs.com/package/@buildonspark/spark-sdk  ⚠️
>
```

init

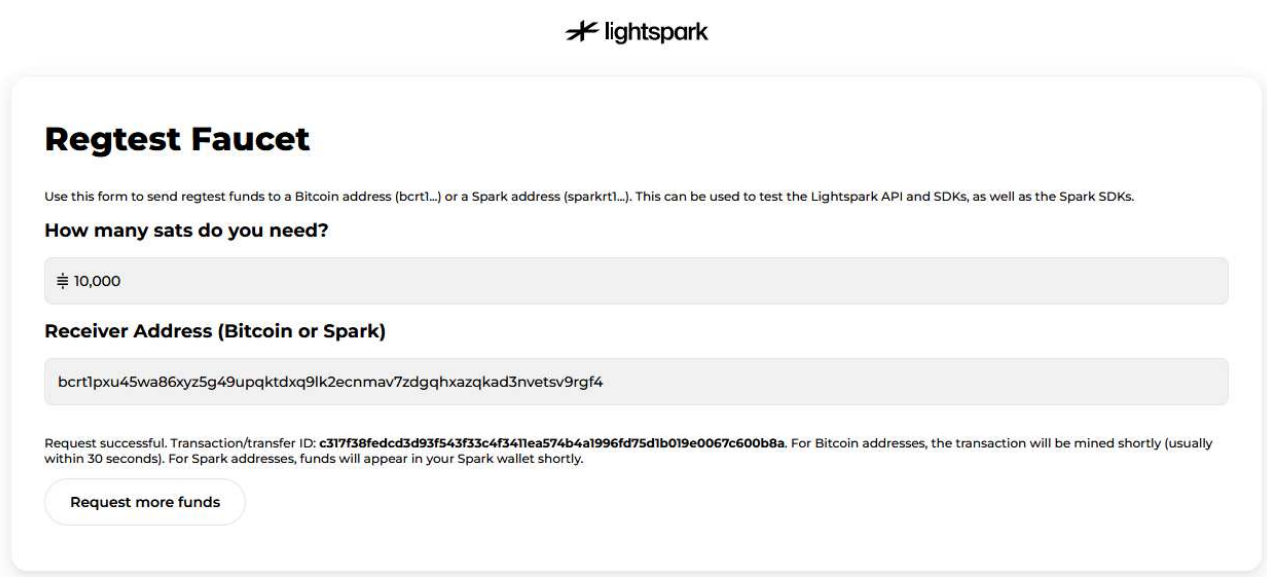
```
> initwallet
Mnemonic: direct high lobster cheap public rib fetch tribe retreat street fat
possible
Network: REGTEST
> Stream connected

> getstaticdepositaddress
This is a multi-use address.
bcrt1pxu45wa86xyz5g49upqktdxq9lk2ecnmav7zdgqhxazqkad3nvetsv9rgf4
```

このアドレスにローカルのregtestから送金したが、それでは意味がなかった。

faucetサイトからこのアドレスに送金っぽいことをするようである。

https://app.lightspark.com/regtest-faucet




```
init
  > getlatesttx bcrt1pxu45wa86xyz5g49upqktdxq9lk2ecnmav7zdgqhxazqkad3nvetsv9rgf4
  c317f38fedcd3d93f543f33c4f3411ea574b4a1996fd75d1b019e0067c600b8a
```

Faucetサイトに表示されているTXIDと一致している。

手元で動かしているregtestは `txindex=0` のためTXIDでは検索できないが、 `gettransaction` で追えないのでfaucet経由でリモートが制御されたということはないと思う。

```
init
  > claimstaticdepositquote
  c317f38fedcd3d93f543f33c4f3411ea574b4a1996fd75d1b019e0067c600b8a
  {
    transactionId:
  'c317f38fedcd3d93f543f33c4f3411ea574b4a1996fd75d1b019e0067c600b8a',
    outputIndex: 0,
    network: 'REGTEST',
    creditAmountSats: 9901,
    signature:
  '30440220445e186f16fff44d84f7c4e0bb151cd0c719794c41b1edbdac4a30b1b6a17f9802202a7579
  82ec8f369c29aa1453b2b6e75a008c2a6cef3dd5463085fc8f8343fc88'
  }

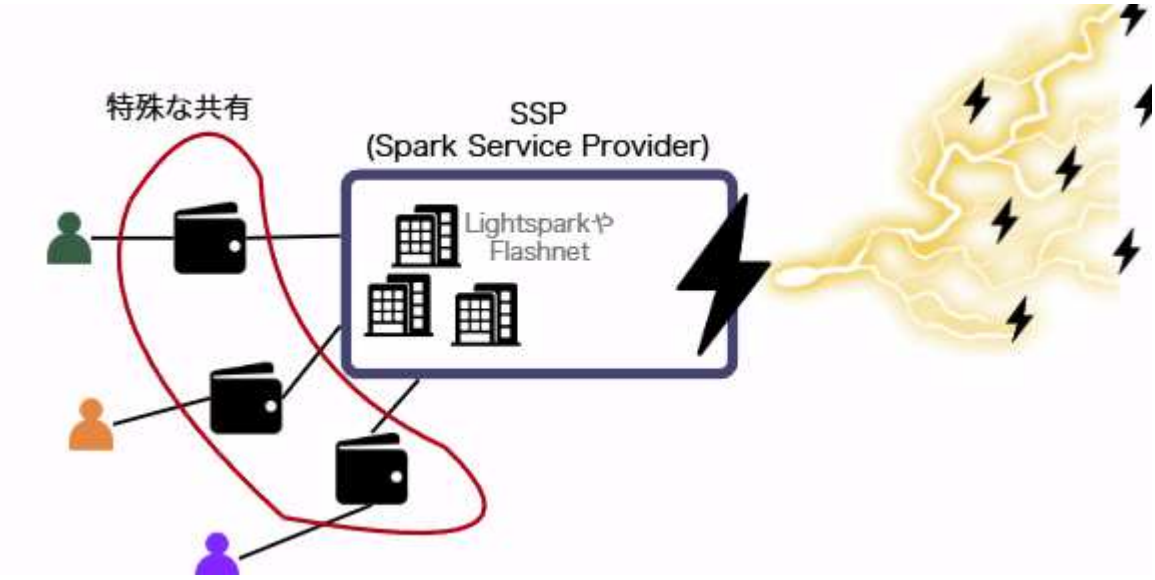
  > claimstaticdeposit
  c317f38fedcd3d93f543f33c4f3411ea574b4a1996fd75d1b019e0067c600b8a 9901
  30440220445e186f16fff44d84f7c4e0bb151cd0c719794c41b1edbdac4a30b1b6a17f9802202a75798
  2ec8f369c29aa1453b2b6e75a008c2a6cef3dd5463085fc8f8343fc88
  { transferId: '019db3ec-adf7-72a1-8025-8111c2124cfc' }
```

```
> Transfer 019db3ec-adf7-72a1-8025-8111c2124cfc claimed. New balance: 9901
Stream reconnecting 1 Infinity 1000 /spark.SparkService/subscribe_to_events
INTERNAL: Received RST_STREAM with code 2 (Internal server error)
Stream connected
Stream reconnecting 1 Infinity 1000 /spark.SparkService/subscribe_to_events
INTERNAL: Received RST_STREAM with code 2 (Internal server error)
Stream connected

> getbalance
Sats Balance:
  Available: 9901
  Owned:     9901
```

概要

Spark 自身は Lightning Network とは別のネットワークである。
ステートチェーンプロトコルで、UTXO を 2-of-2 で共同管理する「共同カストディ」モデルという方式。
ユーザは Lightspark や Flashnet などの運営者連合と
運用側は SSP(Spark Service Provider)と呼ばれたり、バリデーターたちは SOs(Spark Operators)と呼ばれたりする。



スマートコントラクト / トークン

VMはなくスマートコントラクトもない。
ステーブルコインを発行する計画はあるようだ(2024/10/25 introducing spark)。
RGB、Taproot Assets、LRC-20 なども名前に上がっていた(Lightsparkの人が発表した情報？)。

- RGB
  - 理論上は可能だが。。相手ノードやチャネル、ルートも RGB に対応していないといけないのでほぼ無理だろう。
- Taproot Assets
  - RGB と違って送金元と送金先が対応していれば中間は Bitcoinルートでよいそうなので対応しやすいだろう。
- LRC-20
  - Ordinals の Inscription らしい(https://github.com/shrdlu2/lrc-20)
  - なんでこれを挙げたんだろう？ testnet で試験するためとか？

Unilateral Exit

ユーザーが一方的に抜ける Unilateral Exit ができる。Sparkサービスが停止しても抜けることができることになる。
コストは高い。
Spark内部でユーザ資金はツリー構造で管理されていて、ツリーの root から leaf に向けて順番にトランザクションを展開する必要がある。16,348 sats より大きい leaf しか exit できない(exitする手数料の方が高くなって意味が無くなる)。

Statechain(2025/10/08)

- Core Concepts - Spark
- Spark での Statechain: 2025年10月 (ChatGPT) (Grok) / 2025年7月 (ChatGPT)(Gemini3)
- statechains - Bitcoin Optech

"statechain"　はオフチェーンでUTXOの所有権を移転するしくみ。再委譲もできる。
移転する際はトランザクションの発行は行わず、鍵操作だけで行う。
方式

- 署名アダプター
- Eltoo
- ECDSA方式
- FROST変形方式
  - Sparkはこれ。もしかしたら「FROST方式」があって、Spark はカスタマイズしていてこの名前なのかもしれない。

Spark での UTXO管理

カスタマイズされた FROST署名手順

SOs が n人中 t個の署名(SE)が必要、という部分については FROST署名

ユーザと SE の署名が必要な部分は方式が書かれていないが加算集約するそうだ。

全体としてカスタマイズしたFROST署名という呼び方

おおよその手順

はじまり

ユーザーがBTCを自分の鍵とSEのFROST管理鍵の両方で制御される共有署名アドレスに送金する

送金前に Unilateral Exitトランザクションに事前に署名しておく

所有者が変わってからも展開できてしまうが、所有者が変わるときによりタイムロックが短い Unilateral Exitトランザクションを作るので、ちゃんと気付けば取り戻すことができる。

以降、UTXO は「Sparkリーフ(リーフは仮想的なUTXOという意味)」という表現になりオフチェーンで転送できる

所有権の委譲

1. 新しい FROST キーセットを生成

前所有者は所有権を移すオフチェーンのトランザクションで SE のキーシェアを作り直しを要求する

SE は FROST を使用して新しい所有者用のキーシェアに対応した新しい閾値鍵を作る(FROSTの鍵シェア再生プロトコル)

暗号学的セレモニーによって SEのキーシェアを調整(tweak)し、新しい所有者の鍵と新しいキーセットを使った FROST 2-of-2 で署名できるようになる

新しい所有者がキーシェアを受け取ると、今後この UTXO の署名に参加できる人となり前の所有者は参加できなくなる

2. 前の所有者はキーシェアを破棄

前の所有者はともかく、SE が前のキーシェアを無効化しないと意味が無い

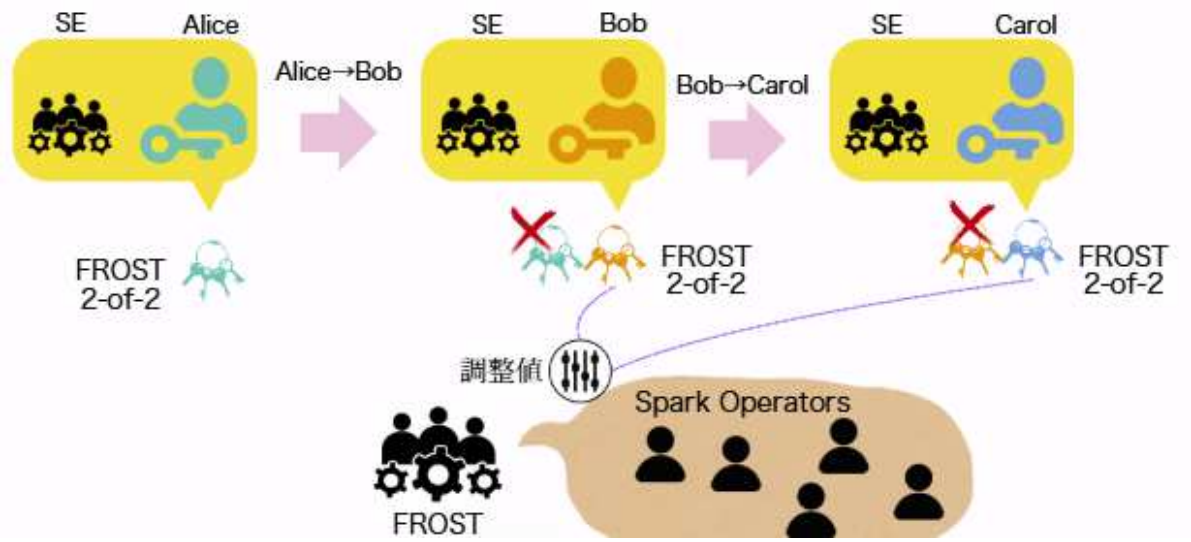
が、削除したことを暗号学的に証明する方法はない

ここがユーザーと共謀できるポイントになる

3. 安全に引き継いだことをオフチェーンに記録する

所有者が移ったことを証明する「statechain update」を作って SOs が署名してオフチェーンに記録する

これは最新の所有者が誰かを示す証拠になる



批判

Voltage vs. Lightspark: Choosing the Right Partner for Lightning Payments

Voltageの記事で、ビジネス的な比較をしただけ。大したことは書いてなかった。

ステートチェーンでのトラストモデルが曖昧だという批判がある。

いくつかのシーンで過去の秘密鍵を削除する必要がある。これは Lightspark 側もきちんと削除することを信用していることになる。この辺が不信を抱かれているらしい。

削除といってもローカルで保持しておけばよさそうではある

鍵が変わるというのは HDウォレットで世代が変わるということ？

Grokとしては批判はそれほど広くは発展していないという見方。

まあ、技術のわかる人以外は騒がないし、よそのサービスだから騒ぎ続けるようなものでもないのだろう。

---

https://github.com/buildonspark/spark

mise

https://mise.jdx.dev/getting-started.html

```
$ curl https://mise.run | sh
```

atlas

https://atlasgo.io/guides/evaluation/install

```
$ curl -sSf https://atlasgo.sh | sh
```

protoc関連

https://github.com/protocolbuffers/protobuf/releases

https://github.com/bufbuild/protoc-gen-validate/releases

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

PostgreSQL

使っていなかったら現在のアカウントでROLE設定しておくと良いだろう。

atlas というツールを経由して使われるのだが、パスワードを付けないときも付けたときも `run-everything.sh` などでは動かなかった。 `postgres://<アカウント>:<パスワード>@localhost...` のように明示的に書いた。

add_role

```
$ sudo su postgres
$ ROLE="アクセスさせたいアカウント"
$ PASS="あなたの使いたいパスワード"
$ echo "CREATE ROLE ${ROLE} CREATEDB LOGIN;" | psql
$ echo "ALTER ROLE ${ROLE} with PASSWORD '${PASS}';" | psql #パスワードを付けないなら不要(NULLで削除)
```

tmux

```
$ sudo apt install tmux
```

run-everything.sh

```
$ ./run-everything.sh
```