postchain_docs Documentation

Release 0.1

August Botsford

May 23, 2018

Contents

Postchain is a modular framework for implementing custom blockchains. Particularly, it's geared towards consortium blockchains (also known as permissioned, enterprise, private, federated blockchains, and, sometimes, distributed ledger technology).

This model is also known as proof-of-authority, to contrast it with proof-of-work and proof-of-stake. A Postchain network consists of a number of nodes, each maintaining an identical set of data. The key point of difference between Postchain and other private blockchains is that it integrates with SQL databases in a very deep way: all blockchain data is stored in an SQL database, transaction logic can be defined in terms of SQL code (particularly, stored procedures). However we should note that Postchain uses SQL as a black box: it is not a database plugin and it works with databases such as PostgreSQL as is, without any special configuration or modification. SQL database tools meet the requirements of a blockchain (atomicity, determinism, rollback) while allowing for greater data complexity, easier deployment, and more secure integration with existing systems.

A whitelist of transaction types defines the rules and capabilities of the system, and transactions which are valid according to the rules of the system are distributed to all nodes using a consensus method which ensures that all nodes arrive at an identical state as long as a majority of nodes are functioning correctly. Committed transactions are stored in a blockchain in the database; a cryptographically secured assertion of state which is difficult to modify, and can be used to reconstruct the entire state of the database.

CHAPTER 1

Contents

1.1 Overview

1.1.1 Public/private blockchains

If we take a Bitcoin as a primary example of a blockchain, there are a number of properties which are specific to cryptocurrencies:

- Fixed supply
- Censorship resistance
- No counterparty risk
- No intermediaries (in transactions)

but these features aren't relevant in broader enterprise contexts. On the other hand, the following features might be relevant:

- Decentralization (no single entity is in control)
- Byzantine fault tolerance (failure of a fraction of nodes does not jeopardise the system)
- End-to-end cryptographic authentication of transactions

Note that properties listed above overlap, and can be also seen as different facets of a single feature: fault-tolerant multiparty data sharing.

1.1.2 Problems with existing blockchains

- Blockchains usually rely on key-value data stores.
- Indexing and history tracking is often done through external processes which increases complexity
- If you have data in two places, which of them do you trust?

1.1.3 Ethereum as a data store

Ethereum offers two data retrieval APIs:

- Object-oriented API through Solidity constant functions
- Index access through web3 & RPC APIs

Both are significantly less convenient and powerful than the relational queries offered by a conventional DBMS.

1.1.4 Consortium database

If the blockchain manages data, it should do it properly. This means:

- A rich data model
- History tracking
- Indexed access
- Flexible queries
- Transactions
- Constraints (security)

Many of these capabilities are features of a relational database. We asked ourselves:

What if we could use a relational database as a blockchain data store **AND** Define blockchain logic using SQL stored procedures?

Postchain is the result of this line of inquiry. It can be understood as components which allow one to implement blockchain-like synchronization on top of an ordinary SQL-hashed database.

In a postchain-based system clients cannot write data directly into a database using SQL queries. Instead, clients submit transactions to the system in form of signed messages which are sent to validator nodes. Validators check if a transaction is well-formed and is properly authorized before applying it.

All validators must apply transactions in the same order so that they all arrive to an identical database state. If every transaction is deterministic, then applying a sequence of transaction will result in the same final state on all nodes. Agreement on transaction order is also necessary to deal with conflicting transactions. Ordering will make sure that all nodes react to conflicting transactions in the same way.

Transactions are organized into blocks. A block is a sequence of transactions with synchronization-related metadata attached to it. Blocks are organized into a chain, as each block points to a previous block. An ordering of blocks in the chain together with an ordering of transactions within the block defines the ordering of transactions within a blockchain, which results in the same database state on every synchronized node. In other words, a blockchain is used for database replication.

It is important to highlight that all of the blockchain state must be in the database. Transaction-handling logic should never rely on data stored elsewhere. This is essentially the big idea about postchain: by keeping all the state in an SQL database, we can rely on its transaction isolation and other ACID properties. This greatly simplifies transaction-handling code as it doesn't need to bother itself with mempool or other implementation details, it should simply interact with the provided database connection. The transaction handling implementation essentially boils down to a single function which validates transactions in the context of the current state of the database and applies changes by performing SQL queries.

1.1.5 Consensus

Validators use a consensus algorithm to select client-submitted transactions into the next block. Transactions are committed to the database only after a super-majority of validator nodes agree on block contents.

A consensus algorithm is also used to synchronize the blockchain across nodes. By default Postchain uses EBFT [LINK] consensus, but can work with other consensus backends.

1.1.6 Postchain structure

Postchain does not require any particular format for transactions and blocks, or a specific consensus algorithm. At its core, Postchain merely defines interfaces of transaction and block components and the general architecture of the system, so different components can work together as long as they implement compatible interfaces. This enables code reuse. For example, new transaction handling logic (new database schema, queries, etc) can be combined with an existing block synchronization algorithm.

1.1.7 Transaction messages

Postchain can work in transactions in any format. It is even possible to use message formats defined for other system, for example, Bitcoin transactions or XML documents used in enterprise applications.

1.2 Architecture

Postchain consists of the following components:

- Core: Defines common interfaces which allow different modules to interoperate with each other.
- Base: Defines base classes which are geared towards enterprise blockchains. They can either be used as is, or serve as a base classes for customization.
- GTX: Defines a generic transaction format. It is a general-purpose format with several useful features, such as native multisignature support. It is optional (it is easy to define custom format), but is recommended. GTX also offers a way to define modules with blockchain logic which can work together with each other.
- API: REST API for submitting transactions and retreiving data.
- EBFT: Consensus protocol, based on PBFT. (Replaceable.)
- Client library/SDK: Currently only JavaScript SDK is available, in future we will offer SDK for JVM. Client library contains functions for composing and parsing transactions, interacting with Postchain node using client API.

1.2.1 Core

The goal of Postchain core is to define interfaces which help with code organization, enable interoperabiity between different modules. It also implements several commonly used classes and functions.

Core should be flexible enough for following scenarios:

- building custom blockchains (public or private/federated)
- indexing data of an existing blockchain (Bitcoin, Ethereum, . . .)
- implementing sidechains (which combine custom blockchains with public blockchains)

What is included in core:

- class interfaces (implied): Block, Transaction, BlockFactory, TransactionFactory
- base implementations: AbstractBlock, BlockFactory, TransactionFactory
- utilities: Storage (pg connector), crypto utilities
- "example" implementations: BlockStore, ByteTx, ByteTxStore, SimpleSignedBlock

It's not clear if "example implementations" are actually a part of the core, or are just examples.

Perhaps core should include more stuff, like logging.

Since JS has no notion of interfaces, they will be defined in documentation.

Framework documentation should also include recommendations on schema design, best practices, etc.

1.2.2 Base

The goal of base is to provide tools for building custom blockchains, particularly enterprise private/federated blockchains. Ideally, it should be possible to use it both as a complete solution, and as a collection of building blocks and helpers.

Base includes the following component:

- inter-peer networking, with ASN.1-based message format and ECDSA message signing
- an implementation of Block which is designed for proof-of-authority kind of blockchain (having a room for validator signatures)
- a PBFT-derived consensus algorithm, designed to work with aforementioned Block implementation
- compiler for a Ratatosk-like language which is used to define transaction implementations (it is also possible to implement it in a custom way)
- client-facing RPC server (submit transactions, query-data)
- client SDK
- some minimal configuration

1.2.3 GTX

Postchain GTX makes it possible to compose application from modules. Using pre-made modules can help to reduce implementation time.

Currently ChromaWay offers only two GTX modules (which are bundled with Postchain code):

- Standard – implements time lock and transaction expiration feature.
- FT – Flexible Tokens, provides a simple way to implement tokens. (Can be used in applications like payments, loyalty points, crowdfunding, securities trade, . . .)

GTX transaction format has following features:

- Format is based on ASN.1 DER serialization (standardized by ITU-T, ISO, IEC)
- Has native support for multi-signature
- Has native support for atomic transactions

GTX transaction is consists of one or more operations, and each operation is defined by its name and list of arguments. E.g. one transaction might encode two operations:

- issue(<Alice account ID>, 1000, USD)

- transfer(<Alice account ID>, <Bob account ID>, 100, USD)

This looks similar to making function calls, so GTX operations can be understood as a kind of RPC, where client submits calls (operations) to be performed on server (network of Postchain nodes). GTX transaction is a batch of such operations signed by clients which wish to perform them. Usually operations update database, but they might only perform checks. GTX transaction is atomic: either all operation succeed, or it fails as a whole. GTX modules

Postchain provides a convenient way to define GTX operations and organize them into modules. Multiple modules can be composed together into a composite module.

Besides operations, modules also define queries which can later be performed using client API.

1.2.4 GTX client SDK

GTX client SDK for JavaScript is provided in postchain-client npm package. Example of use is provided in README file here.

1.2.5 FT

FT comes with its own client library for JS which also includes examples, see *Postchain FT*

1.2.6 Network

Network components take care of "boring" stuff like messaging, signing messages, checking message signatures.

MVP implementaiton will use a rudimentary static configuration, future versions might use dynamic reconfiguration.

1.2.7 Consensus algorithm

Consensus algorithm is inspired by Castro's PBFT, but works on block level and restricts concurrency as much as possible. (It seems IBM is taking similar approach in hyperledger/fabric/consensus/simplebft.)

1.2.8 Block implementation

Might be very similar to SimpleSignedBlock, except with ASN.1 serialization format.

1.2.9 Compiler

A Ratatosk-like language can be used to define a list of actions:

```
(defblockchain ledger
    (actions
        (send-money ((from :type pubkey) (to :type pubkey) (amount :type integer))
            (guard (signatures from))
            (sql (foobar_send_money from to amount)))))
```

A compiler will take this description and use it to define:

- ASN.1 serialization format for transaction
- Transaction classes which handle deserialization and check authorization according to specified guard clauses
- Store classes

Together with SQL schema definition this is enough to define a custom blockchain in the context of Kit.

1.2.10 Client RPC

Postchain nodes will both serve as peers in a Postchain network, and serve as a RPC server for clients.

Clients should be able to:

- submit a transaction; transaction is automatically routed to a primary
- check transaction status
- query blocks & transactions
- (TBD) transaction creation helper (client sends JSON, server constructs & signs transaction)
- (TBD) query blockchain data (???)

One problem is that queries are blockchain-specific. We can either leave that to blockchain implementors (it's not hard to query the DB and return data...), but we might also make use of aforementioned compiler & language to define blockchain-specific queries.

(Another possibility is to user a 3rd-party REST API for read queries, e.g. Postgrest.)

1.2.11 Client SDK

Client SDK should include tools to construct transactions & sign them. It might be necessary to link with compiler-generated message codecs. (One option is for compiler to generate SDK bundle for a specific blockchain.)

1.2.12 Configuration

We should make it easy for user to launch a Postchain Kit server for a specific blockchain(s) and using a pre-defined network peer configuration. In the initial version we only need to make it work with a static configuration.

1.2.13 Transactions

However, to benefit from blockchain-like features, transaction messages should be:

- As simple as possible; they should be easy to analyze and leave no room for ambiguity
- Carry a proof of authorization within them (e.g. transaction signatures)

For example, a good transaction message should include only a type of operation to perform and necessary data, as well as signatures which authorize the operation. For example, an e-money transaction message might be (pay, sender, recipient, amount, sender's signature) tuple.

A message which includes an SQL query in a textual form is an example of a bad message format. While it's technically possible, it will result in poor security characteristics, as it's hard (or impossible) to check whether a free-form SQL query is properly authorized.

1.2.14 Transaction logic

Normally transaction-handling logic is implemented in a JavaScript class which implements a transaction. However, this class is supposed to interact with an SQL database, and thus logic can also be implemented in database's stored procedures.

We recommend to implement cryptography-related parts in JS, and state manipulation and validation parts in SQL, but it's not a hard requirements.

1.3 Postchain FT

1.3.1 Background

Postchain FT is a module which implements tranferrable tokens. This library provides a way to create FT transactions and send them to a Postchain node, and also query information from Postchain node.

Model

1.3.2 Accounts

Users work with FT through accounts. Each account has its own balance, history and it specifies authorization necessary to transfer tokens.

Account descriptor (*account_desc*) is a byte array which describes the account:

1. account type (simple, multi-sig, . . .)
2. public key(s) associated with the account
3. other parameters

Account ID is a hash of account descriptor, which serves as a short ID used to reference the account.

Account needs to be registered in the system before it can used.

FT can work in two modes:

- permissionless, where anybody can register an account
- permissioned, where only specially designated party can register an account

The later mode creates an opportunity to introduce identity checks, KYC/AML compliance, etc.

1.3.3 Assets and issuers

FT supports multiple different assets, such as currencies, to exist within the system. (Currently a list of assets is specified in the configuration at the time blockchain in created.)

Asset identifier is a string. For example, "USD". If there is a need to create different non-fungible USD tokens, we can just give them longer names, e.g.:

- "USD_LHV" are USD tokens created by LHV bank
- "USD_Citi" are USD tokens created by Citi bank

Or you can associate numeric codes, e.g. "001" will be USD from LHV bank.

Each asset has a list of issuers. Issuers can create new tokens. Each issuer has a special account which tracks token issuance, that account value will be negative. Thus total amount of tokens in the system is always zero. Tokens can be sent back to issuer's account to destroy/redeem them.

Amounts is specified in "atoms", i.e. smallest units. For example, for US dollar the smallest unit is cent, thus all amounts are in cents.

1.3.4 GTX

FT module uses GTX – *generalized transaction format*. A GTX transaction includes a list of operations to perform. Each operation is specified by the name of the operation and alist of arguments e.g.:

```
register(xxx)
issue(issuerID, "USD", 1000, receiverID)
```

GTX is encoded using ASN.1 DER.

Tutorial

1.3.5 Initialize library

We will assume node.js environment. Library can also work in browser, but we currently do not provide browser bundle. Instead we recommend using browserify, webpack, etc.

When you create a FTClient instance, you need to pass it Postchain client API URL. (Alternatively, one can pass Postchain restClient)

```
const FTClient = require('ft-client');
const ftClient = new FTClient("http://localhost:7741");
```

If library is used to create and sign transaction offline, null can be passed instead.

1.3.6 Create user keys

ft-client currently lacks means to create cryptographic keys. FT uses secp256k1 digital signatures – same as Bitcoin and Ethereum – so existing libraries can be used.

Note: we use libsecp256k1 data format for public and private keys and signatures. Compact 64-byte signature format is used.

```
    const secp256k1 = require('secp256k1');
    const randomBytes
require('crypto').randomBytes;

    function makeKeyPair () {
        let privKey;
        do {
           privKey = randomBytes(32);
        } while (!secp256k1.privateKeyVerify(privKey));
        const pubKey = secp256k1.publicKeyCreate(privKey);
        return {pubKey, privKey};
    }
```

1.3.7 Create user account

To be able to use FT, user needs to create and register an account. Here's a typical process:

1. Create account descriptor from user's public key.
2. Create an account ID.

3. Pass the account descriptor to an authority which can register accounts, or self-register if FT is used in permissionless mode.
4. Somebody calls *register* operation.

E.g.

```
const user1 = makeKeyPair();
const user1.account_desc = ftClient.makeSimpleAccountDesc(user1.pubKey);
const user1.account_id = ftClient.makeAccountID(user1.account_desc);
const tx = ftClient.makeTransaction([user1.pubKey]);
tx.register(user1.account_desc);
tx.sign(user1.privKey, user1.pubKey);
tx.submitAndWaitConfirmation().then(...)
```

1.3.8 Creating transactions

Code above creates and submits a transaction which register a user. Let's consider it in more details:

1. First you need to call ftClient.makeTransaction and supply it a list of signers, i.e. public keys which will sign this transaction.
2. Then you call operations, e.g. *tx.register(...)* or *tx.send(...)*. One transaction can include multiple operations, e.g. you can register a user and issue some tokens to him in one transaction.
3. Then you sign a transaction by calling *tx.sign(privKey)*, optionally also passing public key.
4. After transaction is signed by all signers, it can be submitted. Usually you want to wait until it's processed, thus you call *tx.submitAndWaitConfirmation()*, which returns a promise.

1.3.9 Issuing tokens

```
tx.issue(issuerID, "USD", 100000, user1.account_id);
```

This would issue 1000.00 USD to user1. Transaction must be signed by USD issuer.

1.3.10 Sending tokens

```
tx.send({
    from: user1.account_id,
    to: user2.account_id,
    amount: 1000,
    assetID: USD
    })
```

This would send 10 USD from user1 to user2. Needs to be signed by user1.

send is a shorthand function. Low-level operation called *xfer* supports multiple inputs and outputs, as well as multiple currencies. E.g. the send operation above can also be specified as

```
tx.xfer([[user1.account_id, "USD", 1000]],
    [[user2.account_id, "USD", 1000]]);
```

You can atomically change dollars for euros this way:

```
tx.xfer([[user1.account_id, "USD", 1000],
    [user2.account_id, "EUR", 500]],
    [[user2.account_id, "USD", 1000],
    [user1.account_id, "EUR", 500]]);
```

This transaction must be signed by both user1 and user2.

1.3.11 Standalone signing

Sometimes you might want to sign a transaction on a different devices. To do this:

1. Get a digest for signing: *tx.getDigestForSigning()*
2. Sign digest using secp256k1.
3. Call *tx.addSignature(pubKey, signature)*

1.3.12 Query data

1. *ftClient.getBalance(account_id, asset_id)*, returns a promise for a number
2. *ftClient.getHistory(account_id, asset_id)*, returns a promise for a list of objects

Example:

```
[ { delta: '100000',
    tx_rid: '30b8029dc4da1a3dae2fcfb6355092db394ef4e48503907ac3d4bf35b0945ba1',
    op_index: 2 },
    { delta: '-5000',
    tx_rid: 'ee9d01bf81037cfaf0ce4f1c4f464db6c1cc111a74d7225c8f318e8b6cbd42cc',
    op_index: 0 } ]
```

1.4 System requirements

1.4.1 Postchain Node JS

Operating system: Linux

Software dependencies:

- Software packages required, for Ubuntu Linux: python libpq-dev make g++
- PostgeSQL 9.4 or later
- NodeJS 7 or later

1.4.2 Postchain Java

Operating system: Any operating system that can run the software dependencies. Linux is recommended.

Software dependencies:

- PostgeSQL 9.4 or later.
- Java 8 Runtime Environment. Either OpenJDK's JRE or Oracle's JRE.

1.4.3 Hardware requirements

Hardware requirements generally depend on the nature of the application.

For small-scale demos we recommend 2 GB RAM, two CPU core (or virtual CPUs in a virtualized environment) and at least 10 GB of disk space.

These are not the minimal requirements, however.

1.5 Application developer's guide

1.5.1 Implementing blockchain logic

In the most typical case, to implement a custom blockchain you only need to implement GTX operations and a module, which can then be plugged into the rest of the system.

This can be done in three steps:

1. Define the schema in an SQL file. It should contain defined tables and stored procedures needed by the application.
2. Define GTX operations by subclassing net.postchain.gtx.GTXOperation.
3. Define the GTX module which maps names to operations, by subclassing net.postchain.gtx.GTXModule interface, or by subclassing net.postchain.gtx.SimpleGTXModule.

- Optionally, if GTX module needs parameters, one can define GTX module factory (net.postchain.gtx.GTXModuleFactory).

GTXTestOp is an example of a simple GTX operation which runs an INSERT query with a user-provided string. It is included into GTXTestModule which includes schema in an inline string and also includes an example of a query implementation.

A more complex example can be found in the tutorial.

If it is necessary to use a custom transaction format, it can be done in following way:

1. Implement net.postchain.core.Transaction interface to define transaction serialization format and semantics.
2. Implement transaction factory (net.postchain.core.TransactionFactory), usually it just calls transaction constructor.
3. Implement BlockchhainConfiguration. Simplest way is to subclass net.postchain.BaseBlockchainConfiguration and override getTransactionFactory method.
4. Implement BlockchainConfigurationFactory.

Steps 2-4 are usually trivial.

1.6 Deployment guide

1.6.1 Running Postchain nodes

To set up Postchain network, the following steps are needed:

1. Prepare jar files which implement blockchain application.
2. **Define the common node configuration, which includes the following:**

(a) blockchain identifier (blockchainrid)

(b) blockchain configuration and module classes

(c) module parameters (if needed)

(d) public keys used by nodes

(e) node network addresses

(f) **Define per-node configuration which includes**

i. database URL (host, username, password), schema

ii. client API port

iii. private key

(g) Configure database (usually creating database and user is enough)

(h) Run Postchain nodes with given configuration on each machine

Example of common configuration

Note that this example includes common.properties.

At least 4 nodes are needed for a fault-tolerant configuration, but it's possible to run network with just 3 nodes.

For development purposes it might be convenient to run Postchain in a single-node mode, in this case it can be configured like this

Postchain node can be launched like this:

```
java -jar postchain.jar -i 2 -c node2.properties
```

Parameter -i specifies node index starting from 0, parameter -c specifies node's configuration file.

Here's an example of module parameters used for FT module:

```
blockchain.1.gtx.modules=net.postchain.modules.ft.BaseFTModuleFactory
blockchain.1.gtx.ft.assets=USD
blockchain.1.gtx.ft.asset.USD.
↪issuers=03f811d3e806e6d093a4bcce49c145ba78f9a4b2fbd167753ecab2a13530b081f8
blockchain.1.gtx.ft.openRegistration=true
```

1.7 Search

- search