BigchainDB Documentation

Release

BigchainDB Contributors

May 17, 2017

Contents

BigchainDB is a scalable blockchain database. That is, it's a "big data" database with some blockchain characteristics added, including decentralization, immutability and native support for assets. You can read about the motivations, goals and high-level architecture in the BigchainDB whitepaper.

At a high level, one can communicate with a BigchainDB cluster (set of nodes) using the BigchainDB Client-Server HTTP API, or a wrapper for that API, such as the BigchainDB Python Driver. Each BigchainDB node runs BigchainDB Server and various other software. The terminology page explains some of those terms in more detail.

CHAPTER 1

More About BigchainDB

Production-Ready?

BigchainDB is not production-ready. You can use it to build a prototype or proof-of-concept (POC); many people are already doing that.

BigchainDB Server is currently in version 0.X. (The Releases page on GitHub has the exact version number.) Once BigchainDB Server is production-ready, we'll issue an announcement.

The BigchainDB Roadmap will give you a sense of the things we intend to do with BigchainDB in the near term and the long term.

Terminology

There is some specialized terminology associated with BigchainDB. To get started, you should at least know the following:

BigchainDB Node

A **BigchainDB node** is a machine or set of closely-linked machines running RethinkDB/MongoDB Server, BigchainDB Server, and related software. Each node is controlled by one person or organization.

BigchainDB Cluster

A set of BigchainDB nodes can connect to each other to form a **BigchainDB cluster**. Each node in the cluster runs the same software. A cluster contains one logical RethinkDB/MongoDB datastore. A cluster may have additional machines to do things such as cluster monitoring.

BigchainDB Consortium

The people and organizations that run the nodes in a cluster belong to a **BigchainDB consortium** (i.e. another organization). A consortium must have some sort of governance structure to make decisions. If a cluster is run by a single company, then the "consortium" is just that company.

What's the Difference Between a Cluster and a Consortium?

A cluster is just a bunch of connected nodes. A consortium is an organization which has a cluster, and where each node in the cluster has a different operator.

How BigchainDB is Decentralized

Decentralization means that no one owns or controls everything, and there is no single point of failure.

Ideally, each node in a BigchainDB cluster is owned and controlled by a different person or organization. Even if the cluster lives within one organization, it's still preferable to have each node controlled by a different person or subdivision.

We use the phrase "BigchainDB consortium" (or just "consortium") to refer to the set of people and/or organizations who run the nodes of a BigchainDB cluster. A consortium requires some form of governance to make decisions such as membership and policies. The exact details of the governance process are determined by each consortium, but it can be very decentralized (e.g. purely vote-based, where each node gets a vote, and there are no special leadership roles).

If sharding is turned on (i.e. if the number of shards is larger than one), then the actual data is decentralized in that no one node stores all the data.

Every node has its own locally-stored list of the public keys of other consortium members: the so-called keyring. There's no centrally-stored or centrally-shared keyring.

A consortium can increase its decentralization (and its resilience) by increasing its jurisdictional diversity, geographic diversity, and other kinds of diversity. This idea is expanded upon in the section on node diversity.

There's no node that has a long-term special position in the cluster. All nodes run the same software and perform the same duties.

RethinkDB and MongoDB have an "admin" user which can't be deleted and which can make big changes to the database, such as dropping a table. Right now, that's a big security vulnerability, but we have plans to mitigate it by:

1. Locking down the admin user as much as possible.
2. Having all nodes inspect admin-type requests before acting on them. Requests can be checked against an evolving whitelist of allowed actions. Nodes requesing non-allowed requests can be removed from the list of cluster nodes.

It's worth noting that the RethinkDB admin user can't transfer assets, even today. The only way to create a valid transfer transaction is to fulfill the current (crypto) conditions on the asset, and the admin user can't do that because the admin user doesn't have the necessary private keys (or preimages, in the case of hashlock conditions). They're not stored in the database.

Kinds of Node Diversity

Steps should be taken to make it difficult for any one actor or event to control or damage "enough" of the nodes. ("Enough" is usually a quorum.) There are many kinds of diversity to consider, listed below. It may be quite difficult to have high diversity of all kinds.

1. **Jurisdictional diversity.** The nodes should be controlled by entities within multiple legal jurisdictions, so that it becomes difficult to use legal means to compel enough of them to do something.
2. **Geographic diversity.** The servers should be physically located at multiple geographic locations, so that it becomes difficult for a natural disaster (such as a flood or earthquake) to damage enough of them to cause problems.
3. **Hosting diversity.** The servers should be hosted by multiple hosting providers (e.g. Amazon Web Services, Microsoft Azure, Digital Ocean, Rackspace), so that it becomes difficult for one hosting provider to influence enough of the nodes.
4. **Operating system diversity.** The servers should use a variety of operating systems, so that a security bug in one OS can't be used to exploit enough of the nodes.
5. **Diversity in general.** In general, membership diversity (of all kinds) confers many advantages on a consortium. For example, it provides the consortium with a source of various ideas for addressing challenges.

Note: If all the nodes are running the same code, i.e. the same implementation of BigchainDB, then a bug in that code could be used to compromise all of the nodes. Ideally, there would be several different, well-maintained implementations of BigchainDB Server (e.g. one in Python, one in Go, etc.), so that a consortium could also have a diversity of server implementations.

How BigchainDB is Immutable / Tamper-Resistant

The word *immutable* means "unchanging over time or unable to be changed." For example, the decimal digits of π are immutable (3.14159...).

The blockchain community often describes blockchains as "immutable." If we interpret that word literally, it means that blockchain data is unchangeable or permanent, which is absurd. The data *can* be changed. For example, a plague might drive humanity extinct; the data would then get corrupted over time due to water damage, thermal noise, and the general increase of entropy. In the case of Bitcoin, nothing so drastic is required: a 51% attack will suffice.

It's true that blockchain data is more difficult to change than usual: it's more tamper-resistant than a typical file system or database. Therefore, in the context of blockchains, we interpret the word "immutable" to mean tamper-resistant. (Linguists would say that the word "immutable" is a *term of art* in the blockchain community.)

BigchainDB achieves strong tamper-resistance in the following ways:

1. **Replication.** All data is sharded and shards are replicated in several (different) places. The replication factor can be set by the consortium. The higher the replication factor, the more difficult it becomes to change or delete all replicas.
2. **Internal watchdogs.** All nodes monitor all changes and if some unallowed change happens, then appropriate action is taken. For example, if a valid block is deleted, then it is put back.
3. **External watchdogs.** A consortium may opt to have trusted third-parties to monitor and audit their data, looking for irregularities. For a consortium with publicly-readable data, the public can act as an auditor.
4. **Cryptographic signatures** are used throughout BigchainDB as a way to check if messages (transactions, blocks and votes) have been tampered with enroute, and as a way to verify who signed the messages. Each block is signed by the node that created it. Each vote is signed by the node that cast it. A creation transaction is signed by the node that created it, although there are plans to improve that by adding signatures from the sending client and multiple nodes; see Issue #347. Transfer transactions can contain multiple inputs (fulfillments, one per asset transferred). Each fulfillment will typically contain one or more signatures from the owners (i.e. the owners before the transfer). Hashlock fulfillments are an exception; there's an open issue (#339) to address that.
5. **Full or partial backups** of the database may be recorded from time to time, possibly on magnetic tape storage, other blockchains, printouts, etc.

6. **Strong security.** Node owners can adopt and enforce strong security policies.
7. **Node diversity.** Diversity makes it so that no one thing (e.g. natural disaster or operating system bug) can compromise enough of the nodes. See the section on the kinds of node diversity.

Some of these things come "for free" as part of the BigchainDB software, and others require some extra effort from the consortium and node owners.

BigchainDB and Byzantine Fault Tolerance

We have Byzantine fault tolerance (BFT) in our roadmap, as a switch that people can turn on. We anticipate that turning it on will cause a severe dropoff in performance (to gain some extra security). See Issue #293.

Among the big, industry-used distributed databases in production today (e.g. DynamoDB, Bigtable, MongoDB, Cassandra, Elasticsearch), none of them are BFT. Indeed, almost all wide-area distributed systems in production are not BFT, including military, banking, healthcare, and other security-sensitive systems.

The are many more practical things that nodes can do to increase security (e.g. firewalls, key management, access controls).

From a recent essay by Ken Birman (of Cornell):

> Oh, and with respect to the BFT point: Jim [Gray] felt that real systems fail by crashing [54]. Others have since done studies reinforcing this view, or finding that even crash-failure solutions can sometimes defend against application corruption. One interesting study, reported during a SOSP WIPS session by Ben Reed (one of the co-developers of Zookeeper), found that at Yahoo, Zookeeper itself had never experienced Byzantine faults in a one-year period that they studied closely.
>
> [54] Jim Gray. Why Do Computers Stop and What Can Be Done About It? SOSP, 1985.

Ben Reed never published those results, but Birman wrote more about them in the book *Guide to Reliable Distributed Systems: Building High-Assurance Applications*. From page 358 of that book:

> But the cloud community, led by Ben Reed and Flavio Junqueira at Yahoo, sees things differently (these are the two inventor's [sic] of Yahoo's ZooKeeper service). **They have described informal studies of how applications and machines at Yahoo failed, concluding that the frequency of Byzantine failures was extremely small relative to the frequency of crash failures** [emphasis added]. Sometimes they did see data corruption, but then they often saw it occur in a correlated way that impacted many replicas all at once. And very often they saw failures occur in the client layer, then propagate into the service. BFT techniques tend to be used only within a service, not in the client layer that talks to that service, hence offer no protection against malfunctioning clients. **All of this, Reed and Junqueira conclude, lead to the realization that BFT just does not match the real needs of a cloud computing company like Yahoo, even if the data being managed by a service really is of very high importance** [emphasis added]. Unfortunately, they have not published this study; it was reported at an "outrageous opinions" session at the ACM Symposium on Operating Systems Principles, in 2009.
>
> The practical use of the Byzantine protocol raises another concern: The timing assumptions built into the model [i.e. synchronous or partially-synchronous nodes] are not realizable in most computing environments...

How BigchainDB is Good for Asset Registrations & Transfers

BigchainDB can store data of any kind (within reason), but it's designed to be particularly good for storing asset registrations and transfers:

- The fundamental thing that one sends to a BigchainDB cluster, to be checked and stored (if valid), is a *transaction*, and there are two kinds: CREATE transactions and TRANSFER transactions.
- A CREATE transaction can be use to register any kind of asset (divisible or indivisible), along with arbitrary metadata.
- An asset can have zero, one, or several owners.
- The owners of an asset can specify (crypto-)conditions which must be satisified by anyone wishing transfer the asset to new owners. For example, a condition might be that at least 3 of the 5 current owners must cryptographically sign a transfer transaction.
- BigchainDB verifies that the conditions have been satisified as part of checking the validity of transfer transactions. (Moreover, anyone can check that they were satisfied.)
- BigchainDB prevents double-spending of an asset.
- Validated transactions are strongly tamper-resistant; see [the section about immutability / tamper-resistance](immutable.html).

BigchainDB Integration with Other Blockchains

BigchainDB works with the Interledger protocol, enabling the transfer of assets between BigchainDB and other blockchains, ledgers, and payment systems.

We're actively exploring ways that BigchainDB can be used with other blockchains and platforms.

Note: We used the word "owners" somewhat loosely above. A more accurate word might be fulfillers, signers, controllers, or tranfer-enablers. See BigchainDB Server issue #626.

BigchainDB and Smart Contracts

One can store the source code of any smart contract (i.e. a computer program) in BigchainDB, but BigchainDB won't run arbitrary smart contracts.

BigchainDB will run the subset of smart contracts expressible using "crypto-conditions," a subset we like to call "simple contracts." Crypto-conditions are part of the Interledger Protocol.

The owners of an asset can impose conditions on it that must be met for the asset to be transferred to new owners. Examples of possible conditions (crypto-conditions) include:

- The current owner must sign the transfer transaction (one which transfers ownership to new owners)
- Three out of five current owners must sign the transfer transaction
- (Shannon and Kelly) or Morgan must sign the transfer transaction
- Anyone who provides the secret password (technically, the preimage of a known hash) can create a valid transfer transaction

Crypto-conditions can be quite complex if-this-then-that type conditions, where the "this" can be a long boolean expression. Crypto-conditions can't include loops or recursion and are therefore will always run/check in finite time.

BigchainDB also supports a timeout condition which enables it to support a form of escrow.

Note: We used the word "owners" somewhat loosely above. A more accurate word might be fulfillers, signers, controllers, or tranfer-enablers. See BigchainDB Server issue #626.

Transaction Concepts

In BigchainDB, *Transactions* are used to register, issue, create or transfer things (e.g. assets).

Transactions are the most basic kind of record stored by BigchainDB. There are two kinds: CREATE transactions and TRANSFER transactions.

CREATE Transactions

A CREATE transaction can be used to register, issue, create or otherwise initiate the history of a single thing (or asset) in BigchainDB. For example, one might register an identity or a creative work. The things are often called "assets" but they might not be literal assets.

BigchainDB supports divisible assets as of BigchainDB Server v0.8.0. That means you can create/register an asset with an initial quantity, e.g. 700 oak trees. Divisible assets can be split apart or recombined by transfer transactions (described more below).

A CREATE transaction also establishes, in its outputs, the conditions that must be met to transfer the asset(s). The conditions may also be associated with a list of public keys that, depending on the condition, may have full or partial control over the asset(s). For example, there may be a condition that any transfer must be signed (cryptographically) by the private key associated with a given public key. More sophisticated conditions are possible. BigchainDB's conditions are based on the crypto-conditions of the Interledger Protocol (ILP).

TRANSFER Transactions

A TRANSFER transaction can transfer an asset by providing inputs which fulfill the current output conditions on the asset. It must also specify new transfer conditions.

Example 1: Suppose a red car is owned and controlled by Joe. Suppose the current transfer condition on the car says that any valid transfer must be signed by Joe. Joe and a buyer named Rae could build a TRANSFER transaction containing an input with Joe's signature (to fulfill the current output condition) plus a new output condition saying that any valid transfer must be signed by Rae.

Example 2: Someone might construct a TRANSFER transaction that fulfills the output conditions on four previously-untransferred assets of the same asset type e.g. paperclips. The amounts might be 20, 10, 45 and 25, say, for a total of 100 paperclips. The TRANSFER transaction would also set up new transfer conditions. For example, maybe a set of 60 paperclips can only be transferred if Gertrude signs, and a separate set of 40 paperclips can only be transferred if both Jack and Kelly sign. Note how the sum of the incoming paperclips must equal the sum of the outgoing paperclips (100).

Transaction Validity

When a node is asked to check if a transaction is valid, it checks several things. Some things it checks are:

- Are all the fulfillments valid? (Do they correctly satisfy the conditions they claim to satisfy?)
- If it's a creation transaction, is the asset valid?
- If it's a transfer transaction:

- Is it trying to fulfill a condition in a nonexistent transaction?
- Is it trying to fulfill a condition that's not in a valid transaction? (It's okay if the condition is in a transaction in an invalid block; those transactions are ignored. Transactions in the backlog or undecided blocks are not ignored.)
- Is it trying to fulfill a condition that has already been fulfilled, or that some other pending transaction (in the backlog or an undecided block) also aims to fulfill?
- Is the asset ID in the transaction the same as the asset ID in all transactions whose conditions are being fulfilled?
- Is the sum of the amounts in the fulfillments equal to the sum of the amounts in the new conditions?

If you're curious about the details of transaction validation, the code is in the `validate` method of the `Transaction` class, in `bigchaindb/models.py` (at the time of writing).

Note: The check to see if the transaction ID is equal to the hash of the transaction body is actually done whenever the transaction is converted from a Python dict to a Transaction object, which must be done before the `validate` method can be called (since it's called on a Transaction object).

Timestamps in BigchainDB

Each block and vote has an associated timestamp. Interpreting those timestamps is tricky, hence the need for this section.

Timestamp Sources & Accuracy

Timestamps in BigchainDB are provided by the node which created the block and the node that created the vote.

When a BigchainDB node needs a timestamp, it calls a BigchainDB utility function named `timestamp()`. There's a detailed explanation of how that function works below, but the short version is that it gets the Unix time from its system clock, rounded to the nearest second.

We advise BigchainDB nodes to run special software (an "NTP daemon") to keep their system clock in sync with standard time servers. (NTP stands for Network Time Protocol.)

Converting Timestamps to UTC

To convert a BigchainDB timestamp (a Unix time) to UTC, you need to know how the node providing the timestamp was set up. That's because different setups will report a different "Unix time" value around leap seconds! There's a nice Red Hat Developer Blog post about the various setup options. If you want more details, see David Mills' pages about leap seconds, NTP, etc. (David Mills designed NTP.)

We advise BigchainDB nodes to run an NTP daemon with particular settings so that their timestamps are consistent.

If a timestamp comes from a node that's set up as we advise, it can be converted to UTC as follows:

1. Use a standard "Unix time to UTC" converter to get a UTC timestamp.
2. Is the UTC timestamp a leap second, or the second before/after a leap second? There's a list of all the leap seconds on Wikipedia.
3. If no, then you are done.

4. If yes, then it might not be possible to convert it to a single UTC timestamp. Even if it can't be converted to a single UTC timestamp, it *can* be converted to a list of two possible UTC timestamps. Showing how to do that is beyond the scope of this documentation. In all likelihood, you will never have to worry about leap seconds because they are very rare. (There were only 26 between 1972 and the end of 2015.)

Calculating Elapsed Time Between Two Timestamps

There's another gotcha with (Unix time) timestamps: you can't calculate the real-world elapsed time between two timestamps (correctly) by subtracting the smaller timestamp from the larger one. The result won't include any of the leap seconds that occured between the two timestamps. You could look up how many leap seconds happened between the two timestamps and add that to the result. There are many library functions for working with timestamps; those are beyond the scope of this documentation.

Interpreting Sets of Timestamps

You can look at many timestamps to get a statistical sense of when something happened. For example, a transaction in a decided-valid block has many associated timestamps:

- the timestamp of the block
- the timestamps of all the votes on the block

How BigchainDB Uses Timestamps

BigchainDB *doesn't* use timestamps to determine the order of transactions or blocks. In particular, the order of blocks is determined by RethinkDB's changefeed on the bigchain table.

BigchainDB does use timestamps for some things. When a Transaction is written to the backlog, a timestamp is assigned called the `assignment_timestamp`, to determine if it has been waiting in the backlog for too long (i.e. because the node assigned to it hasn't handled it yet).

Including Trusted Timestamps

If you want to create a transaction payload with a trusted timestamp, you can.

One way to do that would be to send a payload to a trusted timestamping service. They will send back a timestamp, a signature, and their public key. They should also explain how you can verify the signature. You can then include the original payload, the timestamp, the signature, and the service's public key in your transaction metadata. That way, anyone with the verification instructions can verify that the original payload was signed by the trusted timestamping service.

How the timestamp() Function Works

BigchainDB has a utility function named `timestamp()` which amounts to:

```
timestamp() = str(round(time.time()))
```

In other words, it calls the `time()` function in Python's `time` module, rounds that to the nearest integer, and converts the result to a string.

It rounds the output of `time.time()` to the nearest second because, according to the Python documentation for `time.time()`, "...not all systems provide time with a better precision than 1 second."

How does `time.time()` work? If you look in the C source code, it calls `floattime()` and `floattime()` calls clock_gettime(), if it's available.

```
ret = clock_gettime(CLOCK_REALTIME, &tp);
```

With `CLOCK_REALTIME` as the first argument, it returns the "Unix time." ("Unix time" is in quotes because its value around leap seconds depends on how the system is set up; see above.)

Why Not Use UTC, TAI or Some Other Time that Has Unambiguous Timestamps for Leap Seconds?

It would be nice to use UTC or TAI timestamps, but unfortunately there's no commonly-available, standard way to get always-accurate UTC or TAI timestamps from the operating system on typical computers today (i.e. accurate around leap seconds).

There *are* commonly-available, standard ways to get the "Unix time," such as clock_gettime() function available in C. That's what we use (indirectly via Python). ("Unix time" is in quotes because its value around leap seconds depends on how the system is set up; see above.)

The Unix-time-based timestamps we use are only ambiguous circa leap seconds, and those are very rare. Even for those timestamps, the extra uncertainty is only one second, and that's not bad considering that we only report timestamps to a precision of one second in the first place. All other timestamps can be converted to UTC with no ambiguity.