Byteball: A Decentralized System for Storage and Transfer of Value

Anton Churyumov
tonych@byteball.org

Abstract

Byteball is a decentralized system that allows tamper proof storage of arbitrary data, including data that represents transferrable value such as currencies, property titles, debt, shares, etc. Storage units are linked to each other such that each storage unit includes one or more hashes of earlier storage units, which serves both to confirm earlier units and establish their partial order. The set of links among units forms a DAG (directed acyclic graph). There is no single central entity that manages or coordinates admission of new units into the database, everyone is allowed to add a new unit provided that he signs it and pays a fee equal to the size of added data in bytes. The fee is collected by other users who later confirm the newly added unit by including its hash within their own units. As new units are added, each earlier unit receives more and more confirmations by later units that include its hash, directly or indirectly.

There is an internal currency called 'bytes' that is used to pay for adding data into the decentralized database. Other currencies (assets) can also be freely issued by anyone to represent property rights, debt, shares, etc. Users can send both bytes and other currencies to each other to pay for goods/services or to exchange one currency for another; the transactions that move the value are added to the database as storage units. If two transactions try to spend the same output (double-spend) and there is no partial order between them, both are allowed into the database but only the one that comes earlier in the total order is deemed valid. Total order is established by selecting a single chain on the DAG (the main chain) that is attracted to units signed by known users called witnesses. A unit whose hash is included earlier on the main chain is deemed earlier on the total order. Users choose the witnesses by naming the user-trusted witnesses in every storage unit. Witnesses are reputable users with real-world identities, and users who name them expect them to never try to double-spend. As long as the majority of witnesses behave as expected, all double-spend attempts are detected in time and marked as such. As witnesses-authored units accumulate after a user's unit, there are deterministic (not probabilistic) criteria when the total order of the user's unit is considered final.

Users store their funds on addresses that may require more than one signature to spend (multisig). Spending may also require other conditions to be met, including conditions that are evaluated by looking for specific data posted to the database by other users (oracles).

Users can issue new assets and define rules that govern their transferability. The rules can include spending restrictions such as a requirement for each transfer to be cosigned by the issuer of the asset, which is one way for financial institutions to comply with existing regulations. Users can also issue assets whose transfers are not published to the database, and therefore not visible to third parties. Instead, the information about the transfer is exchanged privately between users, and only a hash of the transaction and a spend proof (to prevent double-spends) are published to the database.

1. Introduction

In Orwell's *1984*, the protagonist Winston Smith works in the Records Department of the Ministry of Truth as an editor, revising historical records, to make the past conform to the ever-changing party line and deleting references to unpersons – people who have been "vaporised," i.e. not only killed by the state but denied existence even in history or memory [1]. What we present here is data storage that is not rewritable. It is a distributed decentralized database where records can neither be revised nor deleted entirely.

Bitcoin [2] was the first system to introduce tamper proof records designed for the specific purpose of tracking the ownership of electronic currency units known as bitcoins. In Bitcoin, all transfers of the currency are represented as transactions that are digitally signed by the current owner of the coin, transactions are bundled into blocks, and blocks are linked into a chain (blockchain) secured by proof of work (PoW) that assures that large computing resources have been invested into building the chain. Any attempt to rewrite anything contained in the chain would therefore require even larger computing resources than those that have already been expended.

Soon after Bitcoin appeared, it became clear that this was more than just a trust-free P2P electronic currency. Its technology became a source of new ideas for solving other problems. At the same time, Bitcoin's deficiencies and limitations equally became clear. Byteball is designed to generalize Bitcoin to become a tamper proof storage of any data, not solely transfers of a single electronic currency, and remove some of the most pressing deficiencies that impede a wider adoption and growth of Bitcoin.

Blocks. In Bitcoin, transactions are bundled into blocks, and blocks are linked into a single chain. Since the blocks are linked linearly, their spacing in time and their size are optimized for near-synchrony among nodes, so that the nodes can share a new block with each other much faster than it typically takes to generate a new block. This ensures that nodes most likely see the same block as the last block, and orphaning is minimized. As Bitcoin grows, blocks become increasingly unwieldy. They are either capped in size, in which case the growth is also capped, or they take too long to propagate to all nodes of the network, in which case there is greater uncertainty about which block is last, and more resources are wasted on extending chains that would later be orphaned. In Byteball, there are no blocks, transactions are their own blocks, and they need not connect into a single chain. Instead a transaction may be linked to multiple previous transactions, and the whole set of transactions is not a chain but a DAG (directed acyclic graph). DAG-based designs have received much attention recently [3-5].

Cost. Bitcoin transactions are secure because it is prohibitively expensive to redo all the PoW included in the blocks created after the transaction. But that also means that it is necessary to pay to build the legitimate PoW that is strong enough to ward off any attackers. This payment is spent for the electricity required to build the PoW. What is important to note here, is that this money goes outside the Bitcoin ecosystem – to energy companies – meaning that the community of Bitcoin holders as a whole is bleeding capital. In Byteball, there is no PoW, instead we use

another consensus algorithm based on an old idea that was known long before Bitcoin.

Finality. Transaction finality in Bitcoin is probabilistic. There are no strict and simple criteria for when you can say that a transaction will never be reversed. Rather, you can only argue that the probability of a transaction being reversed exponentially decays as more blocks are added. While this concept is perfectly clear to those versed in math, it might be a difficult sell to an average Joe who is used to expecting a black-or-white picture in matters of money ownership. To complicate things even further, transaction finality also depends on its amount. If the amount is small, you can be reasonably sure nobody will try to double-spend against you. However, if the amount at stake is greater than the block reward (12.5 BTC at the time of writing), you might speculate that the payer could temporarily rent hashpower to mine another chain of blocks that doesn't contain the transaction that pays to you. Therefore, you have to wait for more confirmations before being sure enough that a high-value transaction is final. In Byteball, there are deterministic criteria for when a transaction is deemed final, no matter how large it was.

Exchange rate. The Bitcoin price is known to be quite volatile. The bigger problem is that this price is not only volatile, it is not bound to anything. Share and commodity prices are also very volatile but there are fundamentals behind them. Share price is largely a function of company earnings, revenue, debt-to-capital ratio, etc. Commodity prices depend, among other factors, on costs of production with various suppliers. For example, if the oil price falls below the production costs of some suppliers for a long time, these suppliers will eventually shut down, decreasing production and causing the price to go up. There is a negative feedback loop. In Bitcoin, there are no fundamentals, and no negative feedback. A Bitcoin price of $500 is no more justified than a price of $50,000 or $5. If the Bitcoin price moves from where it is now, this movement alone will not create any economic forces that would push the price back. It's just wild. In Byteball, the base currency, bytes, is used to pay for adding data into the Byteball database. You pay 1,000 bytes to add 1Kb of data. It is a measure of the utility of the storage in this database, and actual users will have their opinion on what is a reasonable price for this. If the price of byte rises above what you think is reasonable for your needs, you will find ways to store less bytes, therefore you need to buy less bytes, demand decreases, and the price falls. This is negative feedback, common for all goods/services whose demand is driven by need, not speculation. Besides paying in bytes, one can issue other assets and use them as means of payment. These assets might represent, for example, debt expressed in fiat currencies or in natural units (such as kWh or barrels of oil). The price of such assets is naturally bound to the underlying currencies or commodities.

Privacy. All Bitcoin transactions and balances of all addresses are visible on the blockchain. Although there are ways to obfuscate one's transactions and balances, it is not what people have come to expect from a currency. Transactions in bytes (the base currency) in Byteball are equally visible, but there is a second currency (blackbytes), which is significantly less traceable.

Compliance. Bitcoin was designed as an anonymous currency where people have absolute control over their money. That goal was achieved; however, it made

Bitcoin incompatible with existing regulations, and hence inappropriate for use in the financial industry. In Byteball, one can issue assets with any rules that govern their transferability, from no restrictions at all, like Bitcoin, to anything like requiring every transfer to be cosigned by the issuer (e.g. the bank) or restricted to a limited set of whitelisted users.

2. Database structure

When a user wants to add data to the database, he creates a new storage unit and broadcasts it to his peers. The storage unit includes (among other things):

- The data to be stored. A unit may include more than one data package called a *message*. There are many different types of messages, each with its own structure. One of the message types is *payment*, which is used to send bytes or other assets to peers.
- Signature(s) of one or more users who created the unit. Users are identified by their addresses. Individual users may (and are encouraged to) have multiple addresses, like in Bitcoin. In the simplest case, the address is derived from a public key, again similar to Bitcoin.
- References to one or more previous units (parents) identified by their hashes.

References to parents is what establishes the order (only partial order so far) of units and generalizes the blockchain structure. Since we are not confined to one-parent–one-child relationships between consecutive blocks, we do not have to strive for near-synchrony and can safely tolerate large latencies and high throughputs: we'll just have more parents per unit and more children per unit. If we go forward in history along parent-child links, we'll observe many forks when the same unit is referenced by multiple later units, and many merges when the same unit references multiple earlier units (developers are already used to seeing this in git). This structure is known in graph theory as directed acyclic graph (DAG). Units are vertices, and parent-child links are the edges of the graph.

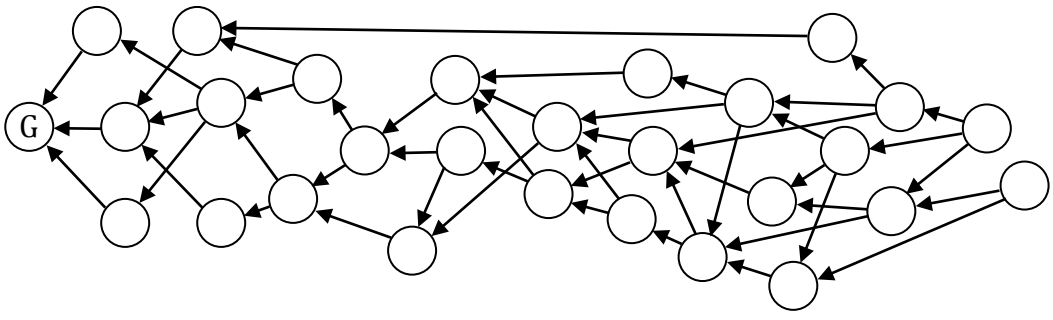

Figure 1. Storage units connected into a DAG. Arrows are from child to parent, G is the genesis unit.

In the special case when new units arrive rarely, the DAG will look almost like a chain, with only occasional forks and quick merges.

Like in blockchains where each new block confirms all previous blocks (and transactions therein), every new child unit in the DAG confirms its parents, all parents of parents, parents of parents of parents, etc. If one tries to edit a unit, he will also have to change its hash. Inevitably, this would break all child units who reference this unit by its hash as both signatures and hashes of children depend on parent hashes. Therefore, it is impossible to revise a unit without cooperating with all its children or stealing their private keys. The children, in turn, cannot revise their units without cooperating with their children (grandchildren of the original unit), and so on. Once a unit is broadcast into the network, and other users start building their units on top of it (referencing it as parent), the number of secondary revisions required to edit this unit hence grows like a snowball. That's why we call this design Byteball (our snowflakes are bytes of data).

Unlike blockchains where issuing a block is a rare event and only a privileged caste of users is in practice engaged in this activity, in a new Byteball unit starts accumulating confirmations immediately after it is released and confirmations can come from anyone, every time another new unit is issued. There is no two-tier system of ordinary users and miners. Instead, users help each other: by adding a new unit its author also confirms all previous units.

Unlike Bitcoin, where an attempt to revise a past transaction requires a large computational effort, an attempt to revise a past record in Byteball requires coordination with a large and growing number of other users, most of whom are anonymous strangers. The immutability of past records is therefore based on the sheer complexity of coordinating with such a large number of strangers, who are difficult to reach, have no interest in cooperation, and where every single one of them can veto the revision.

By referencing its parents, a unit includes the parent. It doesn't include the full content of the parent; rather, it depends on its information through the parent's hash. In the same way, the unit indirectly depends on and therefore includes the parents of the parent, their parents, and so on, and every unit ultimately includes the genesis unit.

There is a protocol rule that a unit cannot reference redundant parents – that is such parents that one parent includes another. For example, if unit B references unit A, then unit C cannot reference both units A and B at the same time. A is already, in a way, contained within B. This rule removes unnecessary links that don't add any new useful connectivity to the graph.

3. Native currency: bytes

Next, we need to introduce some friction to protect against spamming the database with useless messages. The barrier to entry should roughly reflect the utility of storage for the user and the cost of storage for the network. The simplest measure for both of these is the size of the storage unit. Thus, to store your data in the global decentralized database you have to pay a fee in internal currency called *bytes*, and the amount you pay is equal to the size of data you are going to store (including all headers, signatures, etc). Similar to pound sterling, which was equal to one pound of silver when it was first introduced, the name of the currency reflects its value.

To keep the incentives aligned with the interests of the network, there is one exception in size calculation rules. For the purposes of calculating unit size, it is assumed that the unit has exactly two parents, no matter the real number. Therefore, the size of two hashes of parent units is always included in the unit size. This exception ensures that users will not try to include just one parent in an effort to minimize cost. The cost is the same no matter how many parents are included.

To keep the DAG as narrow as possible, we incentivize users to include as many parents as possible (as mentioned before, this does not negatively affect payable size), and as recent parents as possible, by paying part of the unit's fees to those who are first to include it as a parent. We'll define later what exactly is 'first'.

Bytes can be used not only for payment of storage fees (also called commissions), but also can be sent to other users to pay for goods or services or in exchange for other assets. To send a payment, the user creates a new unit that includes a payment message such as the following (from now on, we use JSON to describe data structures):

```
{
	inputs: [
		{
			unit: "hash of input unit",
			message_index: 2, // index of message where this utxo
			was created
			output_index: 0 // index of output where this utxo
			was created
		},
		…
	],
	outputs: [
		{
			address: "RECEIVER ADDRESS",
			amount: 15000   // in bytes
		},
		…
	]
}
```

The message contains:

- An array of outputs: one or more addresses that receive the bytes and the amounts they receive.
- An array of inputs: one or more references to previous outputs that are used to fund the transfer. These are outputs that were sent to the author address(es) in the past and are not yet spent.

The sum of inputs should be equal to the sum of outputs plus commissions (input amounts are read from previous outputs and are not explicitly indicated when spending). The unit is signed with the author's private keys.

The total number of bytes in circulation is 10^{15}, and this number is constant. All bytes are issued in the genesis unit, then transferred from user to user. Fees are collected by other users who help to keep the network healthy (more details about that later), so they stay in circulation. The number 10^{15} was selected as the largest round integer that can be represented in JavaScript. Amounts can only be only integers. Larger units of the currency are derived by applying standard prefixes: 1 kilobyte (Kb) is 1,000 bytes, 1 megabyte (Mb) is 1 million bytes, etc.

4. Double-spends

If a user tries to spend the same output twice, there are two possible situations:

1. There is partial order between the two units that try to spend the same output, i.e. one of the units (directly or indirectly) includes the other unit, and therefore comes after it. In this case, it is obvious that we can safely reject the later unit.
2. There is no partial order between them. In this case, we accept both. We establish a total order between the units later on, when they are buried deep enough under newer units (see below how we do it). The one that appears earlier on the total order is deemed valid, while the other is deemed invalid.

There is one more protocol rule that simplifies the definition of total order. We require, that if the same address posts more than one unit, it should include (directly or indirectly) all its previous units in every subsequent unit, i.e. there should be partial order between consecutive units from the same address. In other words, all units from the same author should be serial.

If someone breaks this rule and posts two units such that there is no partial order between them (nonserial units), the two units are treated like double-spends even if they don't try to spend the same output. Such nonserials are handled as described in situation 2 above.

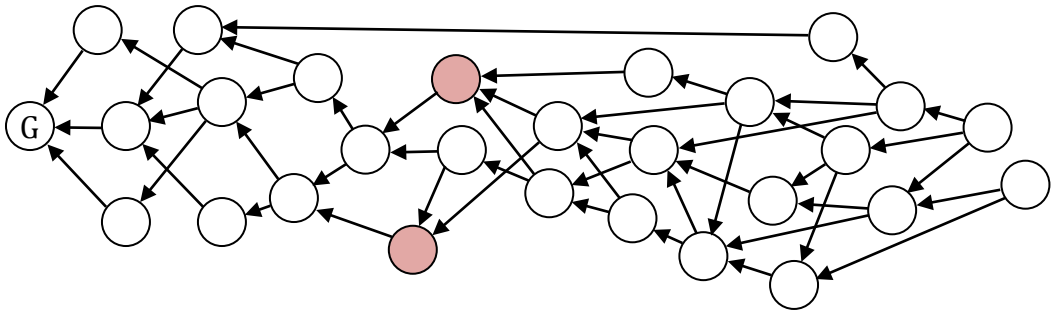

Figure 2. Double-spends. There is no partial order between them.

If a user follows this rule but still tries to spend the same output twice, the double-spends become unambiguously ordered and we can safely reject the later one as in situation 1 above. The double-spends that are not nonserials at the same time are hence easily filtered out.

This rule is in fact quite natural. When a user composes a new unit, he selects the most recent other units as parents of his unit. By putting them on his parents list, he declares his picture of the world, which implies that he has seen these units. He has therefore seen all parents of parents, parents of parents of parents, etc up until the genesis unit. This huge set should obviously include everything that he himself has produced, and therefore has seen.

By not including a unit (even indirectly, through parents) the user denies that he has seen it. If we see that by not including his own previous unit a user denies

having seen it, we'd say it's odd, something fishy is going on. We discourage such behavior.

5. The main chain

Our DAG is a special DAG. In normal use, people mostly link their new units to slightly less recent units, meaning that the DAG grows only in one direction. One can picture it as a thick cord with many interlaced wires inside. This property suggests that we could choose a single chain along child-parent links within the DAG, and then relate all units to this chain. All the units will either lie directly on this chain, which we'll call the *main chain*, or be reachable from it by a relatively small number of hops along the edges of the graph. It's like a highway with connecting side roads.

One way to build a main chain is to develop an algorithm that, given all parents of a unit, selects one of them as the "best parent". The selection algorithm should be based only on knowledge available to the unit in question, i.e. on data contained in the unit itself and all its ancestors. Starting from any tip (a childless unit) of the DAG, we then travel backwards in history along the best parent links. Traveling this way, we build a main chain and eventually arrive at the genesis unit. Note that the main chain built starting from a specific unit will never change as new units are added. This is because on each step we are traveling from child to parent, and an existing unit can never acquire new parents.

If we start from another tip, we'll build another main chain. Of note here is that if those two main chains ever intersect while they go back in history, they will both go along the same path after the intersection point. In the worst case, the main chains will intersect only in genesis. Given that the process of unit production is not coordinated among users, however, one might expect to find a class of main chains that do converge not too far from the tips.

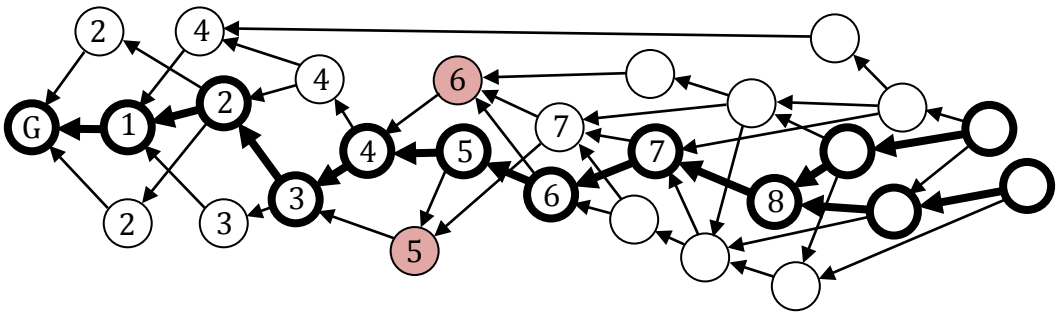

Figure 3. Main chains built from different childless units intersect and then go along the same path. Of the two double-spends, the one with the lower main chain index (5) wins, while the other (with MCI=6) is deemed invalid.

Once we have a main chain (MC), we can establish a total order between two conflicting nonserial units. Let's first index the units that lie directly on the main chain. The genesis unit has index 0, the next MC unit that is a child of genesis has

index 1, and so on traveling forward along the MC we assign indexes to units that lie on the MC. For units that do not lie on the MC, we can find an MC index where this unit is first included (directly or indirectly). In such a way, we can assign an MC index (MCI) to every unit.

Then, of the two nonserials, the one that has a lower MCI is considered to come earlier and deemed valid, while the other is invalid. If both nonserials happen to have the same MCI, there is tiebreaker rule that the unit with the lower hash value (as represented in base64 encoding) is valid. Note that we keep all versions of the double-spend, including those that eventually lose. DagCoin [3] was the first published work that suggested storing all conflicting transactions and deciding which one to treat as valid.

The MC built from a specific unit tells us what this unit's author thinks about the order of past events, i.e. his point of view about the history. The order then implies which nonserial unit to consider valid, as described above. Note that by choosing the best parent among all parents of a given unit, we are simultaneously making a choice among their MCs: the MC of the unit in question will be the MC of its best parent extended forward by one link.

Recognizing that many (or even all) parent units might be created by an attacker, and remembering that the choice of best parent is essentially the choice among versions of history, we should require from our best parent selection algorithm that it favors histories that are "real" from the point of view of the child unit. We hence need to devise a "reality test" that our algorithm would run against all candidate MCs to select the one that scores best.

6. Witnesses

Looking for a "reality test", observe that some of the participants of our network are non-anonymous reputable people or companies who might have a long established reputation, or they are businesses interested in keeping the network healthy. We'll call them *witnesses*. While it is reasonable to expect them to behave honestly, it is also unreasonable to totally trust any single witness. If we know the Byteball addresses of several witnesses, and also expect them to post frequently enough, then to measure the reality of a candidate MC one might travel along the MC back in time and count the witness-authored units (if the same witness is encountered more than once, he is not counted again). We would stop traveling as soon as we had encountered the majority of witnesses. We would then measure the length of the longest path on the graph from the point at which we stopped to the genesis. We'll call this length the *level* of the unit where we stopped, and the *witnessed level* of the parent whose MC we are testing. The candidate MC that yields the greater witnessed level is considered more "real", and the parent bearing this MC is selected as best parent. In case there are several contenders with a maximum witnessed level, we would select the parent whose own level is the lowest. If the tie persists, we would select the parent with the smallest unit hash (in base64 encoding).

This algorithm allows the selection of the MC that gravitates to units authored by witnesses, and the witnesses are considered to be representative of reality. If, for example, an attacker forks from the honest part of the network and

secretly builds a long chain of his own units (shadow chain), one of them containing a double-spend, and later merges his fork back into the honest DAG, the best parent selection algorithm at the merger point will choose the parent that drives the MC into the honest DAG, as this is where the witnesses were active. The witnesses were not able to post into the shadow chain simply because they didn't see it before the merger. This selection of MC reflects the order of events as seen by the witnesses and the user who appointed them. After the attack is over, the entire shadow chain will land on the MC at one point, and the double-spend

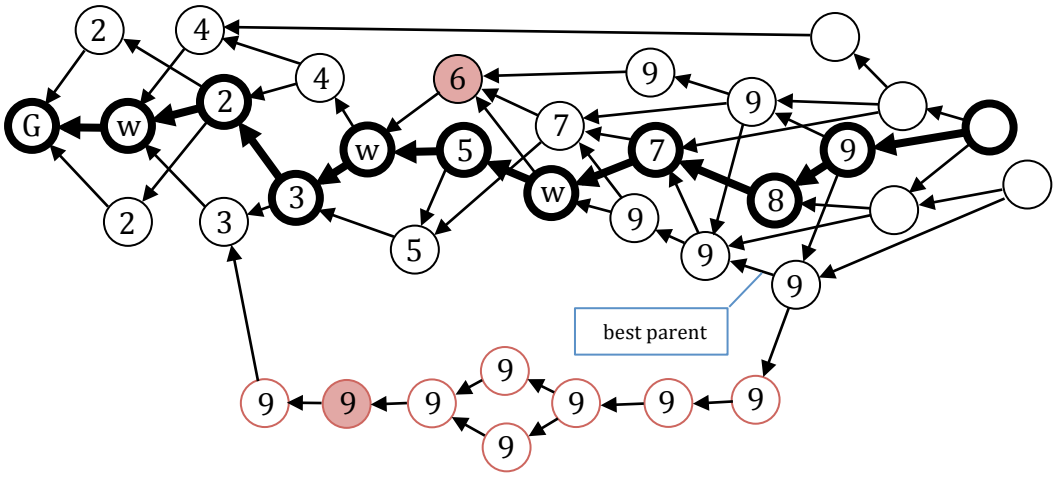

Figure 4. When an attacker rejoins his shadow DAG into the lit DAG, his units lose competition to become best parent as the choice favors those paths that have more witnesses (marked with w).

contained in the shadow chain will be deemed invalid because its valid counterpart comes earlier, before the merger point.

This example shows why the majority of witnesses has to be trusted to post only serially. The majority should not collude with the attacker and post on his shadow chain. Note that we trust the witnesses only to be signs of reality and to not post nonserial units on any shadow chains. We are not giving any of them control over the network or any part thereof. Even for this small duty, it is users who appoint the witnesses and they can change their decisions at any time.

The idea of looking at some known entity as a sign of reality is not new. It has long been known, and some companies have engaged in such activity, that to prove that some data existed before a specific date, one can hash the data and publish the hash in some hard-to-modify and widely witnessed media, like printed newspaper [6]. Witnesses in Byteball serve the same function as the newspaper. Like newspapers, they are well known and trusted. As for newspapers where trust is limited to trusting them to publish the data they are given, witnesses in Byteball are only trusted to post serially, and not much more. Like newspapers, witnesses don't know what's behind the hashes they are witnessing and have few reasons to

care. Newspapers are hard to modify (but possible, and in *1984* they do it), while everything produced by witnesses is protected by digital signatures, which makes any modifications impossible. For reliability, we have several witnesses, not just one, and for speed and convenience, these are online.

Having decided on a list of witnesses, we can then select best the parent and the corresponding history that best fits the definition of reality as "somewhere where these witnesses live". At the same time, the parents themselves might have different witness lists and consequently different definitions of reality. We want the definitions of reality, and histories that follow from them, to converge around something common. To achieve this, we introduce the following additional protocol rule.

The "near-conformity rule": best parents must be selected only among those parents whose witness list differs from the child's witness list by no more than one mutation. This rule ensures that witness lists of neighboring units on the MC are similar enough, therefore their histories mostly agree with one another. The parents whose witness list differs by 0 or 1 mutation will be called *compatible* (with the unit that includes them directly), while the others are incompatible. Incompatible parents are still permitted, but they have no chance of becoming best parent. If there are no compatible potential parents among childless units (an attacker could flood the network with his units that carry a radically different witness list), one should select parents from older units.

The above means that each unit must list its witnesses so that they can be compared. We require that the number of witnesses is exactly 12. This number 12 was selected because:

- it is sufficiently large to protect against the occasional failures of a few witnesses (they might prove dishonest, or be hacked, or go offline for a long time, or lose their private keys and go offline forever);
- it is sufficiently small that humans can keep track of all the witnesses to know who is who and change the list when necessary;
- the one allowed mutation is sufficiently small compared with the 11 unchanged witnesses.

In case a user thinks that any of the witnesses has lost his credibility, or there are just better candidates, the user can replace the witness with a new witness in his list, bearing in mind that his witness list may not differ from that of other units by more than one position. This means that any changes can happen only gradually, and a general consensus is required for a change bigger than one position.

7. Finality

As new units arrive, each user keeps track of his *current MC* which is built as if he were going to issue a new unit based on all current childless units. The current MC may be different at different nodes because they may see different sets of childless units. We require that the current MC be built without regard of witness lists, i.e. the user's own witness list doesn't matter and even incompatible parents can be selected as best parents. That means that if two users have the same set of childless units, but have different witness lists, their current MCs will still be

identical. The current MC will constantly change as new units arrive. However, as we are about to show, a part of the current MC that is old enough will stay invariant.

We expect witnesses (or rather the majority thereof) to behave honestly, therefore necessarily include their previous unit in the next unit authored by the same witness. This means that when a witness composes a new unit, only recent units are candidates to be chosen as parents. We might expect, therefore, that all future current MCs will converge no farther (when traveling back in time) than a particular stability point. Indeed, the genesis unit is a natural initial stability point. Assume we have built a current MC based on the current set of childless units, and there was some point on this MC that was previously believed to be stable, i.e. all future current MCs are believed to converge on or before this point (again, when traveling back in time), and then travel along the same route. If we can find a way of advancing this point forward (away from the genesis), we can prove by induction that a stability point exists.

Note that if we forget about all parents except the best parent, our DAG will be reduced to a tree that consists only of best parent links. Obviously, all MCs will go along the branches of this tree. We then need to consider two cases – when the tree does branch in the current stability point and when it does not – and decide if we can advance the stability point to the next MCI.

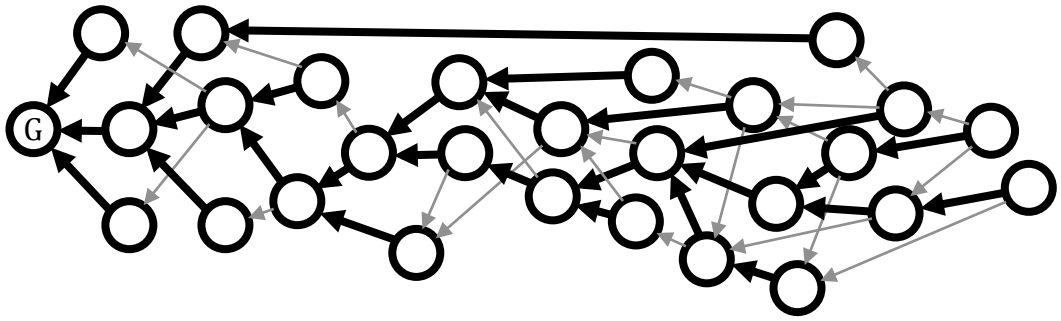

Figure 5. A tree composed of best-parent links. All but one branches stop growing after some point.

First, assume the tree does not branch. We then need to consider the possibility that a new branch will still be added and somehow supported by the witnesses so that it outgrows the existing branch. The other possibility is that the witnesses put so much weight in support of the existing branch, that the requirement of including one's previous unit leaves them no options but continue supporting the existing branch. Let's quantify the latter possibility. Remember that best parent is selected as the parent with the greatest witnessed level. Let's travel back in time along the current MC from the tip until we meet the majority of witnesses (we are referring to witnesses as defined by the unit lying on the current stability point). If at least one of them lies earlier than the current stability point,

we do not try to advance the stability point, otherwise we proceed. In this case, all these witnesses are already "invested" into the current MC. Among these witnesses, we find the minimum witnessed level min_wl. When any of these witnesses posts a new unit, this unit might have parents whose MC leads to the current MC and parents whose MC leads to a competing branch, and the parent with the highest witnessed level will be selected as best parent and will define the direction of the next current MC. Since the witness has to include its previous unit, the witnessed level of the parent leading to the current MC will be at least min_wl. The witnessed level of any parent leading to the alternative branch will never exceed the level of the current stability point, even if all remaining (minority) witnesses flock to the alternative branch. Therefore, if the current MC grows far enough so that min_wl is greater than the level of the current stability point, the majority of witnesses will have to increase support for the existing current MC, the alternative branch has then lost all chances to win, and we can move the stability point forward to the next MCI.

Next, assume the tree does branch. We need to find a condition where the alternative branches will lose any chance to outgrow the current MC. Let's start by defining min_wl as in the previous case. Among all units on the alternative branches, we then select those that increase the witness level, i.e. their own witnessed level is greater than that of every parent. Among these, we find the maximum level. Then, even if all the remaining (minority) witnesses gather on the alternative branches, the witnessed level on the alternative branches will never exceed this maximum level. Therefore, if this maximum level is less than min_wl, game is over for the alternative branches, and we can advance the stability point along the current MC.

Thus, there is a point on the current MC before which the MC will never change (assuming the majority of witnesses don't post nonserial units). The total order defined relative to this MC is therefore also final. If we had nonserials, our decision about which one of them is valid, is final as well. If a new nonserial ever appears that conflicts with anything already on the stable MC, the new nonserial unit will definitely be ordered after the old counterpart, and the new one will be deemed invalid. Therefore, any payment made in the unit included on the stable MC is already irreversible. Unlike Bitcoin where transaction finality is only probabilistic, this is deterministic transaction finality.

Every user builds his own (subjective) current MC based on the units that he sees. Since the propagation of new units is not instant, and they may arrive in different order to different users, the users will have different current MCs and different opinions about the last stable point of the MC at any given time. However, since the current MC is defined solely by the set of units known to the user, in case user B hasn't yet advanced his stability point to the same MCI as user A, he will inevitably do that later – i.e. as soon as he receives the same units as user A, or more. Thus the opinions of different users about the state of any given unit are eventually consistent.

8. Storage of nonserial units

When we decide that a unit is a nonserial, we still have to store it. However, part of its data is replaced with a hash of the data. This rule serves two purposes. First, to reduce storage consumed by a unit that nobody paid for (the entire content of the nonserial unit is deemed invalid, including its payment of commissions). Second, to reduce the utility of the nonserial to the user who sent it, because the hash replaces all useful data that the author wanted to store (for free). This prevents attackers from abusing nonserials as a way to store large amounts of data for free.

The hash that is stored instead of the full content still has some utility to the attacker, as he can store the original data himself and use the hash to prove that the data existed. But remember that:

1. He still has to pay for one unit that is deemed valid
2. If the attacker is already internally storing metadata that is necessary to interpret Byteball data, he could do equally well by just combining all his data into a Merkle tree and using Byteball to store only its Merkle root for the cost of one small unit.

Under this design, there is therefore no self-interest in trying to send nonserials.

It ought to be mentioned that we cannot just reject nonserials the first time we see them. If we did, an attacker could send his nonserials to different users in different order. Different users would then stick to the versions they first received and reject everything based on the other version, so the attacker would succeed in partitioning the network. That's why we have to store both versions and then decide on their order. Even more, users should forward nonserials to peers just like any other units, as the sooner peers learn about the nonserials the better.

We still try to avoid including nonserials if possible: the parent selection algorithm excludes nonserials as long as they are childless. For this reason, it's desirable to help peers learn about nonserials as soon as possible.

9. Balls

After a unit becomes stable (i.e. it is included on the stable part of the MC) we create a new structure based on this unit, we call it a *ball*:

```
ball: {
      unit: "hash of unit",
      parent_balls: [array of hashes of balls based on parent units],
      is_nonserial: true, // this field included only if the unit is
      nonserial
      skiplist_balls: [array of earlier balls used to build skiplist]
}
```

Every ball includes information about all its ancestor balls (via parents), hence the amount of information it depends on grows like snowball. We also have a flag in the ball that tells us if it ended up being invalid (nonserial), and we have references to older balls that we'll use later to build proofs for light clients.

We can only build a ball when the corresponding unit becomes stable and we know for certain whether it is serial. Since the current MCs as viewed by different users are eventually consistent, they will all build exactly the same ball based on the same unit.

10. Last ball

To protect the balls (most importantly, the is_nonserial flag) from modification, we require each new unit to include a hash of the last ball that the author knows about (which is the ball built from the last stable unit, and it lies on the MC). This way, the last ball will be protected by the author's signature. Later on, the new unit itself will be (directly or indirectly) included by witnesses.

If someone who doesn't have the entire Byteball database wants to know if a particular unit is serial, he would give us a list of witnesses he trusts to behave honestly, and we would build a chain of recent units that includes the majority of the said witnesses, then read last ball from the oldest unit of the chain, and use balls to build a hash tree that has the last ball at the top and includes the requested unit somewhere below. This hash tree is similar to a very tall Merkle tree, with additional data fed in at each node. The tree can be optimized using the skiplist.

The reference to the last ball also lets users see what their peers think about the stability point of the MC and compare it with their own vision.

We also require that the last ball lies no sooner than last ball of every parent. This ensures that the last ball either advances forward along the MC or stays in the same position, but never retreats.

To further reduce the degrees of freedom of adversaries, we add one more requirement: a unit's witness list must be compatible with that of each unit that lies on the trailing part of the unit's MC between this unit and the last ball's unit. This requirement ensures that all changes to the witness list first reach stability point before trying another change. Otherwise, an attacker might inject a significantly modified witness list onto the MC and stop posting from the addresses of the new witnesses. In such instances, the stability point would not be able to advance past the stretch occupied by the attacker's witnesses.

The requirement that witness lists of all contemporary units are mostly similar means that all users have mostly similar views about who can be trusted to serve as lighthouses for the community at the current time. This is similar to biology, where organisms of the same species have to have mostly the same genes. Small variance of the witness list allows for evolutionary change that still preserves the integrity of the system.

11. Witness list unit

It is expected that many users will want to use exactly the same witness list. In this case, to save space, they don't list the addresses of all 12 witnesses. Rather, they give a reference to another earlier unit, which listed these witnesses explicitly. The witness list unit must be stable from the point of view of the referencing unit, i.e. it must be included into the last ball unit.

12. Unit structure

This is an example of a unit:

```
{
	version: '1.0',
	alt: '1',
```

```
      messages: [ {
            app: 'payment',
            payload_location: 'inline',
            payload_hash:
            'AegecfpDzh8xvdyIABdynrcP6CTd4Pt42gvRiv0Ftjg=',
            payload: {
                  inputs: [{
                        unit:
                        '7yctnKyuAk5P+mFgFQDdDLza88nkceXYjsTs4e3doQA=',
                        message_index: 0,
                        output_index: 1
                  } ],
                  outputs: [
                        { address: 'DJ6LV5GPCLMGRW7ZB55IVGJRPDJPOQU6',
                        amount: 208 },
                        { address: 'Z36JFFX2AH7X5JQ2V2C6AQUUOWFESKZ2',
                        amount: 3505 }
                  ]
            }
      } ],
      authors: [ {
            address: 'DJ6LV5GPCLMGRW7ZB55IVGJRPDJPOQU6',
            authentifiers: {
                  r:
                  '3eQPIFiPVLRwBwEzxUR5thqn+zlFfLXUrzAmgemAqOk35UvDpa4h
                  79Fd6TbPbGfb8VMiJzqdNGHCKyAjl786mw=='
            }
      } ],
      parent_units: [
            'B63mnJ4yNNAE+6J+L6AhQ3EY7EO1Lj7QmAM9PS8X0pg=',
            'D6O1/D9L8vCMhv+8f70JecF93UoLKDp3e2+b92Yh2mI=',
            'ZxqzWP6q6hDNF50Wax8HUK212lH/KSIRdW5a6T9h3DM='
      ],
      last_ball: '8S2ya9lULt5abF1Z4lIJ4x5zYY9MtEALCl+jPDLsnsw=',
      last_ball_unit: 'bhdxFqVUut6V3N2D6Tyt+/YD6X0W+QnC95dMcJJWdtw=',
      witness_list_unit:
      'f252ZI2MN3xu8wFJ+LktVDGsay2Udzi/AUauE9ZaifY='
}
```

Here:

- version is the protocol version number. The unit will be interpreted according to this version of the protocol;
- alt is an identifier of alternative currency, we'll discuss this later;
- messages is an array of one or more messages that contain actual data;
    - app is the type of message, e.g. 'payment' for payments, 'text' for arbitrary text messages, etc;
    - payload_location says where to find the message payload. It can be 'inline' if the payload is included in the message, 'uri' if the payload is available at an internet address, 'none' if the payload is not published at all, is stored and/or shared privately, and payload_hash serves to prove it existed at a specific time;
    - payload_hash is a hash of the payload in base64 encoding;
    - payload is the actual payload (since it is 'inline' in this example). The payload structure is app-specific. Payments are described as follows:

        - inputs is an array of input coins consumed by the payment. All owners of the input coins must be among the signers (authors) of the unit;
            - unit is hash of the unit where the coin was produced. To be spendable, the unit must be included in last_ball_unit;
            - message_index is an index into the messages array of the input unit. It indicates the message where the coin was produced;
            - output_index is an index into the outputs array of the message_index'th message of the input unit. It indicates the output where the coin was produced;
        - outputs is an array of outputs that say who receives the money;
            - address is the Byteball address of the recipient;
            - amount is the amount he receives;
- authors is an array of the authors who created and signed this unit. All input coins must belong to the authors;
    - address is the author's Byteball address;
    - authentifiers is a data structure that proves the author's authenticity. Most commonly these are ECDSA signatures;
- parent_units is an array of hashes of parent units. It must be sorted alphabetically;
- last_ball and last_ball_unit are hashes of last ball and its unit, respectively;
- witness_list_unit is hash of the unit where one can find the witness list.

All hashes are in base64 encoding.

Note that there is no timestamp field in the unit structure. In Byteball, there are no protocol rules that rely on clock time. it's simply not needed, as it is enough to rely on the order of events alone.

Timestamp is still added to units when they are forwarded from node to node. However, this is only advisory and used by light clients to show in wallets the approximate time when a unit was produced, which may significantly differ from the time it was received as light clients may go offline for extended periods of time.

13. Commissions

As mentioned before, the cost to store a unit is its size in bytes. The commission is split into two parts: headers commission and payload commission. Payload commission is equal to the size of messages; headers commission is the size of everything else. The two types of commissions are distributed differently.

Headers commission goes to one of the future units which takes the payer unit as parent. The receiver is selected only after both the payer unit's MCI and the next MCI become stable. To determine the receiver, we take those children whose MCI is equal to or 1 more than the MCI of the payer. The hashes of each of these children are concatenated with the hash of the unit lying on the next MCI (relative to the payer), and the child with the smallest hash value (in hex) wins the headers

commission. This hashing with the next MC unit is designed to introduce unpredictability (the next MC unit is not known beforehand) and render useless any attempts to improve one's chances of receiving commission by playing with one's own unit hash. At the same time, restricting candidates to those whose MCI is no more than 1 greater than the MCI of the payer, incentivizes the selection of the most recent units as parents. This is useful to keep the DAG as narrow as possible.

We pay only the headers commission and not the entire commission to those who are quick to pick our unit as parent, for the following reason. If we did pay the entire commission, we would have incentivized abusive behavior: split one's data into several chunks and build a long chain of one's own units storing one chunk per unit. All the commissions paid in a previous unit would then be immediately collected by the same user in the next unit. As we pay only the headers commission, such behavior is not profitable because to produce each additional element of the chain one has to spend additional headers commission – roughly the same as one earns. We use the remaining (payload) commission to incentivize others whose activity is important for keeping the network healthy.

Payload commission goes to witnesses. To incentivize witnesses to post frequently enough, we split payload commission equally among all witnesses who are quick enough to post within 100 MC indexes after the paying unit (the faster they post, the faster this unit becomes stable). If all 12 witnesses have posted within this interval, each receives 1/12 of the payload commission. If only one witness has posted, he receives the entire payload commission. In the special case that no witness has posted within this interval, they all receive 1/12 of payload commission. If the division produces a fractional number, it is rounded according to mathematical rules. Because of this rounding, the total commission paid out to witnesses may not be equal to the total payload commission received from the unit's author(s), so the total money supply will change slightly as well. Obviously, the distribution happens only after MCI+100 becomes stable, where MCI is the MCI of the paying unit.

To spend the earned headers commissions or witnessing commissions, the following input is used:

```
inputs: [
	{
		type: "headers_commission",
		from_main_chain_index: 123,
		to_main_chain_index: 196
	},
	{
		type: "witnessing",
		from_main_chain_index: 60,
		to_main_chain_index: 142
	},
	…
]
```

Such inputs sweep all headers or witnessing commissions earned by the author from commission paying units that were issued between main chain indexes from_main_chain_index and to_main_chain_index. Naturally, to_main_chain_index must be stable.

When a unit signed by more than one author earns headers commission, there is so far ambiguity as to how the commission is split among the authors. To remove the ambiguity, each unit that is signed by more than one author must include a data structure that describes the proportions of revenue sharing:

```
unit: {
      …
      earned_headers_commission_recipients: [
            {address: "ADDRESS1", earned_headers_commission_share: 30},
            {address: "ADDRESS2", earned_headers_commission_share: 70}
      ],
      …
}
```

The addresses who receive the commissions needn't be the same as the author addresses – the commission can be sent to any address. Even if the unit is signed by a single author, it can include this field to redirect headers commissions elsewhere.

14. Confirmation time

Confirmation time is the time from a unit entering the database to reaching stability. It depends on how often the witnesses post, since to reach stability we need to accumulate enough witness-authored units on the MC after the newly added unit. To minimize the confirmation period, the witnesses should post frequently enough (which they are already incentivized to do via commission distribution rules) but not too frequently. If two or more witnesses issue their units nearly simultaneously (faster than it typically takes to propagate a new unit to other witnesses), this may cause unnecessary branching of the tree composed of best-parent links, which would delay stability. For this reason, the best confirmation times are reached when the witnesses are well connected and run on fast machines so that they are able to quickly validate new units. We estimate the best confirmation times to be around 30 seconds; this is only reachable if the flow of new units is large enough so that the witnesses earn more from witnessing commissions than they spend for posting their own units.

Despite the period of full confirmation being rather long, a node that trusts its peers to deliver all new units without filtering may be reasonably sure that once a unit was included by at least one witness, plus a typical latency has elapsed (the time it takes a new unit to travel from peer to peer), the unit will most likely reach finality and be deemed valid. Even if a double-spend appears later, it will be likely ordered after this unit.

15. Partitioning risk

The network of Byteball nodes can never be partitioned into two parts that would both continue operating without noticing. Even in the event of a global network disruption such as a sub-Atlantic rat cutting the cable that connects Europe and America, at least one of the sides of the split will notice that it has lost the majority of witnesses, meaning that it can't advance the stability point, and nobody can spend outputs stuck in the unstable part of the MC. Even if someone tries to send a

double-spend, it will remain unstable (and therefore unrecognized) until the connection is restored. The other part of the split where the majority of witnesses happens to be, will continue as normal.

16. Censorship

By design, it is already impossible to modify or erase any past records in Byteball. It is also quite hard to stop any particular types of data from entering the database.

First, the data itself can be concealed and only its hash be actually posted to the database to prove that the data existed. The data may only be revealed after the hash is stored and its unit has been included by other units so that it has become unrevisable.

Second, even when the data is open, the decision to include or not include it in the database is delegated to numerous anonymous users who might (and in fact are incentivized to) take the new unit as a parent. Someone who tries to censor undesirable units will have to not only avoid including them directly (as parents) but also indirectly, through other units. (This is different from Bitcoin where miners or mining pools can, and do, filter individual transactions directly. Besides, Bitcoin users have no say in who is to become a miner.) As the number of units which include the "offending" unit snowballs, any attempt to avoid it would entail censoring oneself. Only the majority of witnesses can effectively impose forbidden content rules – if users choose such witnesses.

17. Choosing witnesses

Reliance on witnesses is what makes Byteball rooted in the real world. At the same time, it makes it highly dependent on human decisions. The health of the system depends on users responsibly setting the lists of witnesses they do trust. This process cannot be safely automated, for example if most users start auto-updating their witness lists to match the lists of most recently observed units, just to be compatible, this can be easily exploited by an attacker who floods the network with his own units that gradually change the predominant witness list to something of the attacker's choosing.

While the maximalist recommendation could be "only edit witness lists manually", which is too burdensome for most users, a more practical approach to witness list management is tracking and somehow averaging the witness lists of a few "captains of industry" who either have interest in caring for the network health or who have earned a good reputation in activities not necessarily connected with Byteball. Some of them may be acting witnesses themselves. Unlike witness lists, the lists of captains of industry don't have to be compatible, and failing to update the list frequently enough doesn't have any immediate negative implications such as being unable to find compatible parents and post a new unit. We expect that most users will use one of a relatively small number of most popular wallets, and such wallets will be set up by default to follow the witness list of the wallet vendor, who in turn likely watches the witness lists of other prominent players.

Witnesses also have their witness lists, and it is recommended that users elect those witnesses who they trust to keep their witness lists representative of ordinary users' beliefs. This is very important because no change to the predominant witness list can pass without approval of the majority of the current witnesses. It is recommended that witnesses and would-be witnesses publicly declare their witness list policy (such as following and averaging witness lists of other reputable users), and that users evaluate their fitness for the job based on this policy, among other factors. Any breach of the declared policy will be immediately visible and will likely trigger a witness replacement campaign. The same is true for an unjustified amendment to the policy. The policy binds the witness and makes him follow public opinion, even when it turns against the witness himself or his friends.

As mentioned before, our protocol rules require that:

1. best parent is selected only among parents whose witness list has no more than 1 mutation;
2. there should be no more than 1 mutation relative to the witness list of the last ball unit;
3. there should be no more than 1 mutation relative to the witness lists of all the unstable MC units up to the last ball unit;
4. the stability point advances only when the current witnesses (as defined in the current stability point) post enough units after the current stability point.

These rules are designed to protect against malicious and accidental forks. At the same time, they imply that any changes of the predominant witness list have to be gradual, and each step has to be approved by the majority of the current witnesses. A one-position change has to first reach stability and recognition of the majority of old witnesses before another change can be undertaken. If the community decides abruptly that two witnesses need to be replaced immediately, then after one change makes its way onto the MC, the second change will be blocked by rule 3 above until the first change reaches stability.

Despite all the recommendations above it is still possible that due to the negligence of industry leaders, such witnesses are elected who later form a cartel and collectively block all attempts to replace any one of them in an attempt to keep the profits they are earning from witnessing commissions. If they do behave this way, it will be evident to everybody because their witness lists will remain unchanged, while the witness lists of most other industry leaders will differ by one mutation (the maximum allowed to remain compatible). If the old witnesses do not give in despite such evident pressure, the only recourse of the pro-change users is a "revolution" – i.e. to start a new coin that inherits all the balances, user addresses, etc from the old coin at some point but starts with a new witness list and adds a special protocol rule to handle this incompatible change at the moment of the schism. To distinguish from the old coin, they would then assign a new value to the 'alt' field (this what 'alt' is for) and use it in all units issued under the new coin. As a result, users will hold two coins (the old alt="1", and the new e.g. alt="2") and will be able to spend both independently. If the split was justified, the old coin will probably be abandoned, but all the data accumulated prior to the schism will be available as normal in the new coin. Since the protocol is almost

identical (except for the rule that handles the schism and the change of alt), it will be easy to update software installed on all user and merchant devices.

If someone just wants to start a new coin to experiment with another set of protocol rules, he can also use the 'alt' field to inherit everything from the old coin, make the switch comfortable for users, and have a large set of users with balances from day one.

18. Skiplist

Some of the balls contain a skiplist array which enables faster building of proofs for light clients (see below). Only those balls that lie directly on the MC, and whose MC index is divisible by 10, have a skiplist. The skiplist lists the nearest previous MC balls whose index has the same or smaller number of zeros at the end. For example, the ball at MCI 190 has a skiplist that references the ball at MCI 180. The ball at MCI 3000 has a skiplist that references the balls at MCIs 2990, 2900, and 2000.

19. Light clients

Light clients do not store the entire Byteball database. Instead, they download a subset of data they are interested in, such as only transactions where any of the user's addresses are spending or being funded.

Light clients connect to full nodes to download the units they are interested in. The light client tells the full node the list of witnesses it trusts (not necessarily the same witnesses it uses to create new units) and the list of its own addresses. The full node searches for units the light client is interested in and constructs a proof chain for each unit in the following way:

1. Walk back in time along the MC until the majority of requested witnesses are met. Collect all these MC units.
2. From the last unit in this set (which is also the earliest in time), read the last ball.
3. Starting from this last ball, walk back in time along the MC until any ball with a skiplist is met. Collect all these balls.
4. Using the skiplist, jump to an earlier ball referenced from the skiplist. This ball also has a skiplist, jump again. Where there are several balls in skiplist array, always jump by the largest distance possible, so we accelerate jumping first by 10 indexes, then by 100, then by 1000, etc.
5. If the next jump by the skiplist would throw us behind the target ball, decelerate by jumping by a smaller distance. Ultimately, leave the skiplist and walk along the MC one index at a time using just parent links.

This chain has witness-authored units in the beginning, making it trustworthy from the light client's point of view. All the elements of the chain are linked by either parent unit links (while accumulating the witnesses), or by last ball reference, or by parent ball links, or by skiplist links. At the end of the chain, we have the unit whose existence was to be proved.

20. Multilateral signing

A unit can be signed by multiple parties. In such instances, the authors array in the unit has two or more elements.

This can be useful, for example. if two or more parties want to sign a contract (a plain old dumb contract, not a smart one). They would both sign the same unit that contains a text message (app='text'). They don't have to store the full text of the contract in the public database, and pay for it – a hash would suffice (payload_location='none'), and the parties themselves can store the text privately.

Another application of multilateral signing is an exchange of assets. Assume user A wants to send asset X to user B in exchange for asset Y (the native currency 'bytes' is also an asset – the base asset). Then they would compose a unit that contains two payment messages: one payment sends asset X from A to B, the other payment sends asset Y from B to A. They both sign the dual-authored unit and publish it. The exchange is atomic – that is, either both payments execute at the same time or both fail. If one of the payments appears to be a double-spend, the entire unit is rendered invalid and the other payment is also deemed void.

This simple construction allows users to exchange assets directly, without trusting their money to any centralized exchanges.

21. Addresses

Users are identified by their addresses, transaction outputs are sent to addresses, and, like in Bitcoin, it is recommended that users have multiple addresses and avoid reusing them. In some circumstances, however, reuse is normal. For example, witnesses are expected to repeatedly post from the same address.

An address represents a definition, which is a Boolean expression (remotely similar to Bitcoin script). When a user signs a unit, he also provides a set of authentifiers (usually ECDSA signatures) which, when applied to the definition, must evaluate it to true in order to prove that this user had the right to sign this unit. We write definitions in JSON. For example, this is the definition for an address that requires one ECDSA signature to sign:

```
["sig",{"pubkey":"Ald9tkgiUZQQ1djpZgv2ez7xf1ZvYAsTLhudhvn0931w"}]
```

The definition indicates that the owner of the address has a private key whose public counterpart is given in the definition (in base64 encoding), and he will sign all units with this private key. The above definition evaluates to true if the signature given in the corresponding authentifier is valid, or otherwise false. The signature is calculated over all data of the unit except the authentifiers.

Given a definition object, the corresponding address is just a hash of the initial definition object plus a checksum. The checksum is added to avoid typing errors. Unlike usual checksum designs, however, the checksum bits are not just appended to the end of the unchecksummed data. Rather, they are inserted into multiple locations inside the data. This design makes it hard to insert long strings of illegal data in fields where an address is expected. The address is written in base32 encoding. The above definition corresponds to address A2WWHN7755YZVMXCBLMFWRSLKSZJN3FU.

When an address is funded, the sender of the payment knows and specifies only the address (the checksummed hash of the definition) in the payment output. The definition is not revealed and it remains unknown to anyone but the owner until the output is spent.

When a user sends his first unit from an address, he must reveal its definition (so as to make signature verification possible) in the authors array:

```
unit: {
      …
      authors: [ {
            address: 'DJ6LV5GPCLMGRW7ZB55IVGJRPDJPOQU6',
            definition: [
                  "sig",
                  {"pubkey":"AsnvZ3w7N1lZGJ+P+bDZU0DgOwJcGJ51bjsWpEqfqB
                  g6"}
            ],
            authentifiers: {
                  r:
                  '3eQPIFiPVLRwBwEzxUR5thqn+zlFfLXUrzAmgemAqOk35UvDpa4h
                  79Fd6TbPbGfb8VMiJzqdNGHCKyAjl786mw=='
            }
      } ],
      …
}
```

If the user sends a second unit from the same address, he must omit the definition (it is already known on Byteball). He can send the second unit only after the definition becomes stable, i.e. the unit where the definition was revealed must be included in the last ball unit of the second unit.

Users can update definitions of their addresses while keeping the old address. For example, to rotate the private key linked to an address, the user needs to post a unit that contains a message such as:

```
unit: {
      …
      messages: [
            …
            {
                  app: "address_definition_change",
                  definition_chash: "I4Z7KFNIYTPHPJ5CA5OFC273JQFSZPOX"
            },
            …
      ],
      …
}
```

Here, definition_chash indicates the checksummed hash of the new address definition (which is not revealed until later), and the unit itself must be signed by the old private keys. The next unit from this address must:

- include this address_definition_change unit in its last ball unit, i.e. it must be already stable;
- reveal the new definition in the authors array in the same way as for the first message from an address.

After the change, the address is no longer equal to the checksummed hash of its current definition. Rather, it remains equal to the checksummed hash of its *initial* definition.

The definition change is useful if the user wants to change the key(s) (e.g. when migrating to a new device) while keeping the old address, e.g. if this address already participates in other long-lived definitions (see below).

21.1. Definition syntax

21.1.1. Logical operators

A definition can include "and" conditions, for example:

```
["and", [
      ["sig", {pubkey: "one pubkey in base64"}],
      ["sig", {pubkey: "another pubkey in base64"}]
]]
```

which is useful when, in order to sign transactions, signatures from two independent devices are required, for example, from a laptop and from a smartphone.

"Or" conditions, such as this:

```
["or", [
      ["sig", {pubkey: "laptop pubkey"}],
      ["sig", {pubkey: "smartphone pubkey"}],
      ["sig", {pubkey: "tablet pubkey"}]
]]
```

are useful when a user wants to use the same address from any of his devices.

The conditions can be nested:

```
["and", [
      ["or", [
            ["sig", {pubkey: "laptop pubkey"}],
            ["sig", {pubkey: "tablet pubkey"}]
      ]],
      ["sig", {pubkey: "smartphone pubkey"}]
]]
```

A definition can require a minimum number of conditions to be true out of a larger set, for example, a 2-of-3 signature:

```
["r of set", {
      required: 2,
      set: [
            ["sig", {pubkey: "laptop pubkey"}],
            ["sig", {pubkey: "smartphone pubkey"}],
            ["sig", {pubkey: "tablet pubkey"}]
      ]
}]
```

("r" stands for "required") which features both the security of two mandatory signatures and the reliability, so that in case one of the keys is lost, the address is still usable and can be used to change its definition and replace the lost 3rd key with a new one.

Also, different conditions can be given different weight, of which a minimum is required:

```
["weighted and", {
      required: 50,
      set: [
            {weight: 40, value: ["sig", {pubkey: "CEO pubkey"}] },
            {weight: 20, value: ["sig", {pubkey: "COO pubkey"}] },
            {weight: 20, value: ["sig", {pubkey: "CFO pubkey"}] },
            {weight: 20, value: ["sig", {pubkey: "CTO pubkey"}] }
      ]
}]
```

21.1.2. Delegation to other addresses

An address can contain reference to another address:

```
["and", [
      ["address", "ADDRESS 1 IN BASE32"],
      ["address", "ADDRESS 2 IN BASE32"]
]]
```

which delegates signing to another address and is useful for building shared control addresses (addresses controlled by several users). This syntax gives the users the flexibility to change definitions of their own component addresses whenever they like, without bothering the other user.

21.1.3. Signatures and authentifiers

In most cases, a definition will include at least one signature (directly or indirectly):

```
["sig", {pubkey: "pubkey in base64"}]
```

Instead of a signature, a definition may require a preimage for a hash to be provided:

```
["hash",{"hash":"value of sha256 hash in base64"}]
```

which can be useful for cross-chain exchange algorithms [7]. In this case, the hash preimage is entered as one of the authentifiers.

The default signature algorithm is ECDSA on curve secp256k1 (same as Bitcoin). Initially, it is the only algorithm supported. If other algorithms are added in the future, algorithm identifier will be used in the corresponding part of the definition, such as for the quantum secure NTRU algorithm:

```
["sig", {algo: "ntru", pubkey: "NTRU public key in base64"}]
```

Multisignature definitions allow one to safely experiment with unproven signature schemes when they are combined with more conventional signatures.

The authentifiers object in unit headers contains signatures or other data (such as hash preimage) keyed by the path of the authentifier-requiring subdefinition within the address definition. For a single-sig address such as

```
["sig", {pubkey: "pubkey in base64"}]
```

the path is simply “r” (r stands for root). If the authentifier-requiring subdefinition is included within another definition (such as and/or), the path is extended by an index into the array where this subdefinition is included, and path components are delimited by a dot. For example, for address definition:

```
["and", [
      ["sig", {pubkey: "one pubkey in base64"}],
```

```
      ["sig", {pubkey: "another pubkey in base64"}]
]]
```

the paths are “r.0” and “r.1”. For a deeper nested definition:

```
["and", [
      ["or", [
            ["sig", {pubkey: "laptop pubkey"}],
            ["sig", {pubkey: "tablet pubkey"}]
      ]],
      ["sig", {pubkey: "smartphone pubkey"}]
]]
```

the paths are “r.0.0”, “r.0.1”, and “r.1”. When there are optional signatures, such as 2-of-3, the paths tell us which keys were actually used.

21.1.4. Definition templates

A definition can also reference a definition template:

```
["definition template", [
      "hash of unit where the template was defined",
      {param1: "value1", param2: "value2"}
]]
```

The parameters specify values of variables to be replaced in the template. The template needs to be saved before (and as usual, be stable before use) with a special message type app=’definition_template’, the template itself is in message payload, and the template looks like normal definition but may include references to variables in the syntax @param1, @param2. Definition templates enable code reuse. They may in turn reference other templates.

21.1.5. Cosigning

A subdefinition may require that the unit be cosigned by another address:

```
["cosigned by", "ANOTHER ADDRESS IN BASE32"]
```

21.1.6. Querying whether an address was used

Another possible requirement for a subdefinition: that an address was seen as author in at least one unit included into the last ball unit:

```
["seen address", "ANOTHER ADDRESS IN BASE32"]
```

21.1.7. Data feeds

One very useful condition can be used to make queries about data previously stored in Byteball:

```
["in data feed", [
      ["ADDRESS1", "ADDRESS2", …],
      "data feed name",
      "=",
      "expected value"
]]
```

This condition evaluates to true if there is at least one message that has "data feed name" equal to "expected value" among the data feed messages authored by the listed addresses "ADDRESS1", "ADDRESS2", .. (oracles). Data feed is a message type that looks like this:

```
unit: {
```

```
      …
      messages: [
            …
            {
                  app: "data_feed",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        "data feed name": "value",
                        "another data feed name": "value2",
                        …
                  }
            },
            …
      ],
      …
}
```

Data fields can be used to design definitions that involve oracles. If two or more parties trust a particular entity (the oracle) to provide true data, they can set up a shared control address that gives the parties different rights depending on data posted by the oracle(s). For example, this address definition represents a binary option:

```
["or", [
      ["and", [
            ["address", "ADDRESS 1"],
            ["in data feed", [["EXCHANGE ADDRESS"], "EURUSD", ">",
            "1.1500"]]
      ]],
      ["and", [
            ["address", "ADDRESS 2"],
            ["in data feed", [["TIMESTAMPER ADDRESS"], "datetime", ">",
            "2016-10-01 00:00:00"]]
      ]]
]]
```

Initially, the two parties fund the address defined by this definition (to remove any trust requirements, they use multilateral signing and send their stakes in a single unit signed by both parties). Then if the EUR/USD exchange rate published by the exchange address ever exceeds 1.1500, the first party can sweep the funds. If this doesn't happen before Oct 1, 2016 and the timestamping oracle posts any later date, the second party can sweep all funds stored on this address. If both conditions are true and the address balance is still non-empty, both parties can try to take the money from it at the same time, and the double-spend will be resolved as usual.

The comparison operators can be "=", "!=", ">", ">=", "<", and "<=". The data feed message must come before the last ball unit as usual. To reduce the risks that arise in case any single oracle suddenly goes offline, several feed provider addresses can be listed.

Another example would be a customer who buys goods from a merchant but he doesn't quite trust that merchant and wants his money back in case the goods are not delivered. The customer pays to a shared address defined by:

```
["or", [
      ["and", [
```

```
            ["address", "MERCHANT ADDRESS"],
            ["in data feed", [["FEDEX ADDRESS"], "tracking", "=",
            "123456"]]
      ]],
      ["and", [
            ["address", "BUYER ADDRESS"],
            ["in data feed", [["TIMESTAMPER ADDRESS"], "datetime", ">",
            "2016-10-01 00:00:00"]]
      ]]
]]
```

The definition depends on the FedEx oracle that posts tracking numbers of all successfully delivered shipments. If the shipment is delivered, the merchant will be able to unlock the money using the first condition. If it is not delivered before the specified date, the customer can take his money back.

This example is somewhat crazy because it requires FedEx to post each and every shipment.

21.1.8. Merkle data feeds

For a more realistic way to achieve the same goal, there is another syntax:

```
["in merkle", [
      ["ADDRESS1", "ADDRESS2", …],
      "data feed name",
      "hash of expected value"
]]
```

which evaluates to true if the specified hash of expected value is included in any of the merkle roots posted in the data feed from addresses "ADDRESS1", "ADDRESS2",... Using this syntax, FedEx would only periodically post merkle roots of all shipments completed since the previous posting. To spend from this address, the merchant would have to provide the merkle path that proves that the specified value is indeed included in the corresponding merkle tree. The merkle path is supplied as one of the authentifiers.

21.1.9. Self-inspection

A definition can also include queries about the unit itself. This subdefinition

```
['has', {
      what: 'input'|'output',
      asset: 'assetID in base64 or "base" for bytes',
      type: 'transfer'|'issue',
      own_funds: true,
      amount_at_least: 123,
      amount_at_most: 123,
      amount: 123,
      address: 'INPUT OR OUTPUT ADDRESS IN BASE32'
}]
```

evaluates to true if the unit has at least one input or output (depending on the 'what' field) that passes all the specified filters, with all filters being optional.

A similar condition 'has one' requires that there is exactly one input or output that passes the filters.

The 'has' condition can be used to organize a decentralized exchange. Previously, we discussed the use of multilateral signing to exchange assets.

However, multilateral signing alone doesn't include any mechanism for price negotiation. Assume that a user wants to buy 1,200 units of another asset for which he is willing to pay no more than 1,000 bytes. Also, he is not willing to stay online all the time while he is waiting for a seller. He would rather just post an order at an exchange and let it execute when a matching seller comes along. He can create a limit order by sending 1,000 bytes to an address defined by this definition:

```
["or", [
	["address", "USER ADDRESS"],
	["and", [
		["address", "EXCHANGE ADDRESS"],
		["has", {
			what: "output",
			asset: "ID of alternative asset",
			amount_at_least: 1200,
			address: "USER ADDRESS"
		}]
	]]
]]
```

The first or-alternative lets the user take back his bytes whenever he likes, thus cancelling the order. The second alternative delegates the exchange the right to spend the funds, provided that another output on the same unit pays at least 1,200 units of the other asset to the user's address. The exchange would publicly list the order, a seller would find it, compose a unit that exchanges assets, and multilaterally sign it with the exchange.

One can also use the 'has' condition for collateralized lending. Assume a borrower holds some illiquid asset and needs some bytes (or another liquid asset). The borrower and a lender can then multilaterally sign a unit. One part of the unit sends the bytes he needs to the borrower, the other part of the unit locks the illiquid asset into an address defined by:

```
["or", [
	["and", [
		["address", "LENDER ADDRESS"],
		["in data feed", [["TIMESTAMPER ADDRESS"], "datetime", ">",
		"2017-06-01 00:00:00"]]
	]],
	["and", [
		["address", "BORROWER ADDRESS"],
		["has", {
			what: "output",
			asset: "base",
			amount: 10000,
			address: "LENDER ADDRESS"
		}]
	]],
	["and", [
		["address", "LENDER ADDRESS"],
		["address", "BORROWER ADDRESS"]
	]]
]]
```

The first or-alternative allows the lender to seize the collateral if the loan is not paid back in time. The second alternative allows the borrower to take back the

collateral if he also makes a payment of 10,000 bytes (the agreed loan size including interest) to the lender. The third alternative allows the parties to amend the terms if they both agree.

The following requirement can also be included in a subdefinition:

```
['has equal', {
      equal_fields: ['address', 'amount'],
      search_criteria: [
            {what: 'output', asset: 'asset1', address: 'BASE32'},
            {what: 'input', asset: 'asset2', type: 'issue', own_funds:
            true, address: 'ANOTHERBASE32'}
      ]
}]
```

It evaluates to true if there is at least one pair of inputs or outputs that satisfy the search criteria (the first element of the pair is searched by the first set of filters; the second by the second) and some of their fields are equal.

A similar condition 'has one equal' requires that there is exactly one such pair.

Another subdefinition may compare the sum of inputs or outputs filtered according to certain criteria to a target value or values:

```
['sum', {
      filter: {
            what: 'input'|'output',
            asset: 'asset or base',
            type: 'transfer'|'issue',
            own_funds: true,
            address: 'ADDRESS IN BASE32'
      },
      at_least: 120,
      at_most: 130,
      equals: 123
}]
```

21.1.10. Negation

Any condition that does not include "sig", "hash", "address", "cosigned by", or "in merkle" can be negated:

```
["not", ["in data feed", [["NOAA ADDRESS"], "wind_speed", ">",
"200"]]]
```

Since it is legal to select very old parents (that didn't see the newer data feed posts), one usually combines negative conditions such as the above with the requirement that the timestamp is after a certain date.

21.2. General requirements

Address definition must have at least one "sig", explicitly or implicitly (such as through an "address").

To avoid consuming too many resources for validation, the total number of operations is limited to 100 per definition, including operations in referenced definitions such as "address" and "definition template".

This number is one of just 9 arbitrary constants that we have in Byteball, the other 8 being: total number of witnesses: 12; max allowed mutations: 1; max number of MC indexes for a witness to get paid: 100; number of parents counted

for header size: 2; max number of messages per unit: 128; max number of inputs or outputs per message: 128; max number of authors per unit: 16; and total money supply: 10^{15}. For comparison, Bitcoin has at least 17 constants [8], while Ethereum defines 30 constants for fee schedule alone [9].

Note that the definition language described above is declarative and consists entirely of Boolean statements, which puts it closer to the language of conventional legal contracts. However, in terms of its expressive power, the language does not come anywhere close to Ethereum smart contracts language. In fact, it doesn't even allow for a trivial 'Hello world' program to be written. This was not our goal. The Byteball definition language was not designed to be comprehensive; rather, it is designed to be comprehensible to the greatest possible number of people, who are not necessarily programmers. Its straightforward syntax allows everyone to interpret and compose simple definitions without the help of a developer (a "lawyer" for the era of smart contracts), and chances of mistakes are minimized.

22. Profiles

Users can store their profiles on Byteball if they want. They use a message like this:

```
unit: {
      …
      messages: [
            ….
            {
                  app: "profile",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        name: "Joe Average",
                        emails: ["joe@example.com", "joe@domain.com"],
                        twitter: "joe"
                  }
            },
            …
      ],
      …
}
```

The amount of data they disclose about themselves, as well as its veracity, is up to the users themselves. To be assured that any particular information about a user is true, one has to look for attestations.

23. Attestations

Attestations confirm that the user who issued the attestation (the attestor) verified some data about the attested user (the subject). Attestations are stored in messages like this:

```
unit: {
      …
      messages: [
```

```
            …
            {
                  app: "attestation",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        address: "ADDRESS OF THE SUBJECT"
                        profile: {
                              name: "Joe Average",
                              emails: ["joe@example.com"]
                        }
                  }
            },
            …
      ],
      …
}
```

The information included in the attestation need not be the same as in user's self-published profile. Indeed, the self-published profile might not even exist at all.

The job of attestors is similar to that of modern certification authorities who verify the real-world identities of subjects and certify that a particular public key (or Byteball address) does belong to a person or organization. We expect them to continue the same activity in Byteball and charge a fee from those who want to prove a link between their real-world and Byteball identities. Witnesses and would-be witnesses will likely want to receive some attestations to increase their trust. Certain asset types may require attestations to transact with the asset (see below).

For applications where an attestation is required but the name of the subject is not important, it is possible to omit the name or other personally identifiable information in the attested profile. The attested profile may even not include any meaningful information about the subject at all, thus leaving him anonymous to everybody but the attestor. The attestor will still keep records about the subject and may disclose them under certain circumstances, as specified in the attestor's terms or if required by law.

24. Assets

We have designed a database that allows immutable storage of any data. Of all classes of data, the most interesting for storage in a common database are those that have social value, i.e. the data that is valuable for more than one or two users. One such class is assets. Assets can be owned by anybody among a large number of people, and the properties of immutability and total ordering of events that we have in Byteball are very important for establishing the validity of long chains of ownership transfers. Assets in Byteball can be issued, transferred, and exchanged, and they behave similarly to the native currency 'bytes'. They can represent anything that has value, for example debt, shares, loyalty points, airtime minutes, commodities, other fiat or crypto currencies.

To define a new asset, the defining user sends a message like this:

```
unit: {
      …
```

```
        messages: [
                …
                {
                        app: "asset",
                        payload_location: "inline",
                        payload_hash: "hash of payload",
                        payload: {
                                cap: 1000000,
                                is_private: false,
                                is_transferrable: true,
                                auto_destroy: false,
                                fixed_denominations: false,
                                issued_by_definer_only: true,
                                cosigned_by_definer: false,
                                spender_name_attested: true,
                                attestors: [
                                        "2QLYLKHMUG237QG36Z6AWLVH4KQ4MEY6",
                                        "X5ZHWBYBF4TUYS35HU3ROVDQJC772ZMG"
                                ]
                        }
                },
                …
        ],
        …
}
```

Here:

- cap is the maximum amount that can be issued. For comparison with the predefined native currency bytes, the bytes cap is 10^{15};
- is_private indicates if the asset is transferred privately or publicly (see below). Bytes are public;
- is_transferrable indicates if the asset can be transferred between third parties without passing through the definer of the asset. If not transferrable, the definer must always be either the only sender or the only receiver of every transfer. Bytes are transferrable;
- auto_destroy indicates if the asset is destroyed when it is sent to the definer. Bytes are not auto-destroyed;
- fixed_denominations indicates if the asset can be sent in any integer amount (arbitrary amounts) or only in fixed denominations (e.g. 1, 2, 5, 10, 20, etc), which is the case for paper currency and coins. Bytes are in arbitrary amounts;
- issued_by_definer_only indicates if the asset can be issued by definer only. For bytes, the entire money supply is issued in the genesis unit;
- cosigned_by_definer indicates if every transfer must be cosigned by the definer of the asset. This is useful for regulated assets. Transfers in bytes needn't be cosigned by anybody;
- spender_attested indicates if the spender has to be attested in order to spend. If he happened to receive the asset but is not yet attested, he has to pass attestation with one of the attestors listed under the definition, in order to be able to spend. This requirement is also useful for regulated assets. Bytes do not require attestation;

- attestors is the list of attestor addresses recognized by the asset definer (only if spender_attested is true). The list can be later amended by the definer by sending an 'asset_attestors' message that replaces the list of attestors;
- denominations (not shown in this example and used only for fixed_denominations assets) lists all allowed denominations and total number of coins of each denomination that can be issued;
- transfer_condition is a definition of a condition when the asset is allowed to be transferred. The definition is in the same language as the address definition, except that it cannot reference anything that requires an authentifier, such as "sig". By default, there are no restrictions apart from those already defined by other fields;
- issue_condition is the same as transfer_condition but for issue transactions.

There can be no more than 1 'asset' message per unit. After the asset is defined, it is identified by the hash of the unit where it was defined (hence the 1 asset per unit requirement).

A transfer of an asset looks like a transfer of bytes, the difference being that there is an extra field for the asset ID:

```
unit: {
      …
      messages: [
            …
            {
                  app: "payment",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        asset: "hash of unit where the asset was
                        defined",
                        inputs: [
                              {
                                    unit: "hash of source unit",
                                    message_index: 0,
                                    output_index: 1
                              },
                              …
                        ],
                        outputs: [
                              {
                                    address: "BENEFICIARY ADDRESS",
                                    amount: 12345
                              },
                              …
                        ]
                  }
            },
            …
      ],
      …
}
```

Before it can be transferred, an asset is created when a user sends an issue transaction. Issue transactions have a slightly different format for inputs:

```
unit: {
      …
      messages: [
            …
            {
                  app: "payment",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        asset: "hash of unit where the asset was
                        defined",
                        inputs: [
                              {
                                    type: "issue",
                                    amount: 1000000,
                                    serial_number: 1,
                                    address: "ISSUER ADDRESS" // only
                                    when multi-authored
                              },
                              …
                        ],
                        outputs: [
                              {
                                    address: "BENEFICIARY ADDRESS",
                                    amount: 12345
                              },
                              …
                        ]
                  }
            },
            …
      ],
      …
}
```

The entire supply of capped arbitrary-amounts assets must be issued in a single transaction. In particular, all bytes are issued in the genesis unit. If the asset is capped, the serial number of the issue must be 1. If it is not capped, the serial numbers of different issues by the same address must be unique.

An asset is defined only once and cannot be amended later, only the list of attestors can be amended.

It's up to the definer of the asset what this asset represents. If it is issuer's debt, it is reasonable to expect that the issuer is attested or waives his anonymity to earn the trust of the creditors.

While end users are free to use or not to use an asset, asset definers can impose any requirements on transactions involving the asset.

By combining various asset properties the definer can devise assets that satisfy a wide range of requirements, including those that regulated financial institutions have to follow. For example, by requiring that each transfer be cosigned by the definer, financial institutions can effectively veto all payments that contradict any regulatory or contractual rules. Before cosigning each payment, the financial institution (who is also the definer and the issuer) would check that the user is indeed its client, that the recipient of the funds is also a client, that both clients have passed all the Know Your Client (KYC) procedures, that the funds are

not arrested by a court order, as well as carry out any other checks required by the constantly changing laws, regulations, and internal rules, including those that were introduced after the asset was defined.

24.1. Bank issued assets

Having the security of being fully compliant (and also assured in the familiar deterministic finality of all funds transfers), banks can issue assets that are pegged to national currencies and backed by the bank's assets (which are properly audited and monitored by the central banks). The legal nature of any operations with such assets is exactly the same as with all other bank money, and is familiar to everybody. The only novelty is that the balances and transfers are tracked in Byteball database instead of the bank's internal database. Being tracked in Byteball database has two consequences:

- (a not so welcome one) all operations are public, which is familiar from Bitcoin and mitigated by using multiple semi-anonymous addresses of which only the bank knows the real persons behind the addresses. Another more robust way to preserve privacy is private payments, which we'll discuss later;
- (a good one) the bank-issued asset can be exchanged for bytes or other assets on-chain, in a peer-to-peer manner, without having to trust any third parties such as exchanges.

The banks here are similar to Ripple gateways.

In the exchange scenario above, one leg of the exchange is payment from one user to another user in a bank-issued asset. If both users are clients of the same bank, this process is straightforward. When users hold accounts at different banks, the banks may facilitate the interbank transfers by opening correspondent accounts at each other. Let's assume user U1 wants to transfer money to user U2 in circumstances where user U1 holds an account at bank B1 and user U2 holds an account at bank B2. Bank B2 also opens an account at B1. U1 then transfers the money to B2's account at B1 (it is an internal bank transfer within B1 which is cosigned by B1). At the same time, B2 (which has just increased its assets at B1) issues new money to its user U2. All this must be atomic. All three participants: (U1, B1, and B2) must therefore sign a single unit that both transfers B1's money from U1 to B2 and issues B2's money to U2.

The net result is that U1 decreased his balance at B1, U2 increased his balance at B2, and B2 increased his balance at B1. The bank B1 will also have a correspondent account at B2, the balance of which will grow as reverse payments are processed from users of B2 to users of B1. The mutual obligations (B1 at B2 and B2 at B1) can be partially cancelled by the banks mutually signing a transaction that sends equal amounts to the respective issuer (it is convenient to have the money auto-destroyed by sending it to the issuer). What is not cancelled can be periodically settled through traditional interbank payments. To trigger the settlement, the bank with a positive net balance sends his balance to the issuer bank, and since there is no reverse transfer in the same transaction, this triggers a traditional payment in fiat money from the issuer to the holder bank.

When there are many banks, setting up direct correspondent relations with each peer bank can be cumbersome. In such instances, the banks agree about a

central counterparty C (a large member bank or a new institution) and pass all payments exclusively through this central counterparty and settle only with it. The same transfer from U1 to U2 will then consist of 3 transactions:

1. U1 sends money to C's account at B1;
2. C issues own money to B2 (or C destroys B2's money it held by returning it to B2);
3. B2 issues its own money to U2.

All 3 transactions are bundled into a single unit and signed by U1, B1 (as the required cosigner for all U1's transactions), C, and B2.

24.2. Non-financial assets

Other applications that are not necessarily financial can use Byteball assets internally. For example, loyalty programs may issue loyalty points as assets and use Byteball's existing infrastructure to allow people to transact in these points, including peer-to-peer (if allowed by the program's rules). The same is true for game developers, who can track game assets on Byteball.

24.3. Bonds

Businesses can issue bonds on Byteball. The legal structure of the issue is the same as for conventional bonds, the only difference being that the depository will now track bond ownership using Byteball rather than an internal database (similar to banks above). Having bonds in Byteball enables their holders to trade directly, without a centralized exchange. When bank money is also on Byteball, an instant delivery versus payment (a fiat payment in this context) becomes possible, without counterparty risk and without any central institution. The title to the bond and payment are exchanged simultaneously as the parties sign the same unit that performs both transfers.

Bonds, if liquid enough, can also be used by third parties as a means of payment.

When a bond is issued, the issuer and the investor would multilaterally sign a common unit that sends the newly issued bonds to the investor and at the same time sends bytes (or another asset used to purchase the bonds, such as a bank-issued fiat-pegged asset) from the investor to the borrower. When the bond is redeemed, they sign another multilateral unit that reverses the exchange (most likely, at a different exchange rate). The price of the bond paid during redemption is its face value, while the price it is sold for when issued must be lower than the face value to reflect interest (assuming zero coupon bond for simplicity). During its lifetime, the secondary market price of the bond stays below face value and gradually approaches it.

In a growing economy where there are many projects to finance, bonds and other debt issued on Byteball to finance investment will be issued more often than they are redeemed. When the economy slows down, the total supply of all bonds shrinks, as there are fewer projects to finance. Thus, the total supply of bonds self regulates, which is important if they are actively used as a means of payment.

If two businesses transact on net-30 terms, both buyer and seller have the option to securitize the trade credit during the 30-day period. For example, the buyer can issue 30-day bonds and use them to pay the seller immediately. The seller can then either wait for the 30 days to pass and redeem the bonds, or use the

bonds as a means of payment to its own suppliers. In this case, it will be the suppliers who redeem the bonds when they mature.

24.4. Commodity bonds

Bonds can be issued in natural units, not just in currencies. For example, a 100-barrel bond entitles its holder to receive 100 barrels of oil when the bond matures; a 1kWh bond entitles the holder to receive 1 kWh of electricity. The holder may also choose to receive the monetary equivalent of the 100 barrels or 1 kWh at the price that is current on the maturity date.

Such bonds (commodity bonds) are in fact very useful for hedging risks. Consider a new oil project that takes many years and large investment before it even starts commercial operation. If financing is sought only in national currencies, the project may never be financed because of uncertain oil prices at the time the new facility starts selling oil. The creditors have to consider the risk that the price will be too low, and as a result the borrower will have to default. Creditors want the risk priced into the interest rate, which means the interest rate becomes too high, and the project never happens.

However, if the project operator could borrow in barrels, the risk of default drastically decreases. Now, the project will likely start as planned and will likely produce the planned volume of oil. It will hence be able to produce and repay all the borrowed barrels within the specified time. There are still other risks, but one huge risk – the market risk – is removed. It is removed from the borrower but shifted to the lenders who now have to consider the chances that oil prices go down and they receive less (in currency terms) than invested. On the other hand, if the prices go up, the lenders get additional profit from the price difference (note that by borrowing in barrels, the borrower waives this upside potential), and there are always investors willing to take a position in a commodity. Since the bond is traded on Byteball, the lenders can easily sell it whenever they like. Unlike oil futures, whose trading is a zero-sum game, the investment in commodity bonds does finance the industry. Also, oil futures are a short-term instrument, while commodity bonds allow one to buy and hold, which is more suitable to long term investors.

There is another category of potential lenders – those who hedge against the opposite risk. For example, airlines would like to hedge against an increase of oil prices, and one way to do that is by buying commodity bonds of oil producing companies, which one expects to correlate with oil prices.

The above is true for any commodity, e.g. electricity, iron ore, gold, other metals, crops, etc.

From the borrower's perspective, commodity bonds can be thought of as a way to sell future production at today's prices. For the lender, it is a way to buy future supplies at today's prices.

If a substantial part of the economy runs on commodity bonds, the leverage cycle is naturally smoothed out even without government intervention since during recessions falling commodity prices automatically reduce the amount of debt.

24.5. Funds

For individual users, it might be difficult to track the huge number of bonds that are available on the market. Instead, they would rather choose to invest in funds that are professionally managed and hold a large diversified portfolio of bonds. The fund would issue its own asset that tracks the aggregate value of the fund's portfolio. Every time an investor buys a newly issued asset of the fund, the fund would use the proceeds to buy bonds. When a user exits, the fund sells some of the bonds it held and destroys the fund-issued assets returned by the user. The fund's asset is not capped; its total supply varies as investors enter and exit. Its value is easily auditable as all the bonds held by the fund are visible on Byteball. Being more liquid than the underlying bonds, the fund's asset has higher chances of becoming a means of payment.

24.6. Settlements

A group of banks can use assets for interbank settlements. Some of the larger banks issue fiat-pegged assets that can only be used by attested users, and only group members can be attested. The asset is backed by the issuing bank's reserves. When a smaller bank wants to settle with another smaller bank, it just sends the asset. The receiving bank can use the asset in the same way to settle with other banks, or redeem it for fiat currency with the issuing bank. The banks can also exchange USD-pegged assets for EUR-pegged assets or similar. All such transfers and trades are settled immediately, they are final and irrevocable. In SWIFT, banks exchange only information about payments, while the actual transfer of money is a separate step. In Byteball, information *is* money.

25. Private payments

So far, we have considered only payments that are sent in the open, i.e. their payloads are included inline and visible to everybody. Remember that Byteball allows the posting of private payloads: the user keeps the payload private (payload_location='none') but posts only its hash to be able to prove that the payload existed at a specific time. To apply that to payments, the sender of the funds also needs to send the private payload to the recipient via private communication channels. The recipient would need to look up the payload hash in Byteball to confirm that it existed. However, that is not enough as having concealed the payload content from other Byteball nodes we also removed their ability to verify that the same output is not spent twice. To restore this ability, we add an additional public field into the unit. This field is called *spend proof*, and it is constructed in such way that:

- it depends solely on the output being consumed, so that an attempt to spend the same output again will produce the same spend proof;
- it doesn't reveal anything about the output being spent.

It is easy to see that this construction satisfies the above requirements:

```
spend_proof = hash({
      asset: payload.asset,
      unit: input.unit,
      message_index: input.message_index,
      output_index: input.output_index,
```

```
      address: src_output.address,
      amount: src_output.amount,
      blinding: src_output.blinding
})
```

Here, payload.asset is the ID of the asset being privately transferred, input refers to the input that consumes a previous output src_output. Private outputs should have an extra field called blinding, which is just a random string designed to make it impossible to pre-image the consumed output knowing its spend proof (all the other fields come from a rather narrow set of possible values that can be iterated through within a reasonable timeframe).

The above spend proof construction applies to transfers. For issues:

```
spend_proof = hash({
      asset: payload.asset,
      address: "ISSUER ADDRESS",
      serial_number: input.serial_number, // always 1 for capped
      assets
      amount: input.amount, // issue amount
      denomination: 1 // always 1 for arbitrary-amounts payments
})
```

Note that spend proof for issue transaction does not include any blinding factor. As such it is possible to learn that a coin was issued, but the recipient of the coin is still hidden from third parties. Also, for transfer transactions, since the payer knows the blinding factor, he can calculate the spend proof that'll be published when the coin is spent. This means that he can know when the payee spends the coin, but he will not see the recipient(s) nor the new blinding factor(s) – and hence will not be able to track the coin any further.

Spend proofs are added into the unit:

```
unit: {
      …
      spend_proofs: [
            {
                  spend_proof: "the above hash in base64",
                  address: "SPENDING ADDRESS" // only if multi-authored
            },
            …
      ],
      …
}
```

Thus, to send a private payment, the sending user should:

- add a random blinding factor to each output;
- not publish the payload but send it to the payee privately, along with the hash of the unit where this payload can be found;
- for each input, add the corresponding spend proof into the unit.

All validators should reject a unit if they see the same spend proof posted from the same address again (provided that the address posts serially, of course). The payee should check that (1) the payload he received privately does hash to payload_hash posted to Byteball by the payer and (2) the spend proofs derived from private payload inputs match those included in the unit.

When a user who received a private payment wants to spend its outputs, he has to forward the private payloads he has received to the new payee, so that the new payee can verify the entire chain of ownership transfers (the history) back to the point where the asset was issued. The length of the history will grow with each transfer.

Note that with the format of payment we have considered so far, each unit can merge outputs from several previous units and produce several new outputs (most often, two). Each previous unit, in turn, depends on several even earlier units, and each output will be later split into several new outputs. Therefore, the number of future units that have at least some "blood" of the initial unit grows exponentially with time. Conversely, the number of ancestors that contribute to the unit's inputs grows exponentially with the number of steps back in history. To avoid such rapid growth of histories, we need to limit the divisibility of the coins, and this is where an asset type with fixed_denominations property set to true proves useful.

26. Fixed denominations assets

A fixed denominations asset exists as a set of indivisible unmergeable coins, very similar to the minted coins and banknotes that everybody is familiar with.

The amount of every coin must be one of a small set of allowed denominations, which should be selected so that it is convenient to represent any practical amount with maximum accuracy and the smallest number of coins. Most modern currency systems have denominations that follow a 1-2-5 pattern: 1, 2, 5, 10, 20, 50, 100, 200, 500, etc. This pattern is also recommended for fixed denomination assets on Byteball.

The coins are initially grouped into packs, similar to packs of paper banknotes. The packs can be split into smaller subpacks or individual coins, but not re-merged. This means that each transfer must have exactly one input (because merging is disallowed), and output amounts must be multiples of the coin denomination (because the denomination is the smallest indivisible amount).

Each transaction, issue or transfer, deals with coins of only one denomination. It cannot issue or transfer coins of different denominations at the same time (but each storage unit can include multiple such transactions). A fixed denominations transaction has almost the same format as a transaction with arbitrary-amounts assets, the difference being that only one input is allowed, the amounts must be multiples of one of the denominations, and a denomination field is added:

```
payload: {
      asset: "hash of unit where the asset was defined",
      denomination: 100,
      inputs: [ // exactly one input
            {
                  type: "issue",
                  amount: 1000000,
                  serial_number: 1, // always 1 for capped assets
                  address: "ISSUER ADDRESS" // only when multi-authored
            }
      ],
```

```
      outputs: [
            {
                  address: "BENEFICIARY ADDRESS",
                  amount: 800 // multiple of 100
            },
            {
                  address: "CHANGE ADDRESS",
                  amount: 999200 // multiple of 100
            }
      ]
}
```

If the asset is capped, the entire supply of each denomination must be issued within a single transaction. Thus, if the asset has e.g. 16 denominations, it'll take 16 transactions to fully issue the asset. If the asset is not capped, the serial numbers of different issues of the same denomination by the same address must be unique.

If several coins need to be issued or transferred (which is usually the case), the payer includes several such messages in the same unit. For transfers, the coin is identified by the unit, message index, and output index where it was previously transferred to the current owner.

For private payments, the payload goes separately and additionally hides the recipients of all outputs except the one that is meant for the payee:

```
payload: {
      asset: "hash of unit where the asset was defined",
      denomination: 200,
      inputs: [{
            unit: "hash of source unit",
            message_index: 2,
            output_index: 0
      }],
      outputs: [
            {
                  output_hash: "hash of hidden part of output that
                  includes address and blinding factor",
                  amount: 800
            },
            …
      ]
}
```

The information that is open in the outputs allows the recipient to verify that the sum of all outputs does match the input. The single output that is meant for the payee is revealed to him as follows:

```
output: {
      address: "BENEFICIARY ADDRESS",
      blinding: "some random string"
}
```

This enables the payee to verify the output_hash as well as construct the future spend proof when he decides to spend the output.

In Byteball, we have a private fixed denominations asset *blackbytes* that is defined by these properties:

```
{
```

```
        cap: 2,111,100,000,000,000,
        is_private: true,
        is_transferrable: true,
        auto_destroy: false,
        fixed_denominations: true,
        issued_by_definer_only: true,
        cosigned_by_definer: false,
        spender_name_attested: false,
        denominations: [
              {denomination: 1,  count_coins: 10,000,000,000},
              {denomination: 2,  count_coins: 20,000,000,000},
              {denomination: 5,  count_coins: 10,000,000,000},
              {denomination: 10,  count_coins: 10,000,000,000},
              {denomination: 20,  count_coins: 20,000,000,000},
              {denomination: 50,  count_coins: 10,000,000,000},
              {denomination: 100,  count_coins: 10,000,000,000},
              {denomination: 200,  count_coins: 20,000,000,000},
              {denomination: 500,  count_coins: 10,000,000,000},
              {denomination: 1000,  count_coins: 10,000,000,000},
              {denomination: 2000,  count_coins: 20,000,000,000},
              {denomination: 5000,  count_coins: 10,000,000,000},
              {denomination: 10000,  count_coins: 10,000,000,000},
              {denomination: 20000,  count_coins: 20,000,000,000},
              {denomination: 50000,  count_coins: 10,000,000,000},
              {denomination: 100000,  count_coins: 10,000,000,000}
        ]
}
```

Note that we have double the number of 2-denomination coins because we need them more often. For example we need two 2s for amounts 4 (2+2) and 9 (5+2+2).

Spend proofs for transfers and issues of private indivisible (fixed denominations) assets are exactly the same as for arbitrary-amounts assets, except that for issues the denomination is not necessarily 1.

Unlike divisible payments, each fixed denomination coin is never merged with other coins. Therefore when the coin is transferred privately, its history grows linearly with time rather than exponentially, and remains manageable (given that computing resources such as storage, bandwidth, and CPU power continue growing exponentially for the foreseeable future).

As the history grows, so does the exposure of private payloads to third parties who are future owners of the same coin. As discussed previously, the growth is rather slow, and the value of private payloads to adversaries arguably decreases with time. However, one should remember that large merchants and exchanges who send and receive many payments every day will probably accumulate very large (but still fragmented) histories. One should hence still avoid address reuse, even for private payments.

Note that in some cases third parties can infer important information even from private payments. For example, after most packs are already split into individual coins, when a user sends a large number of private payment messages in the same unit, an observer might argue that the user is sending coins of maximum denomination because to send an amount that is significantly larger than the maximum denomination, one would probably send multiple maximum denomination coins. From this, the observer might infer the approximate amount of the transfer (but nothing more). To avoid leaking such information, it is

recommended to spread large amounts across multiple addresses and to send them in separate units.

The spend proof approach that we have chosen is not the only one possible. To prove to the recipient that the money he receives has not been spent before, the payer could just send him all the private payloads ever sent from his address. The payee could then check each one and verify that there are no double-spends. We chose not to go this way because it involves unnecessary privacy leakage and adds complexity to the light client code. Instead, we chose to somewhat increase space usage but make the verification simpler.

27. Texts

One can store arbitrary texts using 'text' message type:

```
unit: {
      …
      messages: [
            …
            {
                  app: "text",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: "any text"
            },
            …
      ],
      …
}
```

The interpretation of the text is up to the author and his intended audience; Byteball nodes don't validate it except to check that it is a string. One could use this message type, for example, to send inerasable tweets. The payload may be private, and it can be useful, for example, for storing hashes of users' intellectual property or for storing hashes of contract texts that only a few parties need to know.

28. Arbitrary structured data

One can store arbitrary structured data using 'data' message type:

```
unit: {
      …
      messages: [
            …
            {
                  app: "data",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        key: "value",
                        another_key: {
                              subkey: "other value",
                              another_subkey: 232
                        }
```

```
                }
            },
            …
      ],
      …
}
```

The interpretation of this data is up to the author and his partners that need to see the data, Byteball nodes don't validate it except to check that it is an object. For example, this message type can be used to post Ethereum code for the subset of nodes who understand it, but remember that they cannot reject the unit even if the code is invalid by Ethereum rules.

Like 'payment' and 'text', 'data' messages can be private, in which case only its hash is stored. Continuing our Ethereum example, Ethereum contracts can be run privately if the corresponding spend proofs are also devised where necessary.

29. Voting

Anyone can set up a poll by sending a message with app='poll':

```
unit: {
      …
      messages: [
            …
            {
                  app: "poll",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        question: "Should the United Kingdom remain a
                        member of the European Union or leave the
                        European Union?",
                        choices: ["Leave", "Remain"]
                  }
            },
            …
      ],
      …
}
```

To cast votes, users send 'vote' messages:

```
unit: {
      …
      messages: [
            …
            {
                  app: "vote",
                  payload_location: "inline",
                  payload_hash: "hash of payload",
                  payload: {
                        unit: "hash of the unit where the poll was
                        defined",
                        choice: "Leave"
                  }
            },
            …
```

```
        ],
        …
}
```

Determining which votes qualify is up to the organizer of the poll. Byteball doesn't enforce anything except the stipulation that the choices are within the allowed set. For example, the organizer might accept only votes from attested users or votes from a predetermined whitelist of users. Unqualified votes would hence still be recorded, but should be excluded by the organizer when he counts the votes.

Weighting the votes and interpreting results is also up to the organizer of the poll. If users vote by their balances, one should remember that they can move the balance to another address and vote again. Such votes should be handled properly.

30. Private messaging

For private payments to work, users need a way to securely deliver private payloads to each other. Users, or rather their devices, also need to communicate to assemble signatures for multi-sig addresses.

Since we cannot expect user devices to be constantly online and easily reachable (most of them will be behind NAT), we need a store-and-forward intermediary that is always online, easily reachable, and able to temporarily store any data addressed to a user device.

In Byteball, such an intermediary is called the *hub*, and its operation is similar to email. A hub is a Byteball node that additionally offers a service of storing and forwarding private messages to connected devices. There can be many hubs. Each device that runs a wallet code subscribes to a hub of its choice, and can be reached via this hub (the home hub). The choice of home hub can be changed at any time. Each device has a permanent private key that is unique to the device. The hash of the corresponding public key (more precisely, the hash of the single-sig definition based on this public key) is called the *device address*, and it is written in base32 like the payment addresses. The full device address, including its current hub, can be written as DEVICEADDRESSINBASE32@hubdomainname.com. If the device moves to another hub, the part of its full address before @ stays the same. Unlike email, the name cannot be already "taken".

Every device connects to its home hub using websockets. The hub sends the new messages to the device and the device stays connected to the hub, so that if a new message arrives while the device is connected the new message is delivered immediately. The hub doesn't keep copies of the messages that were successfully accepted by the device. The connection to the hub is TLS encrypted.

When a device wants to send something to another device, it connects to the recipient's hub and sends the message. Unlike email, there is no relay – the sender connects directly to the *recipient's* hub. All communication between devices is end-to-end encrypted and digitally signed so that even the hub (who is the only man in the middle) cannot see or modify it. We use ECDSA for signing and ECDH+AES for encryption.

Before exchanging encrypted messages the devices must be paired, i.e. learn each other's public key. This can happen in various ways, e.g. by scanning a QR

code that encodes the public key and hub domain name of one of the devices, by sending this information over email, or by clicking a byteball:// link on a secure website.

For forward security, every device generates a temporary private key and uploads the corresponding public key to its home hub. Afterwards, the device rotates the key from time to time but keeps a copy of the previous key in case someone sent a message to the previous key while the hub was replacing it. The hub keeps only one version of the temporary public key per subscribed device. The sending device follows these steps to send a message:

1. connects to the recipient's hub;
2. receives the current temporary public key of the recipient from the hub;
3. generates its own one-time ephemeral key pair;
4. derives ECDH shared secret from the recipient's temporary public key and own ephemeral private key;
5. AES-encrypts the message using this shared secret;
6. adds its own ephemeral public key;
7. signs the package with its own permanent key; and
8. sends it to the hub.

The recipient device verifies the signature, derives ECDH secret using the peer's ephemeral public key and own temporary private key, and decrypts the message.

If the sending device fails to connect to the recipient's hub, it encrypts the message to the recipient's permanent key (this encryption is not forward secure since it uses a permanent key) and stores the encrypted message locally for future retries. The purpose of this encryption is to avoid having unencrypted messages lying around. After connection to the recipient's hub succeeds, the device sends this encrypted message, thus encrypting it again (this time, with forward security), so the message is double-encrypted. Note that this is not because single encryption is insufficient, but because we don't want to store unencrypted content for an indefinite time while the connections are retried.

Note that the communication is among devices, not users. Users may (and are recommended to) hold several devices, such as a laptop, a smartphone, and a tablet, and set up multisig addresses with redundancy (such as 2-of-3) that depend on keys stored on multiple devices. When a user needs to sign a transaction, he initiates it on one of his devices. This device then sends the partially signed transaction to the other devices using private messages, collects all the signatures, and publishes the transaction. The private keys stored on each device should never leave that device. When the user replaces one of his devices in a 2-of-3 address, he just uses the other 2 devices to change the address definition and replace the key of the old device with the key of a new device.

The private messages can also be used for encrypted texting between devices. These messages are strictly peer-to-peer, never go into the Byteball database, and can be safely discarded after they are read.

When users pay in blackbytes or other private assets, they have to send private payloads and absolutely need devices that can communicate. They need to know each other's *device* addresses before they even learn each other's *payment* addresses. Once their devices have established communication, the payee can

send his payment address to the payer via chat message. Such a payment scenario also makes it easy to generate a unique payment address for every incoming payment. A merchant can run a chat bot that communicates with users via text messages. When the user is ready to pay the bot generates a new payment address and sends it to the user in a chat message.

31. Conclusion

We have proposed a system for decentralized immutable storage of arbitrary data, including data of social value such as money. Every new unit of data implicitly confirms the existence of all previous units. Revision of past records similar to that in *1984* becomes impossible, as every new unit also implicitly protects all previous units from modification and removal. There is an internal currency that is used to pay for inclusion of data in the decentralized database. The payment is equal to the size of the data to be stored, and other than this payment there are no restrictions on access to the database. Other assets can also be issued and their ownership can be tracked on the database. When tracking payments in the internal currency and other assets, double-spends are resolved by choosing the version of history that was witnessed by known reputable users. Settlement finality is deterministic. Assets can be issued with any rules that govern their transferability, allowing regulated institutions to issue assets that meet regulatory requirements. At the same time, transfers can be hidden from third parties by sending their content privately, directly from payer to payee, and publishing spend proofs to ensure that each coin is spent only once.

References

1. Quoted from Wikipedia https://en.wikipedia.org/wiki/Nineteen_Eighty-Four.
2. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, https://bitcoin.org/bitcoin.pdf, 2008.
3. Sergio Demian Lerner. DagCoin, https://bitslog.files.wordpress.com/2015/09/dagcoin-v41.pdf, 2015.
4. Serguei Popov. The Tangle, http://iotatoken.com/IOTA_Whitepaper.pdf, 2016.
5. TomHolden. Transaction-Directed Acyclic Graphs, https://bitcointalk.org/index.php?topic=1504649.0, 2016.
6. Linked timestamping, https://en.wikipedia.org/wiki/Linked_timestamping.
7. Atomic cross-chain trading, https://en.bitcoin.it/wiki/Atomic_cross-chain_trading.
8. https://github.com/bitcoin/bitcoin
9. Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, http://gavwood.com/Paper.pdf.