# Vegvisir Blockchain Formalized

Robbert van Renesse

Department of Computer Science, Cornell University

## 1 Vegvisir Design

Perhaps the most important concept to understand in Vegvisir is that of a partially ordered set (poset). A poset is a pair $(S, \preceq)$ where $S$ is a set of elements and $\preceq$ is a binary relation between elements of $s$ such that $\forall s, s', s'' \in S$:

- *reflexivity*: $s \preceq s$
- *antisymmetry*: $s \preceq s' \wedge s' \preceq s \implies s = s'$
- *transitivity*: $s \preceq s' \wedge s' \preceq s'' \implies s \preceq s''$

A *Directed Acyclic Graph* (DAG) induces a poset in a natural way, although typically a poset is diagrammed using a *Hasse Diagram*, which is essentially is a DAG with all transitive edges removed.

Two elements $s$ and $s'$ of $S$ are called *comparable* if either $s \preceq s'$ or $s' \preceq s$, and *incomparable* otherwise. A subset $S' \subseteq S$ forms a *cut* of $(S, \preceq)$ if any two elements of $S'$ are incomparable. If $s \prec s'$ (shorthand for $s \preceq s' \wedge s \neq s'$), we will say that $s$ *precedes* $s'$. We say that $s$ is a *parent* of $s'$ if $s \prec s'$ and there is no element $s''$ such that $s \prec s'' \prec s'$.

In Vegvisir, the poset concept shows up in at least three places:

1. its transactions;
2. its blocks;
3. its layered architecture.

### 1.1 Vegvisir Transactions

A *transaction* encodes an *operation* on some object. (One may distinguish multiple objects within the object, but for simplicity we only consider a single object here.) An operation may mutate the state of the object and return a deterministic result. Operations are immutable and form a poset. We require that any cut in this poset uniquely identifies the state of the object. This is related to *Conflict-Free Replicated Data Types* (CRDTs) but more general in that a CRDT is a special case in which no two transactions are comparable. A transaction is uniquely identified by its operation and its set of parents.

An important feature of Vegvisir transactions is their durability. A Vegvisir transaction cannot become less durable over time, but its durability may increase. Currently, durability may be expressed as the set of devices that are known to store a copy of the transaction. We call those devices the *witnesses*. Applications (should) have access to the level of durability of transactions.

## 1.2 Vegvisir Blocks

A block contains an ordered list of transactions. Like transactions, blocks are immutable, have a level of durability, and form a poset. (Essentially, a block aggregates a set of transactions.) Going further, in an instantiation of a Vegvisir blockchain, there is a unique block $g$ that precedes all other blocks in the blockchain. $g$ is called the *genesis block*. Note that a Vegvisir blockchain instantiation therefore forms a semilattice.

[not sure about any of the following]

If participants are all correct, $t \leq t' \implies t \preceq t'$ (but not necessarily vice versa). We require that for two transactions $t$ and $t'$ and two blocks $b$ and $b'$, $t \in b \wedge t' \in b' \wedge t \preceq t' \implies b \preceq b'$, that is, the ordering among blocks has to be consistent with the ordering among transactions. Moreover, the same transaction cannot appear more than once—it is thus possible to identify a transaction by the block that it is in and its index in the block. Also, the durability of a transaction is the same as the durability of the block that it is in.

## 1.3 Vegvisir Architecture

A natural way to deal with complexity in software design is through *layering*. Each layer implements an *abstraction* and typically supports instantiating the abstraction multiple times. An abstraction is usually an *object* with an *interface* and a *specification* that describes how that interface behaves. Different layers may implement the same abstraction. As the objects are typically distributed in nature, their implementations are usually through *protocols*.

Layers form a poset: one layer $L$ may use one or more other layers to implement the abstraction that layer $L$ implements, but there can be no circular dependencies. If $L \preceq L'$, then $L'$ uses $L$ in its implementation. Abstraction dictates that the implementation of $L'$ should only access $L$ through its interface in accordance with $L$'s specification, but should not know anything about its implementation. Through its specification, $L$ is allowed to limit how $L'$ or other layers accesses its interface, but in general should not know anything about those layers. In other words, a layer's interface should be *generic*.

The Vegvisir architecture includes the following layers:

1. each application is a layer in its own right. An application typically has no other layers that use it, but this is not necessary.

2. the *pub/sub layer*: a layer that supports topics, publishing transactions to topics, and subscribing to topics, indicating a desire to receive transactions published to those topics. Transactions are delivered to the subscriber in an order consistent with the transactions poset, that is, a transaction is never delivered until all transactions that precede it are delivered.

3. the *block layer*, implementing a poset of blocks;

4. various *block reconciliation layers*, each implementing an alternative way of reconciling block posets on different devices;

5. various network layers, providing low-level networking support between devices for the block reconciliation layers.

The enumeration above is consistent with the partial ordering among the layers: layers at a particular item are only allowed to use layers at higher numbered items in their implementation.

## 2 Vegvisir Data Types

Vegvisir defines a variety of types: messages, blocks, transactions, operations, and many more. Many of these data types are *container types*, and can contain other generic data types: a message may contain a block, a block may contain a transaction, and a transaction may contain an operation. Many data types are defined within a layer and are only intended for use within that layer; other data types are intended for general use and appear in the interfaces of layers.

Like layer interfaces, data types should be generic: a message can contain different kinds of blocks, a block can contain different types of transactions, and a transaction can contain different kinds of operations. Moreover, none of the data types should have to anticipate what data types it might contain in the future.

In Vegvisir we use Google *Protobuffers* or *protobufs* for short to define Vegvisir data types. Unfortunately, while protobufs are efficient for serialization, the protobuf definition language does not support generics. One common solution around this problem, widely used in the internet, is to do iterative serialization: if data type $d$ can contain a variety of other data types, then define $d$ to have a *byte buffer* and serialize the other data types into this byte buffer. A serialized value of any data type can be included in any container data type.

This begs the question how, upon learning $d$, one should go about deserializing the byte buffer. Protobuf requires that the receiver knows the type of what is being deserialized. The two most commonly used solutions are that either the recipient of $d$ already knows what data type is serialized in the byte buffer (e.g., though a prior handshake protocol), or $d$ also contains an integer that identifies the data type. The latter requires some convention on how these integers are used, but in general this convention should *not* be described by $d$, or one risks that $d$ is no longer truly generic.

(Another possible solution to the lack of generics in protobufs is to define a recursive *Value* data type, but this makes programming with those data types awkward and inefficient.)

# 3 Vegvisir Transaction Graph

As mentioned above, transactions form a poset. One problem is that devices may be Byzantine and try to form circular transaction dependencies. To prevernt this, a transaction is identified by a cryptographic hash function.

A hash function $\mathit{hash} : \mathcal{O} \rightarrow \mathcal{H}$ maps an object in $\mathcal{O}$ to a unique identifier in $\mathcal{H}$. It has the usual properties of cryptographic hash functions:

- it is straightforward to compute $\mathit{hash}(o)$ for any $o \in \mathcal{O}$;
- given a $h \in \mathcal{H}$, it is computationally infeasible to find an $o$ such that $h = \mathit{hash}(o)$ (i.e., *hash* is hard to invert);
- given $h = \mathit{hash}(o)$, it is computationally infeasible to find an $o'$ such that $h = \mathit{hash}(o')$ (i.e., *hash* is *collision-free*);

In this case, $\mathcal{O}$ is the set of transactions. A transaction consists of some opaque description of an operation and a set of transaction identifiers, that is, hashes of parent transactions. Because of the properties of hash functions, it is (computationally) impossible to create cycles in the transaction graph.

# 4 Vegvisir Block Graph

The system is comprised of a set of devices $\mathcal{D} = \{d_0, ...\}$. Some devices are correct and follow the specification below, but others may be *Byzantine* and depart from the specification. We denote by $\mathcal{D}^o \subseteq \mathcal{D}$ the set of correct devices.

A *block* $b \in \mathcal{B}$ is an immutable object with the following fields:

- $b.\mathit{owner} \in \mathcal{D}$: the creator of the block;
- $b.\mathit{txs}$: a set of transactions;
- $b.\mathit{parents} \in 2^{\mathcal{H}}$: a set of hashes of parent blocks.
- $b.\mathit{signature}$: an unforgeable digital signature from the owner of this block.

Let $B \subseteq \mathcal{B}$ be a set of blocks. We call $B$ *complete* iff

- $\forall b \in B : \forall h \in b.\mathit{parents} : \exists b' \in B : \mathit{hash}(b') = h$

Note that a complete set of blocks and their parent relationships induce a *block graph*. We postulate that by the cryptographic properties of the hash function, a finite block graph cannot have cycles in it and therefore forms a Directed Acyclic Graph (DAG). We also observe that the union of two complete sets is a complete set (but not the intersection). We say that $b \leq b'$ if there is a path from $b'$ to $b$. We also say that $b'$ *depends on* $b$.

We call a device $d$ *well-behaved* in $B$ iff

- $\forall b, b' \in B : b.owner = b'.owner = d \implies b \leq b' \vee b' \leq b$

A *Vegvisir Block Graph* (VBG) $B$ has the following properties:

1. $B$ is a complete set of blocks;
2. There is a unique block $g \in B$ such that for all $b \in B : g \leq b$. We call $g$ the *genesis block* of $B$;
3. $\forall d \in \mathcal{D}^o$: $d$ is well-behaved in $B$.

We denote by $\mathcal{VBG}$, $\mathcal{VBG} \subset \mathcal{B}$, the set of of all VBGs.

We call two VBGs with the same genesis block *related*. Note that the union $B_1 \cup B_2$ of two related VBGs $B_1$ and $B_2$ is also a VBG and related to both $B_1$ and $B_2$.

Given a block $b$ in a VBG, we define *ancestry*$(b)$ to be the set of blocks consisting of $b$ and its ancestors all the way to the genesis block, that is:

$$\mathit{ancestry}(b) \triangleq \{b' \in B \mid b' \leq b\}$$

Ancestry is the transitive closure of the parent relationship. Note that *ancestry*$(b)$ is a VBG that is related to $B$, and *ancestry*$(g) = \{g\}$. Also,

$$\forall b' \in \mathit{ancestry}(b) : \mathit{ancestry}(b') \subseteq \mathit{ancestry}(b)$$

Note that *hash*$(b)$ uniquely identifies *ancestry*$(b)$.

We define a function $\mathit{height} : 2^{\mathcal{VBG}} \rightarrow \mathcal{D} \rightarrow \mathbb{N} \cup \{\bot\}$ as follows:

- if $d$ is well-behaved in VBG B, then $\mathit{height}(B, d) = |\{b \in B \mid b.owner = d\}|$, that is, the number of blocks that $d$ owns in B.
- if $d$ is not well-behaved in $B$, then $\mathit{height}(B, d)$ is $\bot$.

We will call *height*$(B)$ the *vector timestamp* of $B$. Also, for each well-behaved device, we will call the most recent block added by $d$ in $B$ the *leader block* of $d$.

# 5 Authorization

We are now going to constrain which devices are allowed to add blocks to a VBG $B$ by constraining the set of parents a block signed by a particular device is allowed to have. We define two special transactions `delegate(d)` and `revoke`$(d)$. We say that a device $d$ is authorized for block $b$ iff both the following conditions hold:

- $\exists b' : b' \leq b \wedge \texttt{delegate}(d) \in b'.txs \wedge (b'.owner \neq d \vee b'.parents = \emptyset)$

- $\forall b' : b' \leq b \implies \texttt{revoke}(d) \notin b.txs$

(Note that only a genesis block can, and should, delegate to its owner.) We call a block *legal* if its owner is authorized for the block. A VBG $B$ then has the additional constraint that all blocks owned by correct devices be legal and only depend on legal blocks.

A block $b$ that is not legal is proof-of-misbehavior of $b.owner$, as are blocks that depend on $b$. We modify the *height* function so that $height(B, d) = \perp$ if $B$ contains a block owned by $d$ that is not legal or depends on a block that is not legal.

# 6 Reconciliation

Each correct device $d \in \mathcal{D}^o$ maintains a set of blocks $d.blocks$. The Vegvisir system maintains the following invariants:

- For each correct device $d \in \mathcal{D}^o$, $d.blocks$ is a VBG;
- For each correct device $d \in \mathcal{D}^o$, $d.blocks$ can only grow over time;
- For any two correct devices $d_1$ and $d_2$, $d_1.blocks$ and $d_2.blocks$ are related (i.e., they have the same genesis block).

The object of reconciliation is for some group of correct devices $D \subseteq \mathcal{D}^o$ and each device $d \in D$, to replace $d.blocks$ with $\bigcup_{d' \in D} d'.blocks$.

Usually the cardinality of $D$ will be two. A simple reconciliation protocol, when there are two devices $d$ and $d'$ could be as follows:

1. $d$ sends vector timestamp $H = height(d.blocks)$ to $d'$. Optionally $d$ can include the maximum number of blocks that $d$ is willing to receive in return.
2. $d'$ uses $H$ to determine $d'.blocks \backslash d.blocks$ (the blocks that $d'$ has but $d$ does not). If this is the empty set, the remainder of this protocol is skipped.
3. $d'$ streams the blocks to $d$. $d'$ sends the blocks in an order so that, upon arrival of a block, $d$ can add the block to $d.blocks$ while maintaining the invariant that $d.blocks$ be complete. (TODO: we should prove that such an order always exists.)
4. $d$ adds each block to $d.blocks$, after checking that adding the block does not violate the invariant that $d.blocks$ is a VBG.
5. After receiving the last block, $d$ determines the set $D^o$ of devices that are well-behaved in $d.blocks$. $d$ then creates a new block $b$ (with possibly an empty set of transactions if none are pending) with the last blocks of the devices in $D^o$ as its parents. $d$ adds $b$ to $d.blocks$ and sends $b$ to $d'$.

6. After receiving $b$, $d'$ checks to make sure that adding $b$ to $d'.blocks$ maintains the VBG invariant and, if so, adds the block to $d'.blocks$.

This protocol can be executed in the opposite direction concurrently. It can also safely be terminated at any time, for example if a network connection breaks. TODO: we have to prove that this protocol satisfies the invariants listed above and results in the VBGs of all devices converging to the same set.

# 7 Witnesses

Let $b$ be a block. A correct device $d$ is said to *witness* a block $b$ if $b \in d.blocks$. Given a set of correct devices $D$, the set of blocks that are witnessed by all devices in $D$ are characterized by $\min_d height(d.blocks)$.

The concept of witness is important in determining which blocks are durable in Vegvisir. Therefore, devices collect signed vector timestamps from the other devices. Note that in Step 1 of the reconciliation protocol described above, the devices already send timestamps to one another. We now require that these be signed. In addition, the signed vector timestamps may be gossiped between devices, that is, the sets of signed vector timestamps should be reconciled as well.

Each device keeps track of the set of vector timestamps received from its peers. For any two vector timestamps issued by a correct device, one will dominate the other, so peers only need to keep the maximum. If a device receives timestamps from the same peer that do not dominate one another, that is a proof-of-misbehavior, and the device can simply mark the peer as faulty. The set of vector timestamps form a square matrix with a row and column for each device.