**The Blockdag Abstraction**

* A *block* is a public key certificate that consists of:
  1. a header, itself consisting of:
     + identifier of creator of this block.
     + list of hashes of parent blocks.
     + a body, consisting of a possibly empty list of *transactions*. Each transaction is a pair (topic, event).
  2. a signature by the creator over the header and the body.
* A block is uniquely identified by a hash over its contents.
* There is one unique vertex that is designated the *genesis block*. The genesis block is the only block without parent hashes.
* A *blockdag* is a DAG with the vertices made up of blocks and the edges made up of parent hashes, with the following constraints:
  1. The blockdag contains the genesis block and it is the unique sink vertex;
  2. There exists a path from every block to the genesis block;
  3. There exists no edge b1 -> b2 if there is another path from b1 to b2;
  4. There is a unique source node that we call the *leader block*.
* The hash of the leader block of a blockdag uniquely identifies the blockdag.
* A blockdag can be extended by atomically adding a new set of blocks, one of which has to serve as the new leader block.
* Any block in a blockdag can be considered leader of a *sub-blockdag* consisting of that block and its ancestors;
* Blockdag B *dominates* blockdag B’ if the leader of B’ is a block in B. We call two blockdags *concurrent* if neither dominates the other.
* Concurrent blockdags can be *reconciled* by taking the union of the vertices and the edges and adding a new leader block that points to the leaders of the blockdags (but there may be other ways). If B dominates B’, then reconciling B and B’ results in B. The reconciled blockdag of a set of blockdags dominates all the blockdags.
* Each node maintains a blockdag. A correct node only ever extends its blockdag by (atomically) adding one or more blocks. The node can either create a new leader block on its own, or reconcile with peer nodes.
* Different nodes have different blockdags that are only guaranteed to have the same genesis block. Nearby nodes always try to reconcile their blockdags.
* The *witness set* for a block b in a blockdag is a set of identifiers formed by the creators of blocks from which block b can be reached in the blockdag (including the creator of block b itself).

**The Blockdag API:**

* Let *bh* be the hash of a block, uniquely identifying the block
* Let B be a blockdag object of a node
* b = B.get\_contents(bh) returns the block b if bh identifiies a block in B. b is a byte string that consists of a header, contents, and signature as described above.
* bh = B.get\_leader() returns the hash of the leader block of B. The leader block changes over time as new blocks get added to B. From the leader block, all blocks currently in B can be explored up to the genesis block through parent hashes.
* bh = B.add\_block([tx]) adds a new block to the blockdag with the given list of transactions and returns the hash of the new block. (If there is no concurrent merging activity, the new block will simply be the new leader block of B.)
* B.add\_listener(listener) registers a listener object with B. Every time a new block is added to B, listener.new\_block(bh) is invoked with the hash to the new block. If multiple blocks are added simultaneously, new\_block() is invoked in an order that is consistent with the partial order induced by the blockdag, with the new leader block always being reported last.
* The following methods are added for convenience:
  + [bh’] = B.get\_parents(bh) returns just the list of hashes of parent blocks of the block identified by bh
  + [tx] = B.get\_transactions(bh) returns just the list of transactions in the block identified by bh.
  + bh = B.get\_genesis() returns the hash of the unique genesis block of B.
  + [bh’] = B.get\_children(bh) returns the list of hashes of children of a block identified by bh. The list of children of a block may change over time.

**Notes on implementation of blockdag reconciliation**

When nodes are nearby, they engage in a *reconciliation protocol*. Such a protocol will generally consist of the block exchanging the hashes of their leader blocks. If they are identical, the protocol can stop right there. If one dominates the other, then the dominating node can simply send the missing blocks to the other node. If they are concurrent, then a more complicated (recursive) reconciliation is required that will result in a new leader block. During the execution of reconciliation, a node learns about new blocks one by one. However, it should only report blocks that are reachable from a leader block, i.e., a block from which all other blocks are reachable.

For example, suppose that there are two nodes N1 and N2. N1 has blockdag G <= X1 and N2 has blockdag G <= X2, where G is the shared genesis block. At some point N2 learns about X1. However, it should not yet report X1 to the listener because X2 is not reachable from X1. Instead, N2 first has to create a new leader block L from which both X1 and X2 are reachable. Then N2 can report first X1 and the new leader block L. Thus, X1 and L appear as if they were atomically added to the blockdag of N2. Similarly, L atomically became the next leader block of N2 after X2.

**The Pub/Sub abstraction, API, and implementation**

On top of the blockdag abstraction, nodes offer a simplified publish/subscribe abstraction. There is a list of topics, and a node can subscribe to topics and publish events to topics. The interface/implementation is something like this:

* prepare(topic, event): create a new (topic, event) transaction, but do not yet publish it
* flush(): add one or more blocks with prepared transactions. The order of the transactions in the blocks should correspond to the order of preparation.
* subscribe(topic, listener). listener(event) is invoked each time a new event is received for the given topic. events are posted in a total order that is consistent with the partial order of the events in the blockdag.

We will provide CRDT libraries so that applications can update CRDTs when they receive events. Note that each block in a blockdag has a well-defined CRDT state.

**Authorization (Validation)**

So far we have ignored authorization and assumed that anybody can add any block with any transactions. However, we need an authorization mechanism, which in turns implies that we need an authentication mechanism. We tacitly referred to this already as blocks have owners and signatures. There is one predefined topic “Users” that is a 2P CRDT set of public key certificates. The genesis block contains transactions that add at least the self-signed public key certificate of the owner ot the genesis block. Only “Users” can sign blocks. That is, in order for a node N to be able to add a block with a certain set of parents, N has to be a User in each of the parents.

Applications can introduce additional authorization required to publish events. For example, suppose we create a 2P set representing a topic “shopping list”. The 2P set has operations to add and remove items on the shopping list. We may want to allow certain users to only add items and other users to only remove items. In order to enable this, we can introduce authorization operations to any topic, publishing authorization and de-authorization events in the form of certificates. To create a new topic, a user (a Vegvisir node) would create a self-signed certificate containing the name “shopping list” and a public key. The topic string used in transactions would be the hash of the certificate, and the first publication would contain this certificate along with a signed list of authorization operations (who is allowed to publish what).