**Concept**

In order to implement a state channel, one has to use the following Cell class.

abstract class Cell[M[\_]: Monad, F[\_]: Traverse, A, B, C](

data: A,

algebra: AlgebraM[M, F, B],

coalgebra: CoalgebraM[M, F, C]

) extends Topos[A, B] {

protected def hyloM(implicit input: A => C) = scheme.hyloM(algebra, coalgebra).apply(input(data))

}

**There are following type parameters:**

The very first type M[\_] is required to be a Monad. State channels can execute some side effects and due to that fact it needs to encapsulate these effects inside a monad.

The second type F[\_] is required to be a Functor (Traverse). This is a structure used for executing state channel recursion. For **L0 consensus** and **L1 consensus** implementations, it's a StackF case class which is explained below.

A is a data holder. To create a Cell one has to provide a data which will be consumed by the state channel. These data are stored inside the Cell for later execution.

B is the algebra output (see: [Catamorpism and algebra](https://constellation-labs.github.io/tessellation/recursion-schemes/#Catamorphism and algebra))

C is the coalgebra input (see: [Anamorphism and coalgebra](https://constellation-labs.github.io/tessellation/recursion-schemes/#Anamorphism and coalgebra))

**StackF**

The common stack is a well-known idea. In the context of an execution chain, the stack consists of two things: either some work to do More(work) or a computed result Done(result).

StackF is an implementation of that idea using recursion schemes:

trait StackF[A]

case class More[A](a: A) extends StackF[A]

case class Done[A](result: Either[CellError, Ω]) extends StackF[A]

where the job to do is parametrized by type. The result has strict type Either[CellError, Ω] as the output from the state channel is either an error of computation or a terminal object.

The following idea allows to write an algebra and coalgebra working at a single step of state channel and then repetitively execute the Cell passing output of the first execution as input to the second execution, until Done(result) is returned.

**Algebra and Coalgebra for StackF structure**

It is pretty straightforward to write an implementation of StackF scheme. Most of the parts are common between state channels so ideally this will be extracted and used as a plug-in code.

Algebra is used for proxying inner value work or result from types More(work)/Done(result) and passing it to the output. Coalgebra is a termination condition for the execution chain - it reacts on a single or multiple classes to return Done class. For every other type it executes repetitively the inner algebra and coalgebra for single steps.

**Inner Algebra and Coalgebra - for single execution step**

This part is responsible for the core logic of a state channel.

Coalgebra consists of all steps used by a Cell and holds the execution order. Algebra is a fold operation which reduces the ending type to the output terminal object.

**L0 State Channel - snapshots**

**Idea**

tbc

**Implementation**

**Step structure**

The input for the state channel is a stream of blocks. It is computed by an initial pipeline to create a final edge of blocks.

The input for the Cell is an L0Edge with checkpoint blocks.

case class L0Edge(blocks: Set[L1Block]) extends Ω

The output of the Cell is a Snapshot class holding all the blocks provided in the L0Edge.

case class Snapshot(blocks: Set[L1Block]) extends Ω

**Pipeline**

This is for now the most complex part of an L0 implementation. The idea behind this pipeline is converting an initial stream of blocks into the final edge that can be consumed by the Cell. Stream of blocks is unordered, so the pipeline must aggregate blocks within height ranges and push the Edge when the snapshot condition is met.

Snapshot condition is a formula allowing the pipeline to emit an edge for the next snapshot: tip - tipDelay >= lastEmittedHeight + heightInterval, where tip is a minimum height of all parents used to create blocks, tipDelay is the initial delay before creating the very first snapshot, lastEmittedHeight is a height of the previous snapshot and heightInterval is the range of snapshot.

The initial pipeline can be reduced to the following pseudocode which works as a more complex scan operator:

case Some((Left(tip), tail)) if tip - tipDelay >= lastEmittedHeight + heightInterval =>

// get cached blocks and output L0Edge with these blocks

case Some((Left(tip), tail)) =>

// increment minimum tip height

case Some((Right(block), tail)) =>

// add block to cache under its height

**Inner Algebra and Coalgebra**

Algebra reacts on SnapshotEnd(blocks) class and folds all the blocks to the single Snapshot class. It also reacts on L0Error class to produce a CellError class.

Coalgebra takes CreateSnapshot(edge) where the edge is a Set of blocks prepared in the initial pipeline and produces SnapshotEnd class.

**StackF Algebra and Coalgebra**

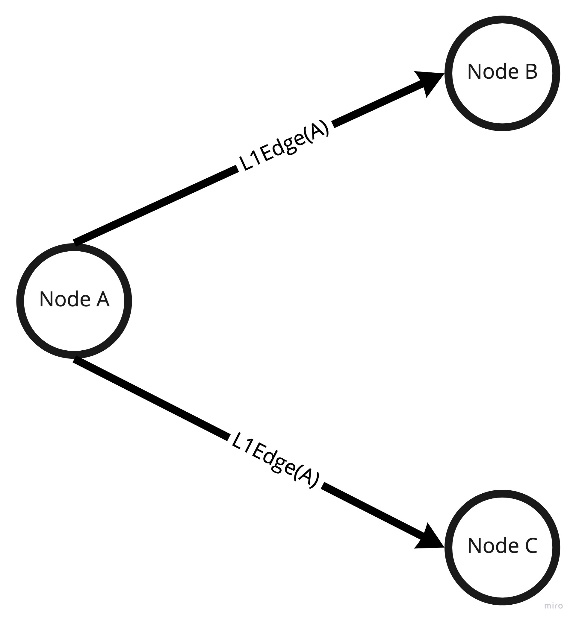
When the inner algebra produces a Snapshot or L0Error, it must end the computation, so here is the conversion to Done(result) class. For every other case it executes the inner hylomorphism over and over again.

# L1 Protocol

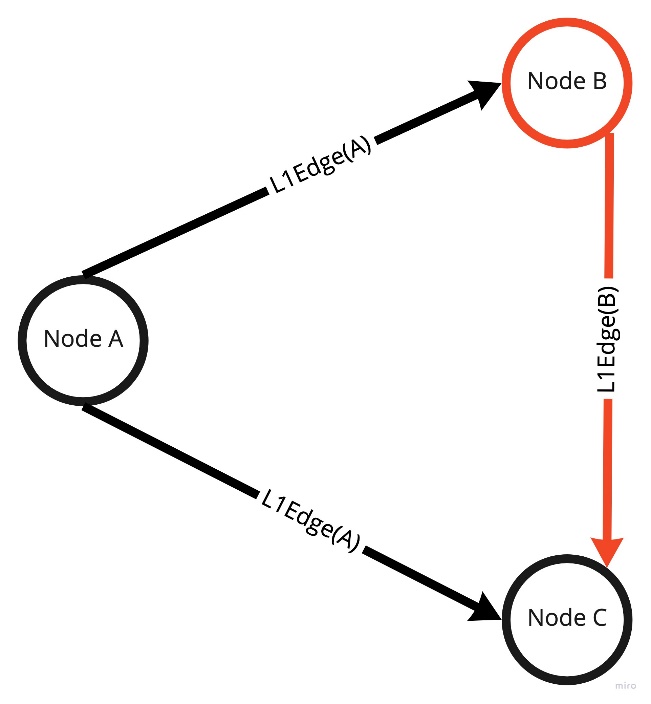
## Concept explanation

L1 is the inner layer of the L0 protocol. The main purpose of that part is to form a tetrahedron of edges during consensus process and produce a block which is then consumed by the L0 layer.

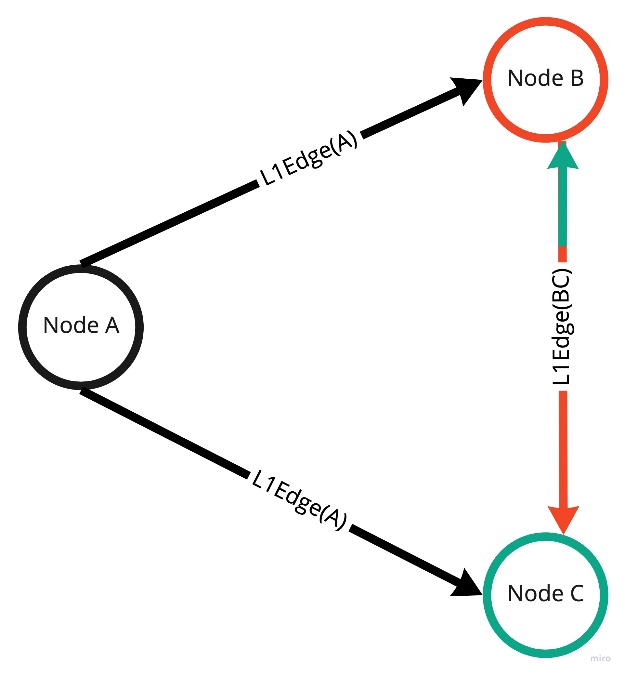
Base input for L1 protocol is an L1Edge formed from a set of L1Transaction (L1Edges are created continuously over time as the process is implemented as Pipe[IO, L1Transaction, L1Edge). The consensus owner selects 2 facilitators and forms L1Edge(A) with each facilitator. L1Edge(A) stands for L1Edge which contains all transactions proposed by Node A. By forming L1Edge Node A sends proposals and forces Node B and Node C to participate in consensus round.



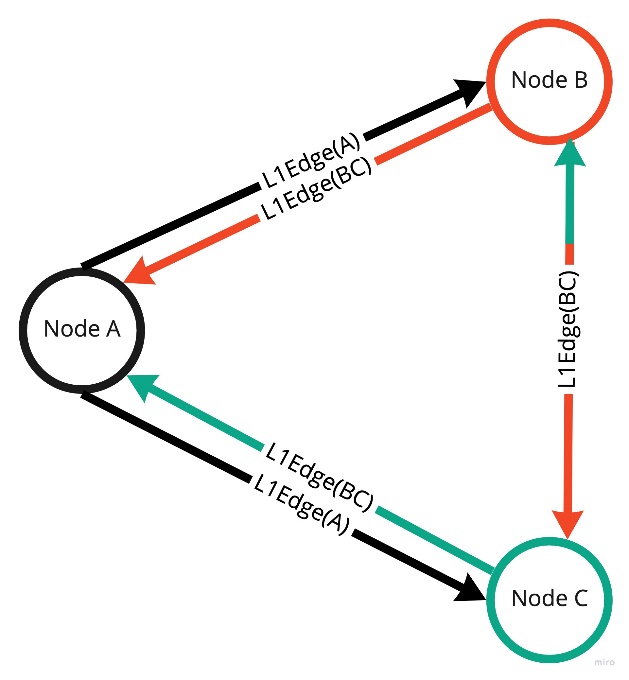
Depending on which edge is formed faster either Node B or Node C starts forming a triangle (first dimension of a tetrahedron). Let's assume that L1Edge from Node A to Node B was formed first. In that case Node B is forced to create another L1Edge(B) with Node C. It also means that Node B proposes transactions to Node C.



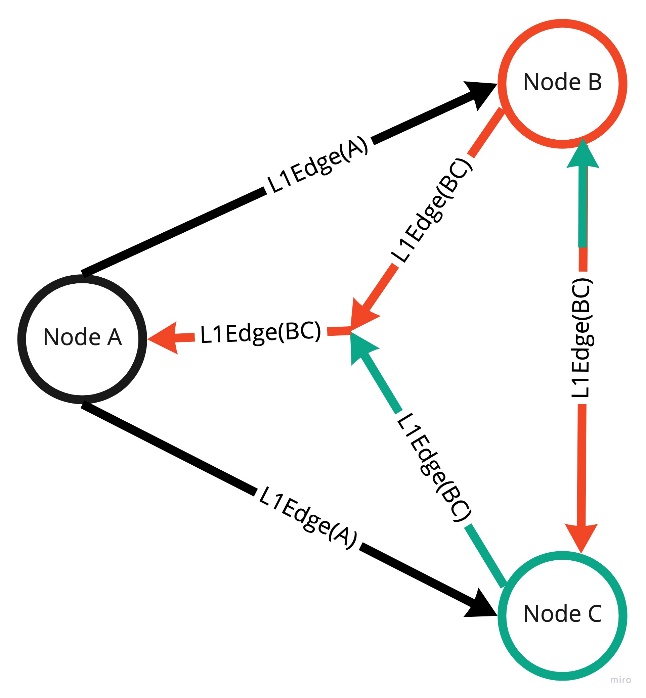
In the meantime, edge formed from Node A to Node C forced Node C to ask Node B for proposal which means that Node B and Node C need to exchange proposals between each other bidirectionally. Such structure starts forming the second dimension of a tetrahedron. The whole process is asynchronous so forming edges to Node B and Node C happens in parallel.



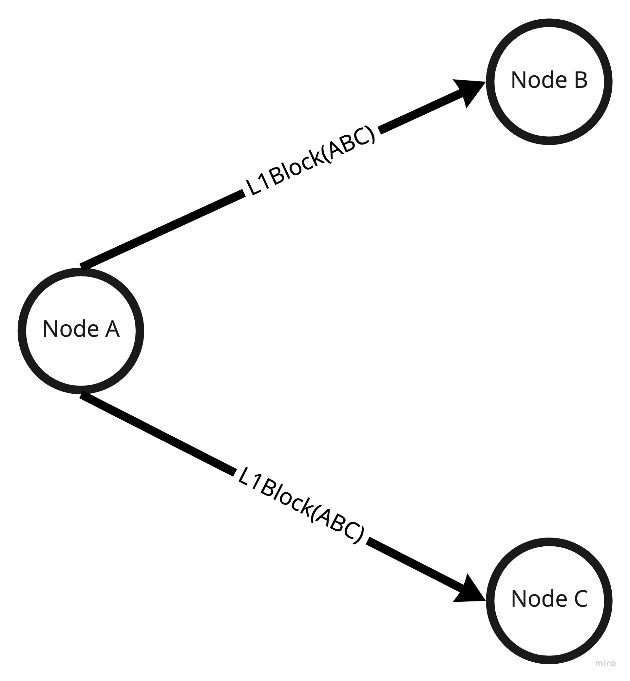
Once Node B and Node C exchanged the proposals then these nodes need to send accumulated proposals back to the consensus owner (Node A). That being said both facilitators form L1Edge back to Node A which forms a complete structure of 2 triangles.



The role of Node A is to validate if L1Edge from Node B and Node C contains the same transactions inside. If so then the consensus owner can confirm that facilitators exchanged proposals properly. That being said both L1Edges are equal, so it forms a single edge back from facilitators and forms a tetrahedron.



The last step is to create a block by the consensus owner and send it to facilitators and other nodes in the network.



## Implementation explanation

The L1 protocol is based on the concept of recursive structures and [droste](https://github.com/higherkindness/droste) library or to be more specific on [hylomorphism](https://en.wikipedia.org/wiki/Hylomorphism_(computer_science)). All the steps needed to perform a consensus are implemented via CoalgebraM[StateM, L1ConsensusF, Ω]. Each step is a unique type of recursive scheme so the whole process is just a recursive structure of steps which is then executed by folding the structure via AlgebraM[StateM, L1ConsensusF, Either[CellError, Ω]] .Coalgebra and Algebra are needed to execute a hylomorphism scheme.hyloM(L1ConsensusStep.algebra, L1ConsensusStep.coalgebra). Technically speaking hylomorphism is just a catamorphism after anamorphism. That being said anamorphic part creates (unfolds) the structure and catamorphic part runs (folds) that structure to create a block. By traversing over the structure of steps it executes the steps and folds to either L1Block or CellError in case of failure.