1. overview

Bootstrapping is the process by which a node sync's up to the network's current state. For example, if a node is brand new, it needs to get the correct database from its peers as it starts off with no data. The process by which it gets these blocks is called bootstrapping. Another use-case is when we fall out of sync with the network. For example, when a given node crashes or needs to be restarted, it has to sync up with the network from the current state once it comes back online.

This document will discuss the design of the bootstrapping module for Logos.

2. execution concept

The bootstrapping approach taken by Logos is to update batch state blocks, micro blocks and epoch blocks. We do not update using accounts or individual state blocks. All the blocks are serialized/deserialized instead of being copied. This gives us the advantage of serializing only what we need for the batch state blocks increasing performance of the system.

Bootstrapping is initiated in an alarm, and starts off by getting the information needed to decide if its ahead or behind. This information, referred to as tips represents the last blocks in the database for a given chain. We decide if we are ahead by checking sequence numbers on the peer. In the case where the bootstrap node is actually ahead of the bootstrap peer, the sequence numbers of the peer will be less than those of the bootstrapping node. Once this is determined, we do not bootstrap from that peer, and the peer initiates a bootstrap on its end to sycn up. We do not push blocks to peers in this design.

Once we decide we are behind (our sequence number is less than the peer), we issue a pull request for the blocks we are behind on. Once we receive the blocks, we are able to validate and apply them to the database. Note that the order of retrieving blocks works by going micro block to micro block such that each of the batch state blocks of the current micro is downloaded and validated/applied prior to the current micro. And once the batch state blocks are all applied, we validate/apply the current micro block. We then proceed to get the next micro block in the system. If we reach the last micro block in an epoch, we get the next micro and the next epoch after we validate the current epoch.

The basic algorithm is as follows:

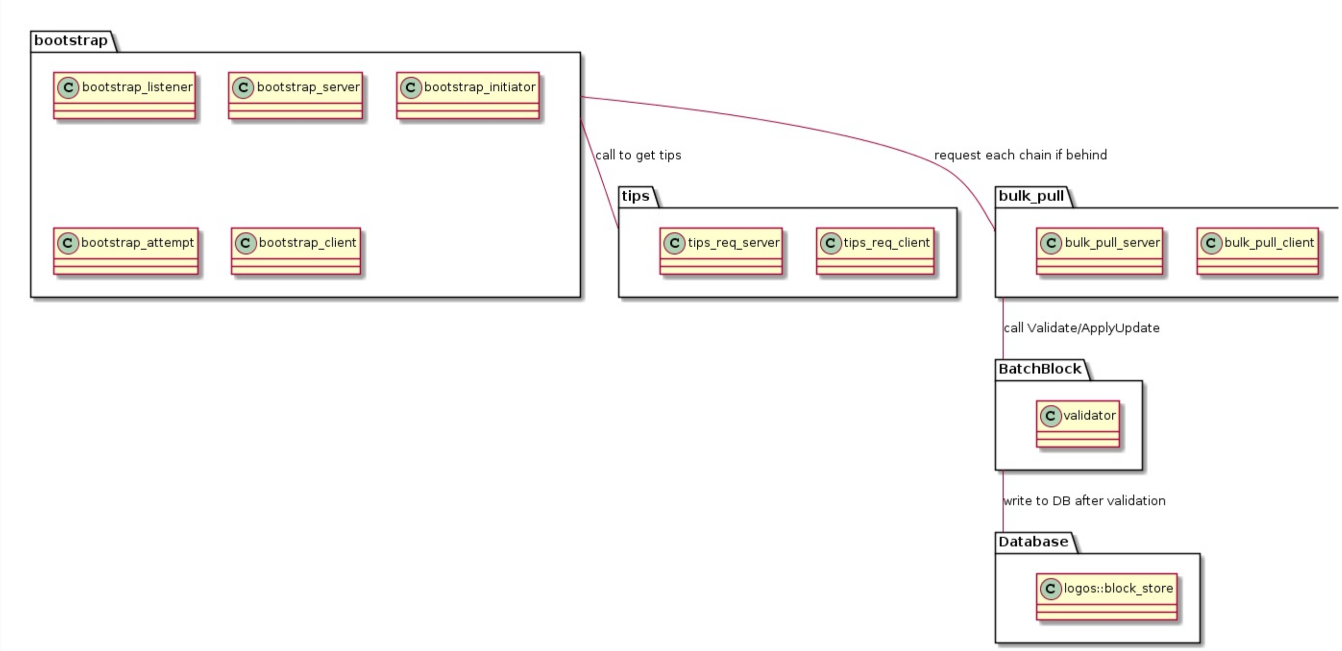
```
        while (true) {
            get tips from peer
            if behind for delegate i
                request a pull request
        }
```

And subsequently, the bootstrap client logic will issue a request from the peer for the pull requested.

```
        while (still_pulling) {
            send peer a request for the pull
            receive the response in the form of a
                block type
            validate/apply the block
        }
```

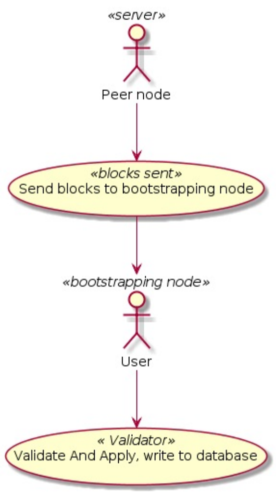
Currently, the system supports retrieving epoch, micro, and batch state blocks.

3. interfaces



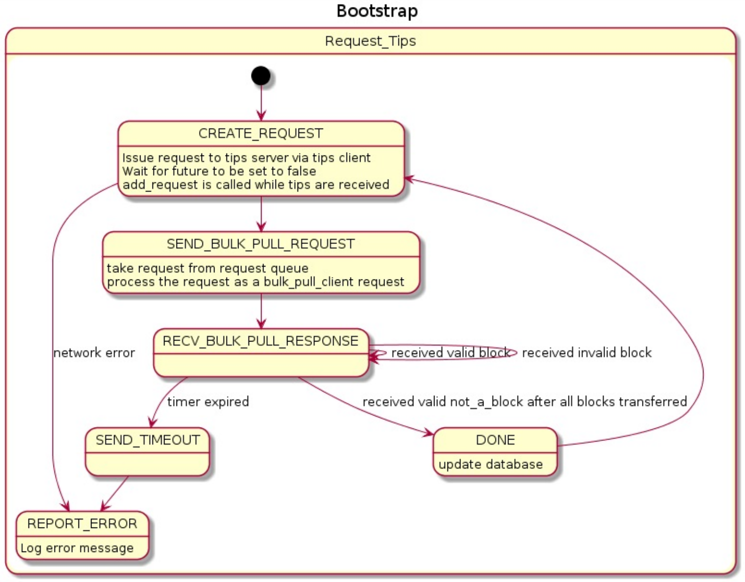
4. use-case diagrams

The use case here is that a node that is bootstrapping, requests from the peer node its state, and ultimately retrieves its blocks where the bootstrapping node will validate and apply the blocks to the database in attempt to sync up the two databases.

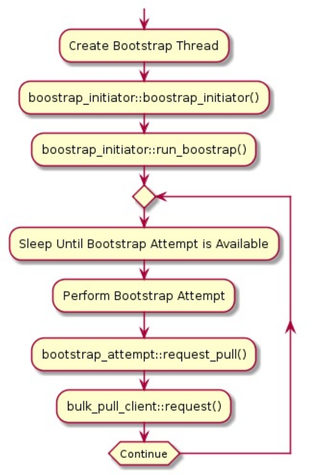

«server»
Peer node
«blocks sent»
Send blocks to bootstrapping node
«bootstrapping node»
User
« Validator»
Validate And Apply, write to database

5. state diagrams

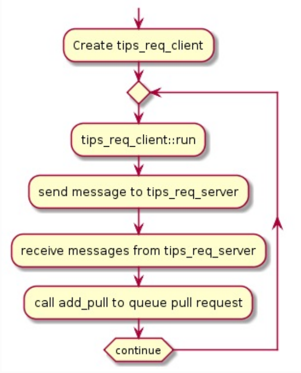
Overall state :


Bootstrap
Request_Tips
CREATE_REQUEST
Issue request to tips server via tips client
Wait for future to be set to false
add_request is called while tips are received
SEND_BULK_PULL_REQUEST
take request from request queue
process the request as a bulk_pull_client request
RECV_BULK_PULL_RESPONSE
received valid block
received invalid block
network error
timer expired
received valid not_a_block after all blocks transferred
SEND_TIMEOUT
DONE
update database
REPORT_ERROR
Log error message

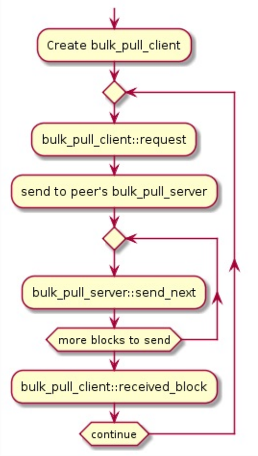
The intial sequence for initiating bootstrapping is as follows :


Create Bootstrap Thread
boostrap_initiator::boostrap_initiator()
boostrap_initiator::run_boostrap()
Sleep Until Bootstrap Attempt is Available
Perform Bootstrap Attempt
bootstrap_attempt::request_pull()
bulk_pull_client::request()
Continue

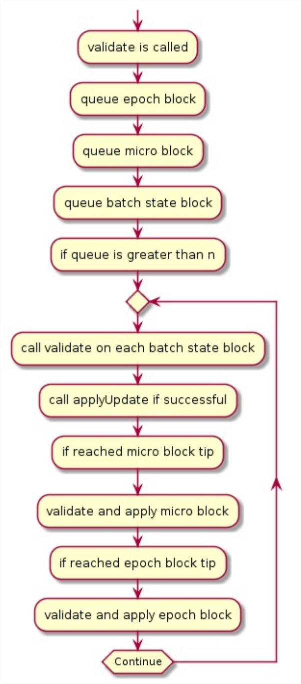
Responding to tips request and setting up a pull request :


Create tips_req_client
tips_req_client::run
send message to tips_req_server
receive messages from tips_req_server
call add_pull to queue pull request
continue

Responding to a pull request :


Create bulk_pull_client
bulk_pull_client::request
send to peer's bulk_pull_server
bulk_pull_server::send_next
more blocks to send
bulk_pull_client::received_block
continue

Validation of blocks :


validate is called
queue epoch block
queue micro block
queue batch state block
if queue is greater than n
call validate on each batch state block
call applyUpdate if successful
if reached micro block tip
validate and apply micro block
if reached epoch block tip
validate and apply epoch block
Continue

6. classes

The overview of the system is that it has four main components and a few helper classes/structures. The overall design is we have:

1. bootstrap
2. bulk_pull
3. tips
4. validator

The bootstrapping module primarily contains low-level multi-threading as well as networking. Its main classes are:

1. bootstrap_listener
2. bootstrap_server
3. bootstrap_initiator
4. bootstrap_attempt
5. bootstrap_client

The bulk_pull module consists of the following main classes:

1. bulk_pull_client
2. bulk_pull_server

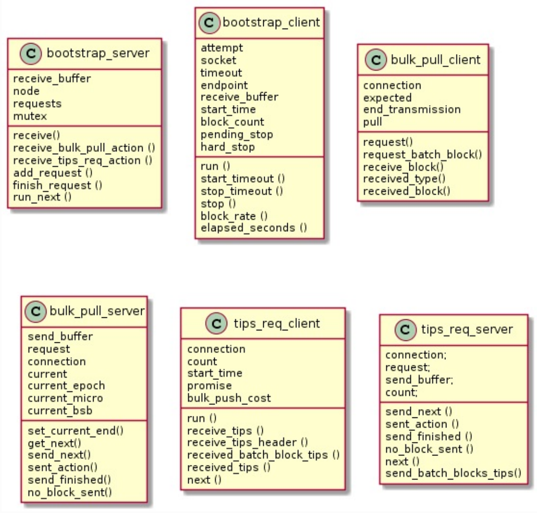
The tips module consists of the following main classes:

1. tips_req_client
2. tips_req_server

The validator consists of one class:

1. BatchBlock::validator

An overview of the main classes :



bootstrap_listener

This class is used to listen to connections from peers. In **bootstrap_listener::start()** are the calls to accept server sockets. The main function here is **logos::bootstrap_listener::accept_action** where new server connections are made and stored in a container called **connections**.
Of importance is the call to:

```
auto connection (std::make_shared<logos::bootstrap_server> (socket_a, node.shared ()));
```

Which creates the bootstrap_server class from which all servers are called. The start of the bootstrap_server logic is in the method **receive** which is also called from **accept_action**.

bootstrap_server

This class is responsible for invocking the other servers. Note that this design uses a visitor design pattern rather than inheritance. That is to say, the bulk_pull_server, and tips_req_server do not inherit from bootstrap_server. However, they are called from the server using a visitor pattern. See **request_response_visitor** for more information. The design allows for the server to be available as a **connection** member variable inside the bulk_pull_server, and tips_req_server. A similar design is used in implementing clients.

As aforementioned, the main starting point for bootstrap_server is:

```
bootstrap_server::receive()
```

Which calls a Boost ASIO async_read call to start the server read from its socket. Additional methods and their purpose are:

```
void logos::bootstrap_server::receive_bulk_pull_action (boost::system::error_code const & ec, size_t size_a)
void logos::bootstrap_server::receive_tips_req_action (boost::system::error_code const & ec, size_t size_a)
void logos::bootstrap_server::add_request (std::unique_ptr<logos::message> message_a)
void logos::bootstrap_server::finish_request ()
void logos::bootstrap_server::run_next ()
```

Where **receive_bulk_pull_action** receives the bulk_pull requests from client and **receive_tips_req_action** receives the tips requests from client. **add_request** adds a request to the **requests** queue which is defined as:

```
std::queue<std::unique_ptr<logos::message>> requests;
```

Note that bootstrapping uses messages that derive from **logos::message** for representing a pull request or tip request for example.

finish_request is called when the bulk_pull_server finishes processing the requests. In our case, finishes walking a chain and sending all of the blocks requested back to the client. It is important to have this function called via:

```
connection->finish_request()
```

From inside the bulk_pull_server.

run_next is a method called to process the next request from the queue. Note that how this works is by retreiving a request from the requests queue mentioned above, and applying a visitor pattern as follows:

```
request_response_visitor visitor (shared_from_this ());
requests.front()->visit (visitor);
```

Where the **request_response_visitor** handles the various message types by invocking the correct server. For example, in the case of bulk_pull we have the request encapsualted in **logos::pull_info** object, and we apply the following method from the visitor:

```
void bulk_pull (logos::bulk_pull const &) override
```

Which in turn calls the bulk_pull_server in this manner:

```
auto response (std::make_shared<logos::bulk_pull_server> (connection, std::unique_ptr<logos::bulk_pull> (static_cast<logos::bulk_pull *> (connection->requests.front ().release ()
response->send_next ();
```

Notice how the connection is passed into the constructor of the bulk_pull_server and that the connection object is the bootstrap_server class.

bootstrap_initiator

The initiator class is what starts bootstrapping. It can be called from a unit test, for example:

```
node2.bootstrap_initiator.bootstrap (node1.network.endpoint ());
```

Where we specify to a node a peer to bootstrap from. In normal operation, the code calls the following in a loop from node.cpp:

```
logos::node::ongoing_bootstrap()
{   ...
    bootstrap_initiator.bootstrap ();
    ...
}
```

Where the bootstrap is launched every n seconds. This interval is currently set to 300s (but in testing is set to much smaller value).

The main methods from bootstrap_initiator are:

```
void logos::bootstrap_initiator::bootstrap ()
void logos::bootstrap_initiator::bootstrap (logos::endpoint const & endpoint_a, bool add_to_peers)
void logos::bootstrap_initiator::run_bootstrap ()
bool logos::bootstrap_initiator::in_progress ()
std::shared_ptr<logos::bootstrap_attempt> logos::bootstrap_initiator::current_attempt ()
void logos::bootstrap_initiator::stop ()
```

The **bootstrap()** and **bootstrap(logos::endpoint,bool)** methods initiate bootstrapping. In the former, we select a peer from a list, and in the later, we directly initiate bootstrapping from the specified endpoint. We currently use a specific endpoint for testing.

run_bootstrap is the method that runs the bootstrapping for a **bootstrap_attempt**.

The methods **in_progress** and **current_attempt** return true if there is a bootstrap in progress (i.e., when the attempt object is not null) and return the current attempt object respectively.

The **stop** method stops the bootstrapping (by calling **attempt->stop()**)

bootstrap_attempt

This class encapsulates a given bootstrap request and effecitively is the top-level control of bootstrapping requests and networking. It is contained in **bootstrap_initiator** and itself contains the following main methods:

```
void run ();
bool consume_future (std::future<bool> &);
void populate_connections ();
bool request_tips (std::unique_lock<std::mutex> &);
void request_pull (std::unique_lock<std::mutex> &);
void request_push (std::unique_lock<std::mutex> &);
void add_connection (logos::endpoint const &);
void pool_connection (std::shared_ptr<logos::bootstrap_client>);
void stop ();
void requeue_pull (logos::pull_info const &);
void add_pull (logos::pull_info const &);
bool still_pulling ();
bool still_pushing ();
void process_fork (MDB_txn *, std::shared_ptr<logos::block>);
unsigned target_connections (size_t pulls_remaining);
bool should_log ();
```

The main method here is **run()** which drives the bootstrapping. From there we see calls to request_tips (which obtains the tips from the peer and produces as a side-effect the **logos::pull_info** structures queing them up into the **pulls** vector. From there, the system decides to issue **request_pull** calls on for each pull request.

Another important method here is **populate_connections** which is responsible for managing peer connections. It currently checks which peers are slow and attempts to disconnect from them. It also decides to allocate new peer connections to satisfy pull requests.

pool_connection allows the client servers to place a connection (with a live socket) onto an idle queue for re-use at a later point. Note that care must be taken that all re-usuable sockets are queued up upon successful completion of a request.

The helper methods **add_pull**, **still_pulling**, **target_connections**, and **still_pushing** are used to add a pull request (called from tips client), deciding whether the system is still doing pull requests, calculating how many connections to create based on number of pulls outstanding, and determine if we are still in the process of handling push requests.

bootstrap_client

The bootstrap_client is the driver of all client requests. It is similar to the server class in that it envokes the correct client to handle one of the possible requests (tips,push,pull). The main methods of this class are:=

```
void run ();
void start_timeout ();
void stop_timeout ();
void stop (bool force);
double block_rate () const;
double elapsed_seconds () const;
```

The **run** method is the start of the client logic. **start_timeout**, **stop_timeout** control timeouts on the socket connections. Note that all read/write operationsuse boost::asio **async_read/async_write**. The **stop** method stops the client. Passing in false implies we stop after the operation specific client finishes its request (i..e, after we finish pulling) and passing in true implies we stop immediately.

The other two functions are for calculating the block rate and elapsed time (since the client started), respectively.

bulk_pull_client

The **bulk_pull_client** is responsible for making a connection to the peer given a request for a pull. A pull request is encapsulated using **logos::pull_info** class and is stored in a queue called **pulls** inside the bootstrap.cpp code. A given pull is taken from the top of the queue and processed using **request_pull**. This is a call to the **bulk_pull_client** passing in the pull_info an the connection object which is the bootstrap_client class. Inside the client is the code that handles making asynchronous requests via boost::asio to the server.

The main methods of this class are:

```
void request ();
void request_batch_block();
void receive_block ();
void received_type ();
void received_block (boost::system::error_code const &, size_t);
```

Where **request** is the start of the request to the peer. **request_batch_block** is specifically for logos blocks. The async_read/async_write calls used require callbacks and are setup in a so-called composed operation. The methods which are listed in order of execution are **receive_block**, **received_type**, and **received_block**. These receive the blocks in three stages, first the response from the initial request, then the type (a single character read and tested for block type), and then the actual block that is sent. In our case, one of **Epoch**, **Micro**, or **Batch State Blocks**. Once the code receives a block, it calls the validator class to pass the block (which is represented as a shared_ptr) to the validator for processing.
The validator uses the respective **Validate** and **ApplyUpdate** methods of each of the supported block types.

bulk_pull_server

The **bulk_pull_server** is responsible for transferring blocks from the peer to the client. It is created inside bootstrap.cpp via the method **logos::bootstrap_listener::accept_action** which creates a connection object using: **auto connection (std::make_shared (socket_a, node.shared ()));**

Notice that this is where the connection for the **bootstrap_server** is created. The actual call to create **bulk_pull_server** happens in bootstrap.cpp, method in **request_response_visitor** called **bulk_pull**. The visitor is called from **bootstrap_server::run_next** where a request is pulled of the queue and processed using the visitor pattern.

The main methods in this class are:

```
void set_current_end ();
std::unique_ptr<logos::block> get_next ();
void send_next ();
void sent_action (boost::system::error_code const &, size_t);
void send_finished ();
void no_block_sent (boost::system::error_code const &, size_t);
```

Where **set_current_end** sets the hash that when reached implies we reached the end of transmission. **get_next** gets the next block in the chain, **send_next** sends the next block. Like the **bulk_pull_client**, the server uses boost::asio to do the async_write. Therefore, it requires several methods for performing the composed operation. These methods are (in order of execution), **sent_action**, **send_finished**, and **no_block_sent**. The code uses **sent_action** to acknowledge the block has been sent. **send_finished** is the last call made to send the last block marked by a **not_a_block** type. This is important because it signals to the client the end of transmission. The server does not validate the blocks it sends, as the validation is left on the client side.

tips_req_client

The **tips_req_client** sends the initial message to the **tips_req_server**. In the message, is a request for tips for batch state blocks, micro blocks and epoch blocks. The message is relatively simple, and its response is handled in the is class. It is expected that the server sends back its tips (the peer's tips) for which we intend to decide what to do with each tip in terms of making requests to the bootstrap code to pull or push. The main methods here are:

```
void run ();
void receive_tips ();
void receive_tips_header ();
void received_batch_block_tips (boost::system::error_code const &, size_t);
void received_tips (boost::system::error_code const &, size_t);
```

run starts the process of sending a request to the server. **receive_tips**, **receive_tips_header**, **receive_batch_block_tips**, and **received_tips** are networking code to receive the tips, and **received_tips** does the processing of calling the bootstrap code to request a pull or push. Note that the logic deciding what to do is based on sequence numbers returned from the server. Because we request the entire chain for a given delegate, the chains have to be linked from micro block transitions.

tips_req_server

The **tips_req_server** is responsible for sending the peer's tips so that the requesting node can determine where it is. The main methods here are:

```
void send_next ();
void sent_action (boost::system::error_code const &, size_t);
void send_finished ();
void no_block_sent (boost::system::error_code const &, size_t);
void next ();
void send_batch_blocks_tips();
```

Where **send_next** decides which tip to send next, **send_action** is used for asynchronous write of the tip to the network, and **send_finished** and **no_block_sent** send the signal that the transmission is complete. **send_batch_block_tips** is the main method here which is responsible for sending tips information for each delegate. Note that the client always requests the epoch and micro blocks first, followed by batch blocks.

BatchBlock::validator

This class is purpose is to **Validate** and **ApplyUpdate** each incoming block. It sorts the queue of blocks, and calls the validation and apply of the batch state blocks, when it reaches a micro block, it validates the micro block, and when it reaches the epoch block, it validates the epoch block. The micro block is reached when we process its batch state block tips, and the epoch block is reached when we process its micro block tips. The system processes each request concurrently and queues the blocks up for processing as the downloading continues.

The main methods here are:

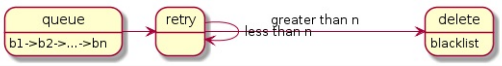
```
void add_micro_block(std::shared_ptr<bulk_pull_response_micro> &m)
void add_epoch_block(std::shared_ptr<bulk_pull_response_epoch> &e)
bool validate(std::shared_ptr<bulk_pull_response> block);
```

Where **add_micro_block** and **add_epoch_block** add received micro and epoch blocks to the validator on seperate queues. And **validate** is used to determine if the batch block is valid. Note that we queue a certain number of blocks before attempting to validate, and if we can not proceed, we wait for more blocks to arrive.

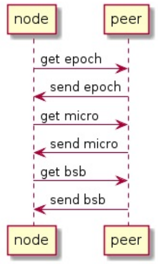
The code wraps up the epoch, micro, and batch blocks into its own internal structures to keep the modules seperated and so that bootstrapping knows little of the other components except to call the
required methods.

7. sequence diagrams

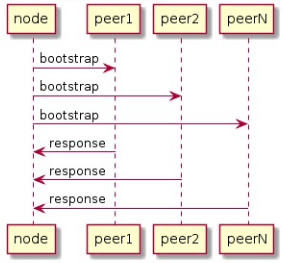
An illustration of retry logic.



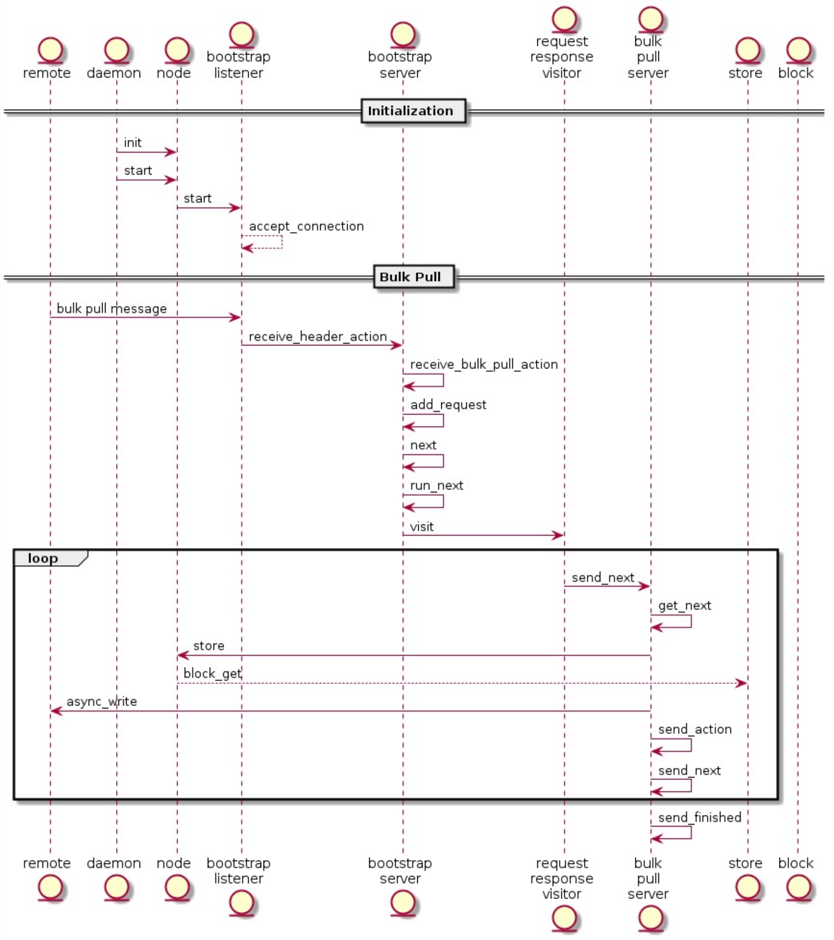
An illustration of logical bootstrapping.



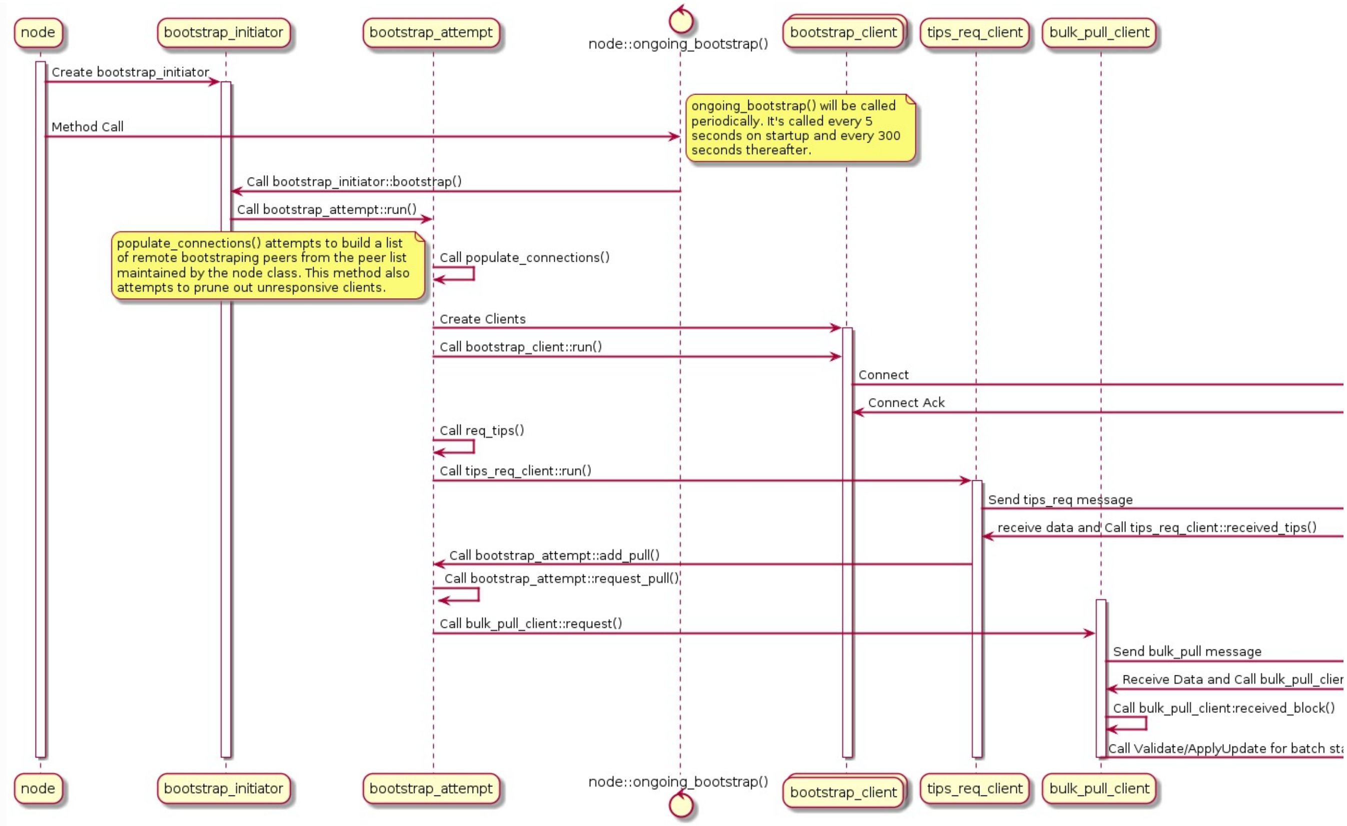
An illustration of parallel bootstrapping.



An illustration of the sequence for bootstrap initialization and start of bulk_pull requests.



An illustration of the system functionality. This diagram shows how bootstrapping works when its up and running. Main ideas is the request for tips, the response, and then the subsequent pulls.


node
bootstrap_initiator
bootstrap_attempt
node::ongoing_bootstrap()
bootstrap_client
tips_req_client
bulk_pull_client
Create bootstrap_initiator
Method Call
ongoing_bootstrap() will be called periodically. It's called every 5 seconds on startup and every 300 seconds thereafter.
Call bootstrap_initiator::bootstrap()
Call bootstrap_attempt::run()
populate_connections() attempts to build a list of remote bootstraping peers from the peer list maintained by the node class. This method also attempts to prune out unresponsive clients.
Call populate_connections()
Create Clients
Call bootstrap_client::run()
Connect
Connect Ack
Call req_tips()
Call tips_req_client::run()
Send tips_req message
receive data and Call tips_req_client::received_tips()
Call bootstrap_attempt::add_pull()
Call bootstrap_attempt::request_pull()
Call bulk_pull_client::request()
Send bulk_pull message
Receive Data and Call bulk_pull_clier
Call bulk_pull_client:received_block()
Call Validate/ApplyUpdate for batch sta
node
bootstrap_initiator
bootstrap_attempt
node::ongoing_bootstrap()
bootstrap_client
tips_req_client
bulk_pull_client

8. resources

- Currently, the code is too aggressive in allocating sockets, and more work is needed to make sure we don’t run out of sockets in a given run.
- The requests submitted by the tips client should wait until a given delegate chain is retrieved before submitting more requests for that delegate. This is currently done in *tips_req_client*

9. implementation and algorithms

- The validation algorithm is currently:

```
queue epoch blocks
queue micro blocks
queue batch state blocks

bool validate()
{
     when batch state blocks queue is greater than n do
        sort queue based on timestamp
        iterate through each batch state block
            validate batch state block i
            apply update on batch state block i

        if we reach the next micro block tips
           validate micro block
           apply update on micro block

        if we reach the next epoch block tips
           validate epoch block
}
```

- Description for logical bootstrapping:

Client sends sequence numbers and hashes to server

Server decides to initiate bootstrapping on its end if it’s behind by
Calling connection->node->ongoing_bootstrap()
using sequence numbers as a guide.
Note that ongoing_bootstrap() is called automatically every 5 minutes,
but I believe we can call it early in the case we are behind.

The server then sends back a response
indicating to proceed with next peer.
This allows the client to set the promise to false and not initiate pulls from this peer.

Otherwise, the server sends back the range for doing a pull of epoch blocks
using the client’s epoch block tip as a start and its own epoch block tip as an end.
The client then initates the pull of the epoch blocks from the server.

When the epoch block tip is equal to the peer’s, the server sends
back the range of micro blocks using the client’s micro block tip as a start
and its own micro block tip back to the client. The client then pulls the
chain of micro blocks. Note that when we send epoch
block chains and pull epoch blocks, in the next run,
we will get the micro block tips as we sync up the epoch blocks.

If we reach a state where the micro block tips are equal, we have to send the bsb block range (from the client’s tip to our tip).

This means, the tips client has to understand which type of block to pull given the aforementioned responses the tips server can give.

When micro blocks arrive (in their callback) we request the associated bsb blocks. And the bsb blocks will arive for each micro block by way of an individual pull for each of the delegate bsb tips as specified in the micro block. As they arrive, they will be validated and applied to the database.

When we reach the end of epoch block transmission, it should pull the remaining micro blocks by going from the last epoch block micro block tip to the micro block tip on the peer. This will be done in the tips client/server.

When we reach the end of micro block transmission, it should pull from the last micro block bsb tips to the peer’s bsb block tips for a request to be made for remaining bsb blocks. This will be done in the tips client/server.

Note that the peer’s tip for epoch, micro, and bsb can be retrieved and perhaps sent to the pull request if needed.

This implies changes in the tips.cpp code and in the callback handlers of epoch/micro block.

Note that in the current case, and in this case, we would be validating bsb blocks first, followed by micro and epoch and that this is the normal sequence of things similar to the way consensus processes the blocks.

- parallel bootstrapping:

Parallel bootstrapping refers to the process by which a node will bootstrap with a set of networked peers. It selects peers from a list and tries to bootstrap from them. We maintain a list of peers using either p2p module or our list of delegates. The list is a set of boost asio endpoints. Then, in order to bootstrap, we select a peer from the list in an iterative way. The peer is then queried for its tips. The selection of peers happens in populate_connections() by calling a function called bootstrap_peer(). Once we have a peer, we start by checking our tips. If we are not in sync, we will request blocks by iterating through micro block by micro block. When a micro block is downloaded, its bsb blocks are requested in seperate pulls. Each pull may be requested on a seperate peer. When we have all the bsb blocks, we will proceed to validating the micro block, and only then, we will advance to the next micro block. Each pull request can potentially be from a different peer.

Each request is made to a seperate peer and each peer responds with the blocks that were asked of it. We will need to integrate with p2p peer list and incorporate blacklisting. Our design for blacklisting is to re-use the p2p code for this, and then to keep track of which ip address sent us which block (using the IP Address or an integer representing the peer in the list), and when a criteria is reached, such as maximum number of retries, we can decide to add the peer to a blacklist. The API should include adding the peer to the list (perhaps using a boost endpoint as a way of identifying the peer) and a way to check if the peer is in the blacklist. To implement, we would check in populate_connections() if the peer is blacklisted, and avoid using that peer for a connection. When we decide that a block is no good (in validator) we can add the peer to the blacklist. The p2p peer list has to keep the current set of peers to communicate with and has to populate the bootstrapping peer list (or we can change the implementation of logos::peer_container::bootstrap_peer to return a logos::endpoint based on what is in the peer list and perhaps taking into consideration if the peer is blacklisted or not).

- consolidate tips into one message:

As mentioned in our logical bootstrapping design, the bootstrapping node will send its tips information to the peer. The peer will use that information to decide if its ahead or behind. If it is behind, it will initiate bootstrapping on its end and signal to the bootstrapping node to proceed using another peer. The message containing tips information is a single message that contains information about the epoch block tip, the micro block tip, and each individual delegates tips as opposed to sending a message for each delegate, we send just one representing a snapshot of the sytem.