Identity Management

1. Overview

The Identity Management (IM) subsystem provides the overall identification capability for a node. It guides the Logos node software to

1. ascertain its role within the Logos Ecosystem,
2. keep track of key communication endpoints, namely delegates' consensus and transaction-acceptor (TxA) addresses and ports, and
3. as a consensus participant, communicate its local endpoint(s) to the Logos network, as well as establish connections with peer delegates and their TxA's.

The main additional component introduced by IM is the concept of a "**sleeve**" within the core software. A sleeve is a container that stores and manages the node's governance identity. It can be in various states (*Section 5.1*), which affects the types of role(s) the software can assume.

Additionally, action settings (*Section 5.2*) impact the actual behavior the node exhibits in the network, including if / when it participates in consensus and if / when it advertises its communication endpoints. The software provides an interface for user to modify and view the node identity states and action settings.

2. Execution Concept

Identity management is driven by internal events, user input (during software run), user-provided settings, as well as p2p messages.

The node operator can enable / disable all identity management-related actions in the configuration file. If control over identity management is disabled, all IM-related user input will be rejected.

IM provides an interface to modify the node's identity and connectivity settings:

1. On user input, the node software can lock/unlock its sleeve.
2. On user input, the sleeve can modify its stored identity content.
3. On user input, the node software schedules a change to its activation status, either immediately or at the start of a future epoch.
4. On user input, the node adds or deletes TxA's

Node operators can therefore manage the nodes' identity and enable/disable network participation through these actions by triggering sleeve state changes (see *Section 7.1* for details).

The IM component also serves as an endpoint communication scheduler. If the node is an **activated** delegate for the upcoming epoch,

1. around one hour before the epoch start, it advertises consensus and TxA endpoints;
2. around thirty minutes before the epoch start, it advertises the same endpoints;
3. every ten minutes starting at one hour before the epoch start, it requests other delegates' endpoints that are missing from its internal records.

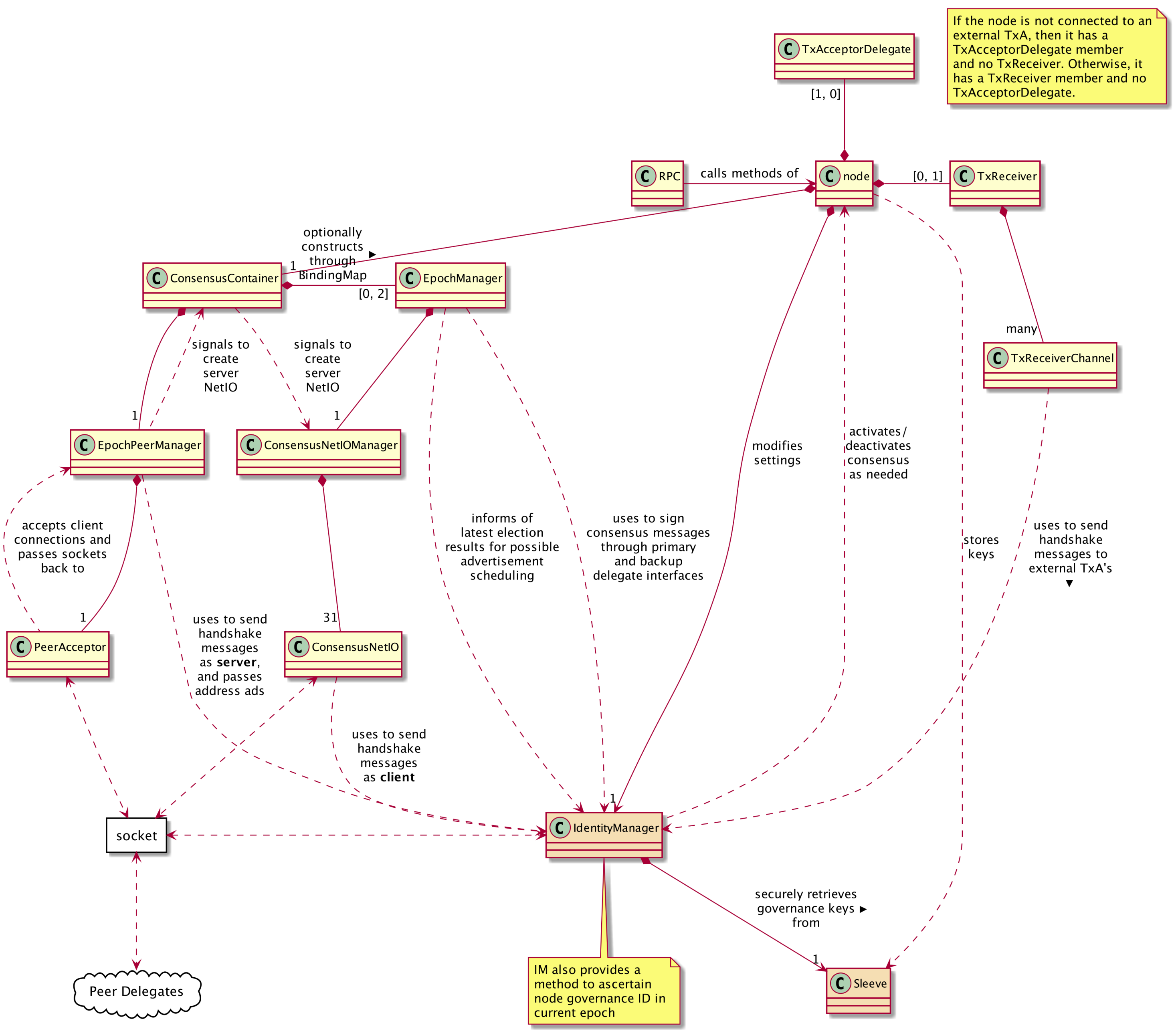
Finally, it acts as record-keeper and relay of key network consensus endpoints. Receiving other delegates' endpoint advertisement messages triggers interaction with the database and possible rebroadcast to peers.

Details of these events are covered in the sections below.

3. Interfaces

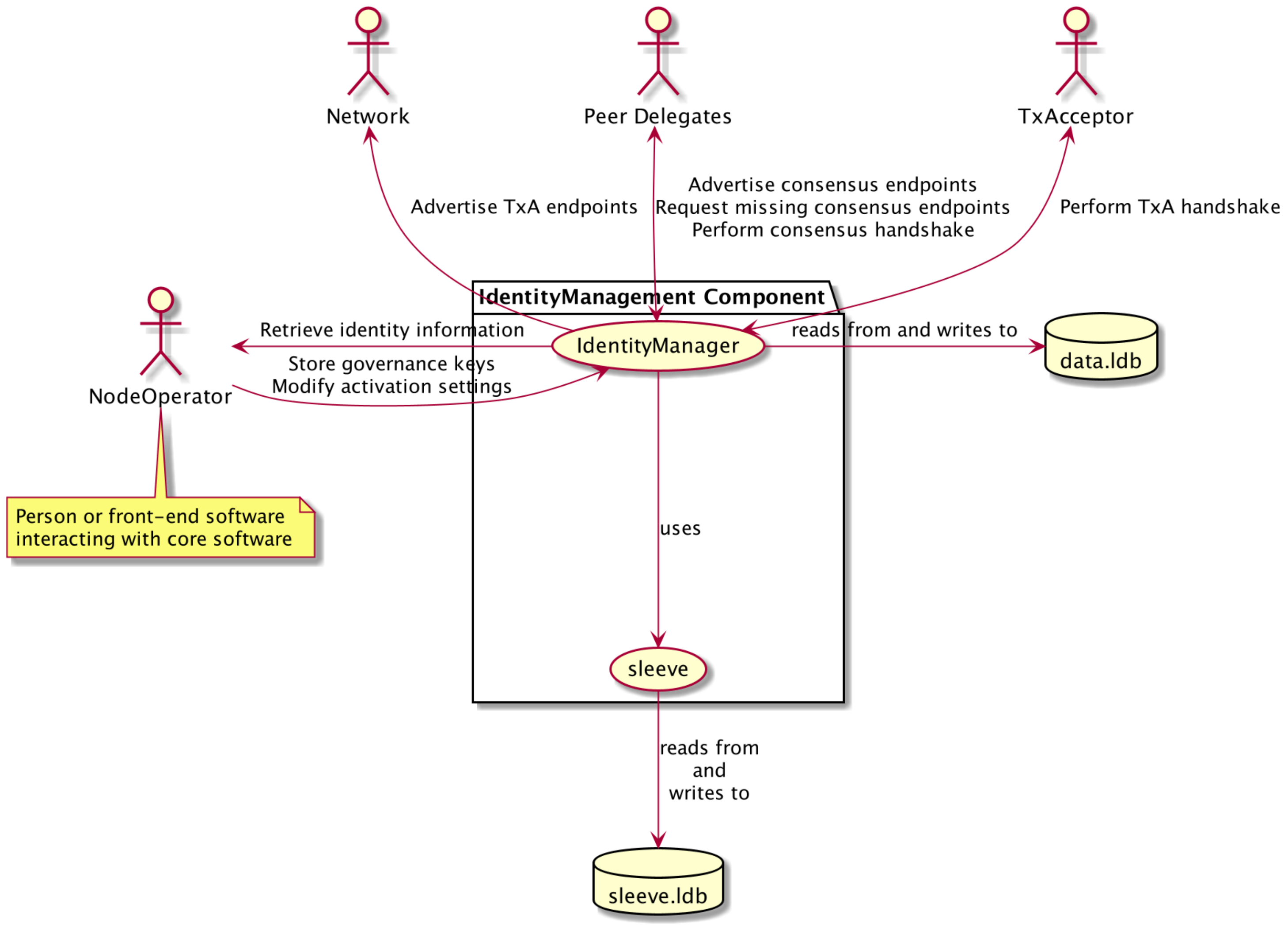
Various core componenets rely on IdentityManager for all operations related to the node's own identity. These include sending handshake messages to delegate peers either as server or client, sending handshake messages to TxA's for authentication, and ascertaining whether to participate in consensus in a given epoch. IdentityManager in turn relies on the Sleeve class as a secure way to manage and access the node's identity data.

New/modified classes are darkened. Some classes' members are less relevant and omitted. Note that access to IM is often indirectly provided by `ConsensusContainer` *(through* `EpochManager` *)*


If the node is not connected to an external TxA, then it has a TxAcceptorDelegate member and no TxReceiver. Otherwise, it has a TxReceiver member and no TxAcceptorDelegate.
TxAcceptorDelegate
[1, 0]
RPC
calls methods of
node
[0, 1]
TxReceiver
optionally constructs through BindingMap ▸
1
ConsensusContainer
[0, 2]
EpochManager
many
TxReceiverChannel
signals to create server NetIO
signals to create server NetIO
1
1
EpochPeerManager
ConsensusNetIOManager
modifies settings
activates / deactivates consensus as needed
accepts client connections and passes sockets back to
informs of latest election results for possible advertisement scheduling
uses to sign consensus messages through primary and backup delegate interfaces
stores keys
uses to send handshake messages to external TxA's ▾
1
31
PeerAcceptor
uses to send handshake messages as server, and passes address ads
ConsensusNetIO
uses to send handshake messages as client
1
socket
IdentityManager
securely retrieves governance keys ▸ from
1
Peer Delegates
IM also provides a method to ascertain node governance ID in current epoch
Sleeve

4. Use Case Diagrams

The identity management component responds to the node operator's commands through secure RPC to apply setting changes and to store governance keys. If the node is identified as a delegate (see *Section 7* below for identification mechanisms), this component also interfaces with the Logos network (for TxA endpoints), peer delegates (consensus-related operations), as well as its own Tx Acceptor(s). Internally, IdentityManager accesses `data.ldb` for endpoint advertisement and peer delegate data directly, and `sleeve.ldb` for secret private governance key data.

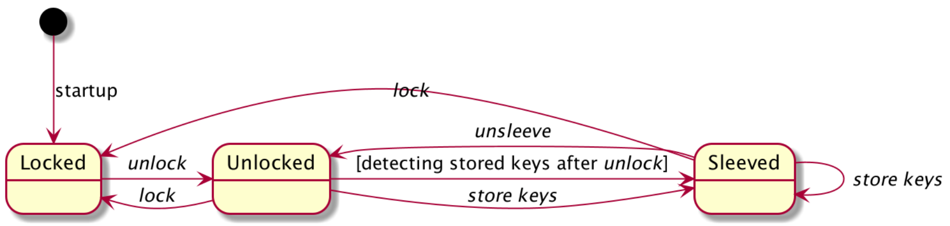

Network
Peer Delegates
TxAcceptor
Advertise TxA endpoints
Advertise consensus endpoints
Request missing consensus endpoints
Perform consensus handshake
Perform TxA handshake
IdentityManagement Component
IdentityManager
Retrieve identity information
Store governance keys
Modify activation settings
NodeOperator
reads from and writes to
data.ldb
Person or front-end software
interacting with core software
uses
sleeve
reads from
and
writes to
sleeve.ldb

5. State Diagrams

a refresher on state definitions, refer to the "Node Identity Management Requirements" document.

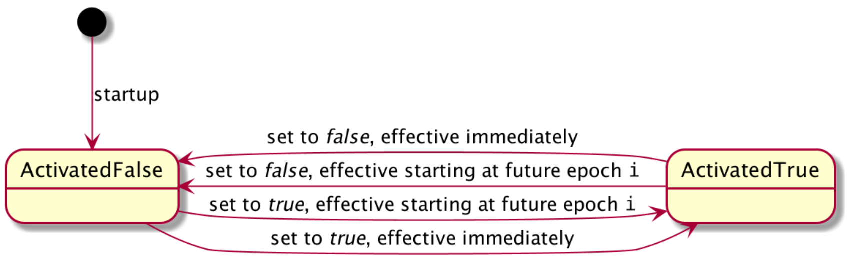
Sleeve states and activation setting are entirely orthogonal to each other. See *Section 7.1.3* for how their combination affects node actions.

5.1. Sleeve States

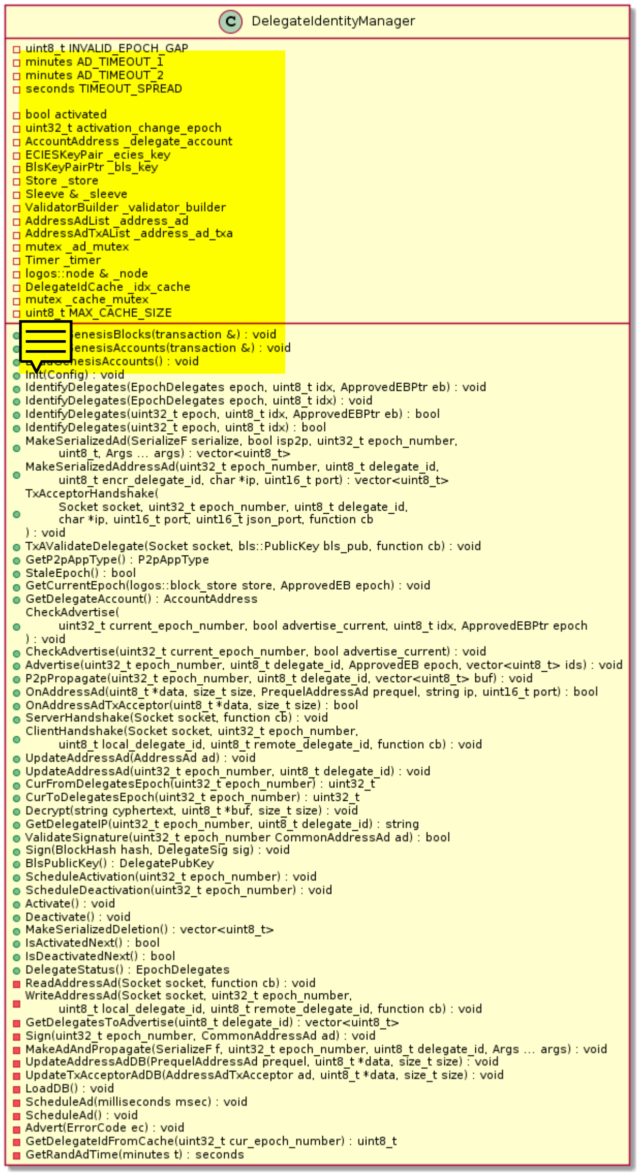


5.2. Activation Settings

See Section 7.1.3 for example activation timeline.


startup
ActivatedFalse
ActivatedTrue
set to *false*, effective immediately
set to *false*, effective starting at future epoch i
set to *true*, effective starting at future epoch i
set to *true*, effective immediately

6. Classes

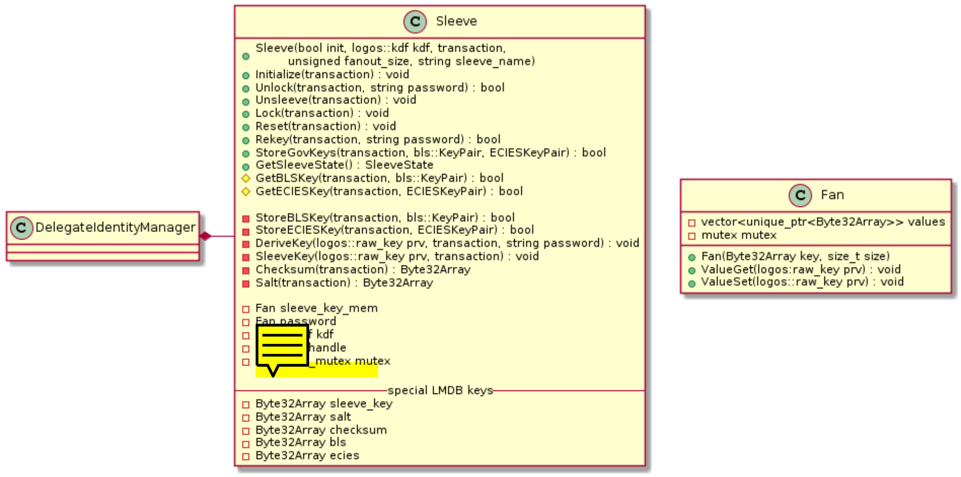


```
DelegateIdentityManager

uint8_t INVALID_EPOCH_GAP
minutes AD_TIMEOUT_1
minutes AD_TIMEOUT_2
seconds TIMEOUT_SPREAD

bool activated
uint32_t activation_change_epoch
AccountAddress _delegate_account
ECIESKeyPair _ecies_key
BlsKeyPairPtr _bls_key
Store _store
Sleeve & _sleeve
ValidatorBuilder _validator_builder
AddressAdList _address_ad
AddressAdTxAList _address_ad_txa
mutex _ad_mutex
Timer _timer
logos::node & _node
DelegateIdCache _idx_cache
mutex _cache_mutex
uint8_t MAX_CACHE_SIZE

CreateGenesisBlocks(transaction &) : void
CreateGenesisAccounts(transaction &) : void
LoadGenesisAccounts() : void
Init(Config) : void
IdentifyDelegates(EpochDelegates epoch, uint8_t idx, ApprovedEBPtr eb) : void
IdentifyDelegates(EpochDelegates epoch, uint8_t idx) : void
IdentifyDelegates(uint32_t epoch, uint8_t idx, ApprovedEBPtr eb) : bool
IdentifyDelegates(uint32_t epoch, uint8_t idx) : bool
MakeSerializedAd(SerializeF serialize, bool isp2p, uint32_t epoch_number,
        uint8_t, Args ... args) : vector<uint8_t>
MakeSerializedAddressAd(uint32_t epoch_number, uint8_t delegate_id,
        uint8_t encr_delegate_id, char *ip, uint16_t port) : vector<uint8_t>
TxAcceptorHandshake(
        Socket socket, uint32_t epoch_number, uint8_t delegate_id,
        char *ip, uint16_t port, uint16_t json_port, function cb
) : void
TxAValidateDelegate(Socket socket, bls::PublicKey bls_pub, function cb) : void
GetP2pAppType() : P2pAppType
StaleEpoch() : bool
GetCurrentEpoch(logos::block_store store, ApprovedEB epoch) : void
GetDelegateAccount() : AccountAddress
CheckAdvertise(
        uint32_t current_epoch_number, bool advertise_current, uint8_t idx, ApprovedEBPtr epoch
) : void
CheckAdvertise(uint32_t current_epoch_number, bool advertise_current) : void
Advertise(uint32_t epoch_number, uint8_t delegate_id, ApprovedEB epoch, vector<uint8_t> ids) : void
P2pPropagate(uint32_t epoch_number, uint8_t delegate_id, vector<uint8_t> buf) : void
OnAddressAd(uint8_t *data, size_t size, PrequelAddressAd prequel, string ip, uint16_t port) : bool
OnAddressAdTxAcceptor(uint8_t *data, size_t size) : bool
ServerHandshake(Socket socket, function cb) : void
ClientHandshake(Socket socket, uint32_t epoch_number,
        uint8_t local_delegate_id, uint8_t remote_delegate_id, function cb) : void
UpdateAddressAd(AddressAd ad) : void
UpdateAddressAd(uint32_t epoch_number, uint8_t delegate_id) : void
CurFromDelegatesEpoch(uint32_t epoch_number) : uint32_t
CurToDelegatesEpoch(uint32_t epoch_number) : uint32_t
Decrypt(string cyphertext, uint8_t *buf, size_t size) : void
GetDelegateIP(uint32_t epoch_number, uint8_t delegate_id) : string
ValidateSignature(uint32_t epoch_number CommonAddressAd ad) : bool
Sign(BlockHash hash, DelegateSig sig) : void
BlsPublicKey() : DelegatePubKey
ScheduleActivation(uint32_t epoch_number) : void
ScheduleDeactivation(uint32_t epoch_number) : void
Activate() : void
Deactivate() : void
MakeSerializedDeletion() : vector<uint8_t>
IsActivatedNext() : bool
IsDeactivatedNext() : bool
DelegateStatus() : EpochDelegates
ReadAddressAd(Socket socket, function cb) : void
WriteAddressAd(Socket socket, uint32_t epoch_number,
        uint8_t local_delegate_id, uint8_t remote_delegate_id, function cb) : void
GetDelegatesToAdvertise(uint8_t delegate_id) : vector<uint8_t>
Sign(uint32_t epoch_number, CommonAddressAd ad) : void
MakeAdAndPropagate(SerializeF f, uint32_t epoch_number, uint8_t delegate_id, Args ... args) : void
UpdateAddressAdDB(PrequelAddressAd prequel, uint8_t *data, size_t size) : void
UpdateTxAcceptorAdDB(AddressAdTxAcceptor ad, uint8_t *data, size_t size) : void
LoadDB() : void
ScheduleAd(milliseconds msec) : void
ScheduleAd() : void
Advert(ErrorCode ec) : void
GetDelegateIdFromCache(uint32_t cur_epoch_number) : uint8_t
GetRandAdTime(minutes t) : seconds
```

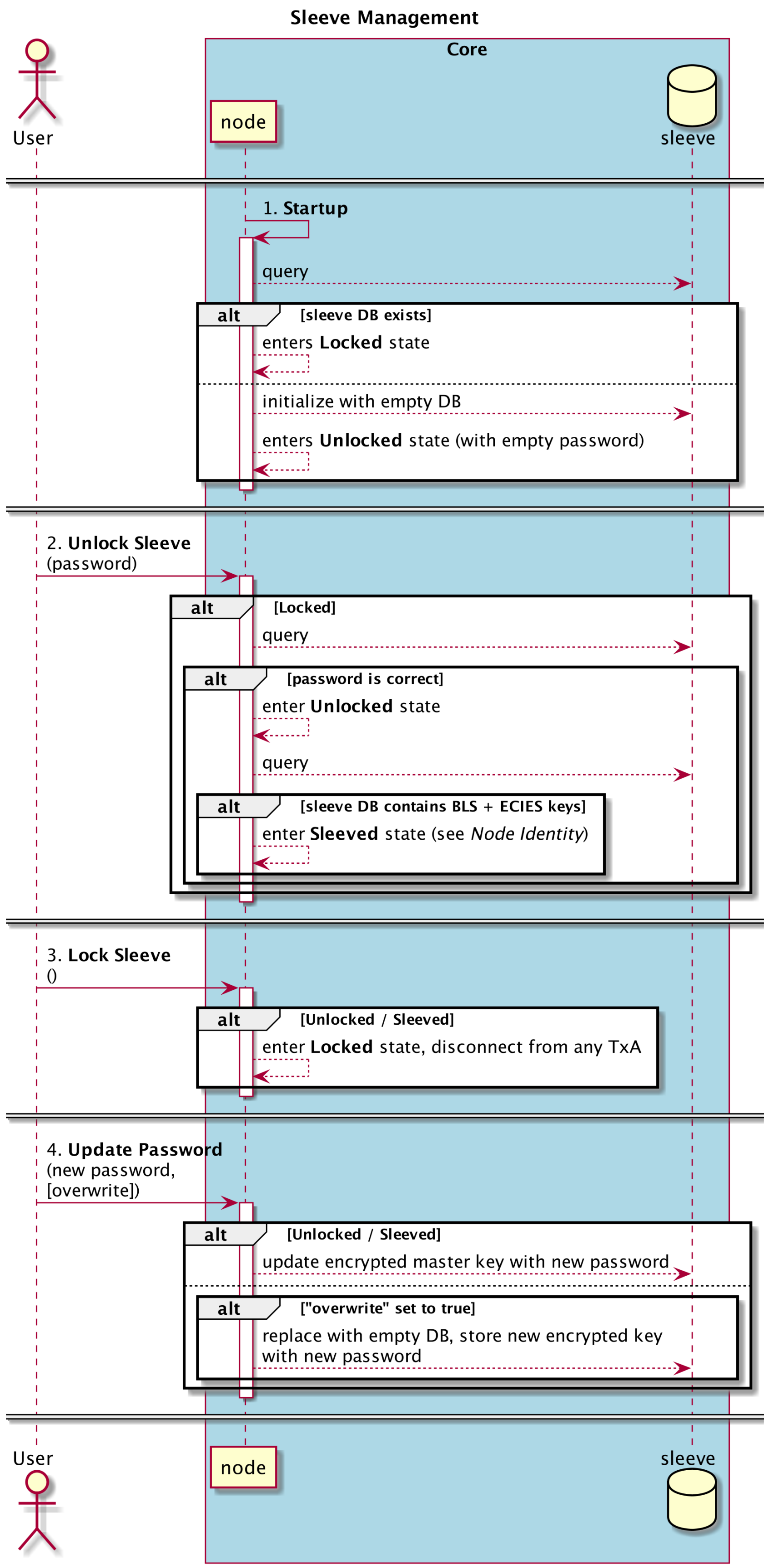

DelegateIdentityManager
Sleeve
Sleeve(bool init, logos::kdf kdf, transaction, unsigned fanout_size, string sleeve_name)
Initialize(transaction) : void
Unlock(transaction, string password) : bool
Unsleeve(transaction) : void
Lock(transaction) : void
Reset(transaction) : void
Rekey(transaction, string password) : bool
StoreGovKeys(transaction, bls::KeyPair, ECIESKeyPair) : bool
GetSleeveState() : SleeveState
GetBLSKey(transaction, bls::KeyPair) : bool
GetECIESKey(transaction, ECIESKeyPair) : bool
StoreBLSKey(transaction, bls::KeyPair) : bool
StoreECIESKey(transaction, ECIESKeyPair) : bool
DeriveKey(logos::raw_key prv, transaction, string password) : void
SleeveKey(logos::raw_key prv, transaction) : void
Checksum(transaction) : Byte32Array
Salt(transaction) : Byte32Array
Fan sleeve_key_mem
Fan password
f kdf
handle
_mutex mutex
special LMDB keys
Byte32Array sleeve_key
Byte32Array salt
Byte32Array checksum
Byte32Array bls
Byte32Array ecies
Fan
vector<unique_ptr<Byte32Array>> values
mutex mutex
Fan(Byte32Array key, size_t size)
ValueGet(logos:raw_key prv) : void
ValueSet(logos::raw_key prv) : void

7. Scenarios and Sequence Diagrams

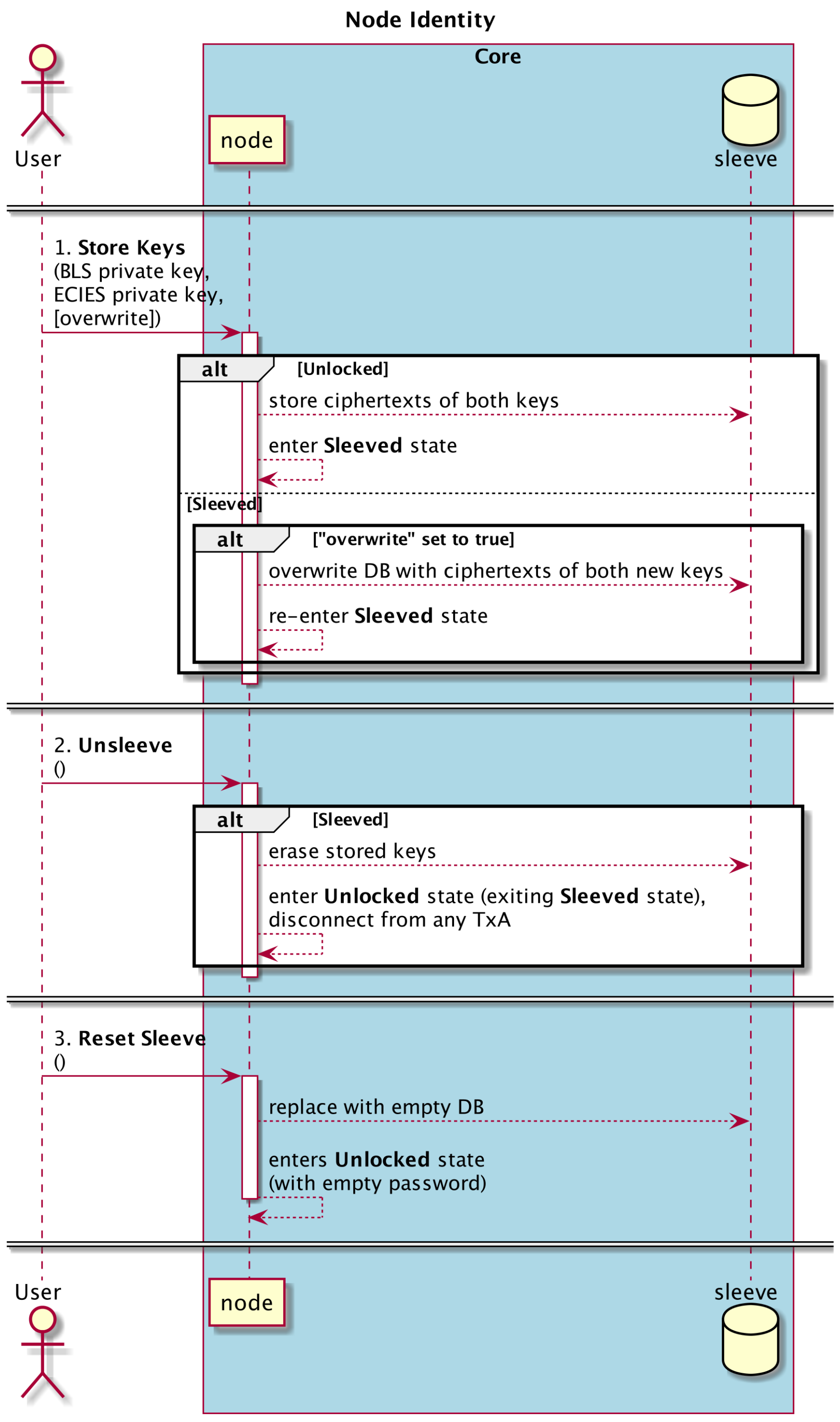
7.1 Command Sequences

This section covers the software's behavior in response to various node operator commands, which are received through secure RPC and passed to the `node` class.

7.1.1 Sleeve Management


Sleeve Management
Core
User
node
sleeve
1. Startup
query
alt [sleeve DB exists]
enters Locked state
initialize with empty DB
enters Unlocked state (with empty password)
2. Unlock Sleeve (password)
alt [Locked]
query
alt [password is correct]
enter Unlocked state
query
alt [sleeve DB contains BLS + ECIES keys]
enter Sleeved state (see Node Identity)
3. Lock Sleeve ()
alt [Unlocked / Sleeved]
enter Locked state, disconnect from any TxA
4. Update Password (new password, [overwrite])
alt [Unlocked / Sleeved]
update encrypted master key with new password
alt ["overwrite" set to true]
replace with empty DB, store new encrypted key with new password
User
node
sleeve

7.1.2 de Identity Setup

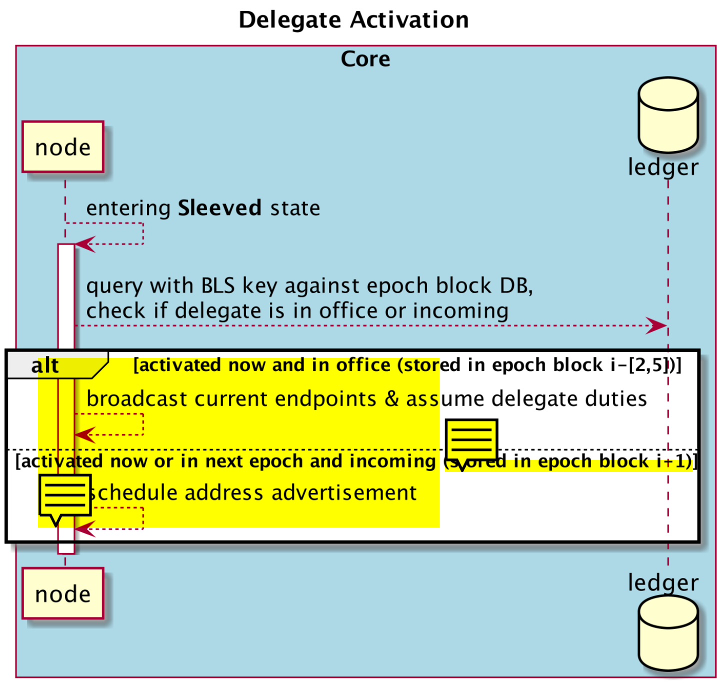

Node Identity
Core
User
node
sleeve
1. Store Keys (BLS private key, ECIES private key, [overwrite])
alt [Unlocked]
store ciphertexts of both keys
enter Sleeved state
[Sleeved]
alt ["overwrite" set to true]
overwrite DB with ciphertexts of both new keys
re–enter Sleeved state
2. Unsleeve ()
alt [Sleeved]
erase stored keys
enter Unlocked state (exiting Sleeved state), disconnect from any TxA
3. Reset Sleeve ()
replace with empty DB
enters Unlocked state (with empty password)
User
node
sleeve

7.1.3 Delegate Activation

Delegate activation setting provides node operators the flexibility to deploy reliable delegate node infrastructure as they see fit. The same delegate identity can be seamlessly migrated between physical nodes without interruption to consensus participation, and each node's load-balancing capabilities can be modified in real time.

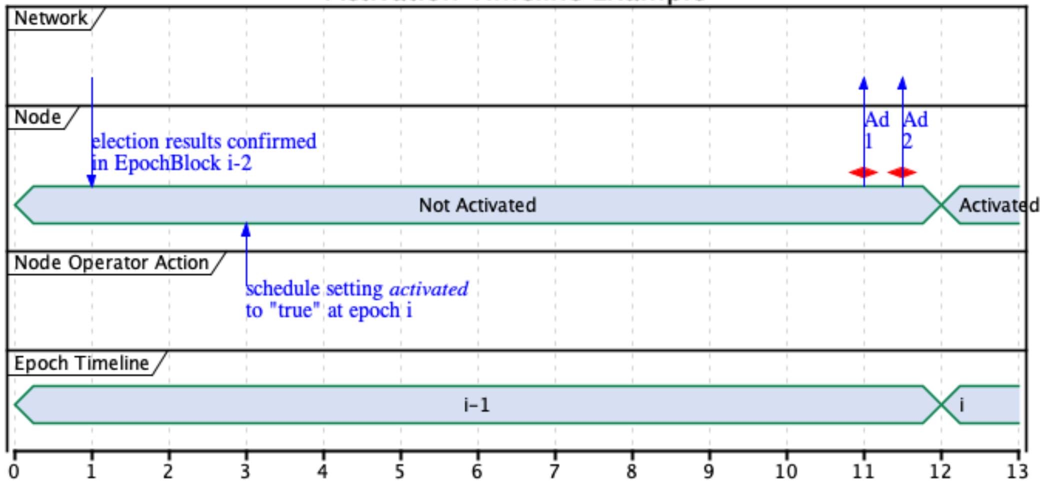
Detailed execution sequences are explained here. It is important to note that a scheduled activation change at epoch `i` can begin taking certain effects in epoch `i-1`.

If a node is activated but is not sleeved, or sleeved but the governance identity is not incumbent or incoming, then the activation status has no immediate effect. When a node enters or re-enters (when an old identity is overwriten with a new one) *Sleeved* state, it performs the check below against its activation setting to determine subsequent actions.


Delegate Activation
Core
node
ledger
entering **Sleeved** state
query with BLS key against epoch block DB,
check if delegate is in office or incoming
alt
[activated now and in office (stored in epoch block i-[2,5])]
broadcast current endpoints & assume delegate duties
[activated now or in next epoch and incoming (stored in epoch block i+1)]
schedule address advertisement
node
ledger

activated is set from false to true for a future epoch `i`: the node performs endpoint advertisement according to CheckAdvertise (broadcast some time around 1 hour and 30 minutes before transitioning from `i-1` to `i`) with epoch number `i`, if its sleeve identity is delegate in epoch `i`. If new TxA's are added before scheduled broadcast takes place, the software does nothing immediately, as these new endpoints will be included in the broadcast messages. If, however, new TxA's are added after the 1-hour mark before transition, the software performs adhoc broadcast. Receivers will append the new endpoints to their database entry associated with this delegate-epoch.

Activation Timeline Example

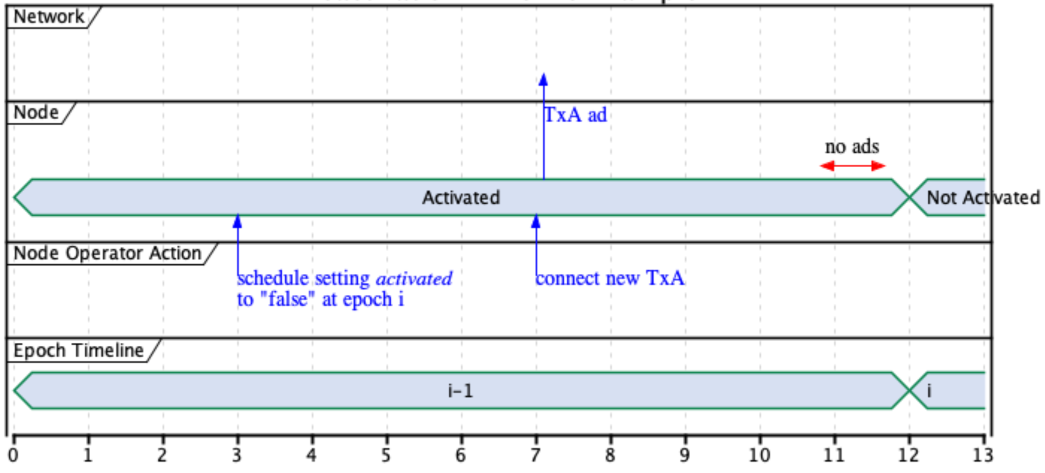


Node is **Sleeved** and is elected for the term starting from epoch i

activated is set from false to true for immediate effect: if the node's identity is a current delegate, the node advertises as if it just started up, immediately broadcasting its endpoints with **current** epoch number.

activated is set from true to false for a future epoch `i` : then the node shall stop performing auto endpoint advertisement according to CheckAdvertise with epoch number `i` starting from epoch `i-1` (i.e. the last time this node can possibly broadcast is towards the end of epoch `i-2` , for its upcoming endpoints for epoch `i-1`). The node, however, still performs delegate actions in epoch `i-1` (but not in `i`) if it's incumbent. Moreover, the incumbent node continues to broadcast newly connected TxA endpoints while in epoch `i-1` (advertisement messages use the current epoch number `i-1`). This enables a node going out of service to still swap in new TxA's during its last epoch if needed.

Deactivation Timeline Example

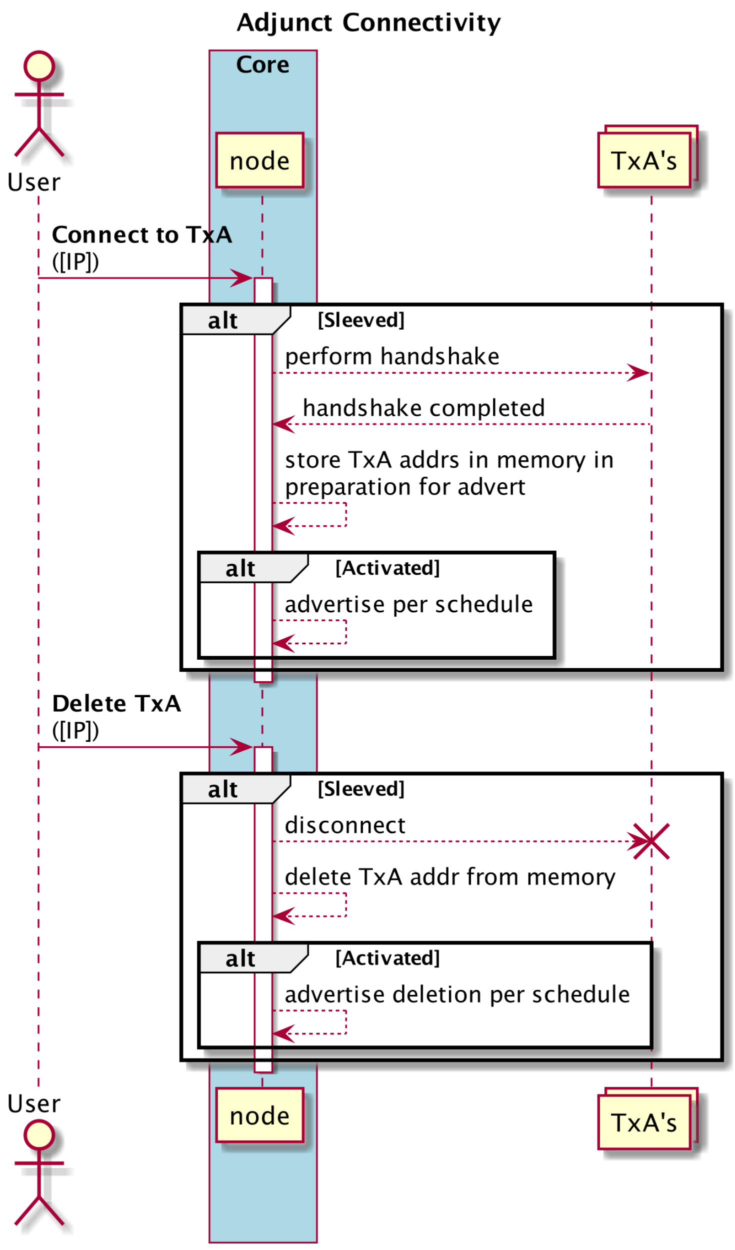


Node is **Sleeved** and is a delegate for all epochs shown

activated is set from true to false for immediate effect: the node broadcasts deletion messages for all its current endpoints (TxA and consensus), and stops performing delegate actions. Note that scheduled future deactivation (above) does not involve endpoint deletion broadcast, since nodes reset their endpoint database entries after each epoch.

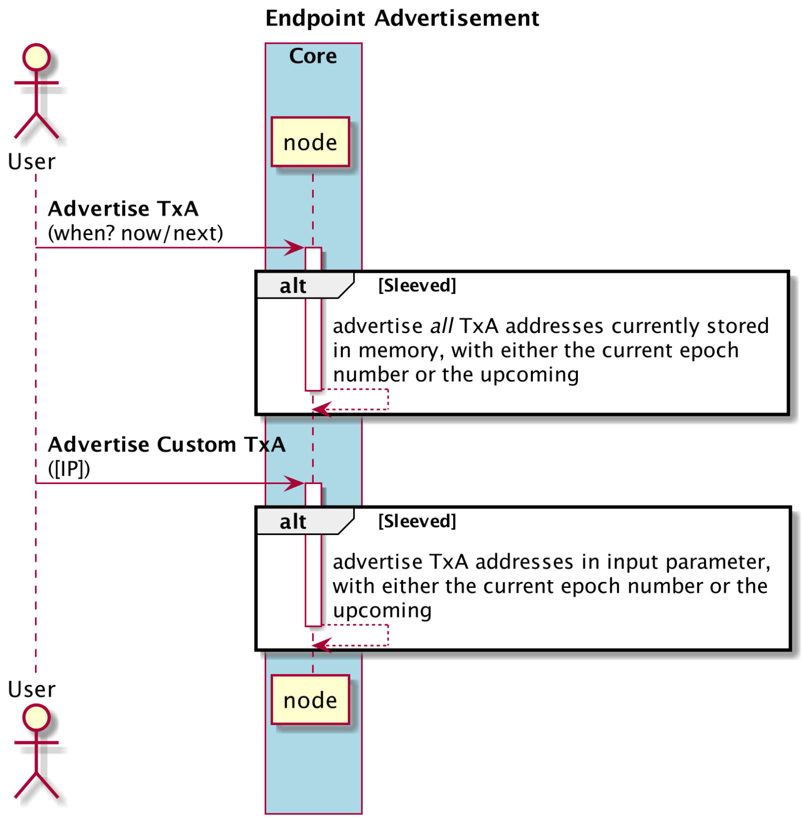
The [illegible]eat here is at no point should two nodes be sleeved with the same delegate identity, which is up to the node operators to ensure. If a "double-sleeving" event occurs, the network may have conflicting views of this delegate's endpoint, as a regular p2p node identifies this delegate with whichever endpoint advertisement it receives first and the order of message arrival is not guaranteed.

7.1.4 [illegible]junct Connectivity



7.1.5 Endpoint Advertisement

Additional commands are provided for ad hoc TxA endpoint advertisement.


Endpoint Advertisement
Core
User
node
Advertise TxA
(when? now/next)
alt
[Sleeved]
advertise *all* TxA addresses currently stored in memory, with either the current epoch number or the upcoming
Advertise Custom TxA
([IP])
alt
[Sleeved]
advertise TxA addresses in input parameter, with either the current epoch number or the upcoming
User
node

In a potential future implementation of consensus gateways, similar designs for connectivity and endpoint advertisement can be adopted.

7.2 Use Case Scenarios

Two main use cases, 1) fast replacement of TxA's and 2) migrating a delegate's identity across nodes, are covered here. The core software doesn't support scheduled TxA replacement since it is less crucial and can be handled entirely by client-end scheduling.

7.2.1 In-Epoch Replacement of TxA's

In this scenario, we would like to replace immediately a delegate node's TxA’s (already connected and running) with new ones when the delegate is in the middle of performing delegate actions. The node is both `sleeved` and `activated`. We execute the following:

1. “txa connect” adds new TxA’s. The node automatically broadcasts the new endpoints with the current epoch number.
2. “txa delete” instructs the node to broadcast endpoint deletion messages with the current epoch number, and then to disconnect from the specified TxA’s.

3. shut down the disconnected TxA's separately.

Note that some network participants may receive the deletion messages too late and attempt to forward transactions to the now shut-down TxA's. They will learn with certainty, however, of the forwarding failure through client-side network errors.

7.2.2 Delegate Migration

In this scenario, we would like to migrate one delegate (identified by a persistent BLS key) from one node to another in between terms.

Suppose we wish to switch from node `a` to node `b` between term 0 (epoch 0-3) and term 1 (epoch 4-7). Suppose further we are in epoch 2, currently node `a` is *sleeved* and *activated* , and node `b` is *locked*, *not activated*. We execute the following:

1. unlock node `b` and store delegate's governance keys on node `b`. Node `b` is now *sleeved* but *not activated*.
2. connect node `b` with its own set of TxA's. These endpoints are not yet broadcast.
3. schedule node `a` to set `activated` to "false" after transitioning from epoch 3 => 4. Node `a` will not broadcast its upcoming endpoints for epoch 4 while in epoch 3, per *Section 7.1.3* ; however, if node `a` connects to additional TxA's in epoch 3, those additional endpoints are broadcast immediately, with the recipients knowing this message is only valid for the current epoch)
4. schedule node `b` to set `activated` to "true" after transitioning from epoch 3 => 4. Node `b` broadcasts its upcoming endpoints for epoch 4 during epoch 3 per the pre-defined advertisement schedule.

We will observe the same behavior even when we perform the above action sequence in epoch 3 instead of 2. If the above is executed too late (after node `a` already broadcast its endpoints for epoch 4), however, we may instruct node `a` to broadcast an endpoint delete message, in addition to executing the above.

8. Resources

A new sleeve database, `sleeve.ldb` , is created to store secret identity key data in encrypted format (see *Section 9* for details), separate from `data.ldb` , which contains ledger data. Currently, the sleeve database only stores at most one identity key, since a given node can be identified with at most one governance identity.

The database contains the following special keys (see usage in *Section 9*):

Key	Definition
`sleeve_key`	master secret key to the database, in encrypted format
`salt`	salt for encrypting master key and checksum, as well as for key-derivation function
`checksum`	value for verifying password validity
`bls`	governance BLS key, in encrypted format
`ecies`	governance ECIES key, in encrypted format

9. Implementation and Algorithms

9.1 Sleeve Access & Key Management

The sleeve serves as a data container for secret identity key data. Users have the option to lock or unlock the sleeve, update sleeve password, store identity key, as well as to re-initialize the sleeve in case the current password cannot be recovered. This section provides detailed explanation on *database*-related implementations.

We use AES encryption with counter (CTR) mode of block cipher operation for symmetric-key encryptions, and Argon2 as key derivation function.

9.1.1 Initialization / Reset

Upon node initialization, the node creates an empty sleeve database if one is not already present. On sleeve reset command, the node replaces the existing database with an empty one. After the empty DB is created, initialization proceeds as follows:

1. Generate a random (using an OS-provided RNG, same for all random generation below) salt (256-bit), and store it under `salt` mdb key.
2. Generate a random sleeve master key (256-bit).
3. Initialize an empty password (all `0` s, 256-bit) and store in memory using fan-out (see below).
4. Encrypt master key with initial password and the first half of salt as IV (AES blocks are 128-bit); store the resulting ciphertext under `sleeve_key` mdb key, as well as in memory using fan-out.
5. Encrypt an all-zero value (256-bit) with the master key and the same IV as above; store the resulting ciphertext under `checksum` key (This order of operation (encrypt-then-MAC) is chosen for its security benefits)
6. Derive an encryption key from an empty password string and full salt (256-bit), decrypt the master sleeve key, then encrypt it again with the new derived key, and store the new ciphertext under `sleeve_key` .

The in-memory password is now reset to the argon2 key-derived value of the empty string "".

Note that, after the initialization process is complete, the raw sleeve master key is cleared from memory.

Note on Fan-out Implemtation

The purpose of the fan-out operation is to create an extra speed bump for a potential attacker, with the effect that it is difficult to discover the password even when a malicious process reads the Logos core software process memory (e.g. through `ptrace()` when run under the same user ID or as `root`). This does not fundamentally mitigate the attack, however.

To store a secret `key`, the fan-out data structure uses `count+1`-sized vector of smart pointers (where `count` can be 1024), each referencing a chunk of data. On initialization, the first `count` values are randomly generated, and the last field is the resulting value of `key` repeatedly XOR'd with each of the previous values.

To retrieve a stored `key`, we simply initialize an empty value and XOR it with the `count+1` stored values.

Whenever a new `key` value is subsequently set, we retrieve the old value (per above), XOR the vector's first referenced value with it (so that at this intermediary step, XOR-ing all values would yield an empty value), then XOR the first value again with the new `key` (modifying the 1st element twice in total).

9.1.2 Password Input

The validity of a password is verified by

1. decrypting sleeve master key (`sleeve_key`) with attempted password and IV (the first half of the value under `salt`), and
2. calculating a new checksum.

The password is valid if the new value matches the one stored in the database (`checksum`), in which case the software will also store the password in memory using fan-out.

9.1.3 Password Update (Rekey)

If the sleeve is currently *unlocked* / *sleeved*, the software

1. derives a new encryption key with new password (and salt),
2. decrypts the sleeve key with the old, in-memory derived key (and IV),
3. encrypts it with the new derived key and IV, and
4. updates the `sleeve_key` database entry as well as the in-memory derived key.

If the sleeve is *locked* and an "overwrite" option is provided, the software repeats the same steps from *Section 9.1.1*, with the exception that it uses the new password in step 6 (instead of the empty string "").

Otherwise, the operation fails.

9.1.4 Governance Key Storage

On user-provided private key input (for both BLS and ECIES), the software

1. generates the corresponding public key,
2. encrypts the plaintext private key with sleeve master key and the first 128 bits of the public key as IV,
3. store (under either `bls` or `ecies`) the tuple (public key, ciphertext), and
4. pass the key pair to IdentityManager in plaintext.

To produce the plaintext private key in an unlocked sleeve, we simply use the sleeve master key and the IV from the public key.

We may store the plaintext keypair in-memory using fan-out as well, but the decision depends on the magnitude of the performance loss as a fraction of BLS / ECIES signing time. Benchmarking will be performed.

For other sensus-related actions potentially triggered by successful key storage, see *tion 7.1.2*.