Upgrading a subnet

IPC Upgrades

As IPC evolves, upgrades are necessary to introduce new features, enhance security, improve scalability, and address new challenges. However, it is critical that all upgrades seamlessly transition without causing forks or consensus failures. This ensures that all nodes in the network remain synchronized.

In this document, we describe how IPC achieves full upgradability of all its components and ensures seamless progression of the blockchain network without compromising consensus.

IPC Actor upgrades

The on-chain logic for IPC is implemented in two contracts, the IPC gateway and the subnet actor, as well as an auxiliary contract use for UX purposes, the registry.

- The gateway is a singleton actor that lives in every IPC subnet and implements the common logic for IPC. It is responsible for managing the collateral of subnet, enforcing the firewall requirement, and the cross-net interactions.
- The subnet actor is a user-defined actor that implements the specific logic of a subnet. This contract is deployed in the parent from which the child subnet wants to be deployed. There is one subnet-actor for each child subnet in the parent.
- The subnet registry behaves as a subnet actor factory that offers users a convenient way of deploying instances of the reference implementation of the subnet actor in a network.
-

The IPC actors are implemented using the diamond pattern to make it easier to upgrade them. The following sections describes how to upgrade each of these actors

Upgrading the Subnet actor

Once you have youdeployed your own subnet you might want to make changes to the subnet actor, for example make custom logic changes, add new functionality, fix bugs, etc.

The code for the subnet actor is located incontracts/src/subnet which you can directly edit to make your changes. Once ready, you run the following steps to upgrade your already deployed subnet actor:

1. First, you must know your subnet ID which was returned when you created a child subnet (usingipc-cli subnet create
2.). Lets say when you created your subnet that your subnet ID was:/r314159/t410fkp4r67rks3ok4bvbn2rjojhkbig2rwvp4nmor5q
3. We must convert the subnet ID to an ETH address. To do this, visit theBeryx address converter
4. and input thet410
5. address in the Filecoin address input, in this example this would bet410fkp4r67rks3ok4bvbn2rjojhkbig2rwvp4nmor5q
6. . ClickConvert to ETH
7. and it will compute the Ethereum address as0x53f91f7e2a96dcae06a16ea29724ea0a0da8daaf
8. .
9. You must set theRPC_URL
10. andPRIVATE_KEY
11. environmental variables to point to your network provider and the private key of the address you want to use for the deployment, respectively
12. Now you should be able to upgrade the subnet actor by running the following command (change NETWORK if you have deployed your subnet on another network)
13.

```

Copy

    cd contracts/ make upgrade-sa-diamond
    SUBNET_ACTOR_ADDRESS=0x53f91f7e2a96dcae06a16ea29724ea0a0da8daaf NETWORK=calibrationnet

```

A successfull output looks like:

```

Copy Successfully generated 84 typings! Compiled 14 Solidity files successfully (evm target: paris).

## Facet Bytecode Not Found:
```

Facet Name: SubnetActorGetterFacet Libraries: - SubnetIDHelper: 0x1CaA8B5a18d0401f2089aC82063AaecA08142744 Address: 0xcb4EE21d081e0cBdd3D82D94114bEAa497f11C4a

Diamond Facet Upgrade:

Diamond Address: 0x53f91f7e2a96dcae06a16ea29724ea0a0da8daaf Replacement Facet Name: SubnetActorGetterFacet

Deployment Status:

New replacement facet (SubnetActorGetterFacet) deployed.

Facet Bytecode Not Found:

Facet Name: SubnetActorCheckpointingFacet Libraries:

Address: 0xd5C97F52030DFE7595b6cD92868EcE5dC175C224

Diamond Facet Upgrade:

Diamond Address: 0x53f91f7e2a96dcae06a16ea29724ea0a0da8daaf Replacement Facet Name: SubnetActorCheckpointingFacet

Deployment Status:

New replacement facet (SubnetActorCheckpointingFacet) deployed.

```

If you run themake upgrade-sa-diamond command again, you should see no output which would also confirm that the upgrade has succeeded.

Fendermint upgrades

Fendermint incorporates a builtinUpgradeScheduler , enabling the execution of hardcodedUpgrade migrations at predetermined block heights to advance the on-chain state. Fendermint also supports functionality to halt at predeterminedhalt_height in order to switch binary versions.

Upgrade scheduler

At the start of processing each block, Fendermint checks theUpgradeScheduler whether it contains anUpgrade for that block height. Only a singleUpgrade can be scheduled for each block height. If an upgrade is found to be scheduled, it will execute themigration function that is associated with thatUpgrade . If the upgrade migration returns an error it will crash the node.

The migration has access to thestate_tree andblockstore which allows the user to write migrations such as patching state, sending messages, deploying solidity contracts, deploying new WASM actors, etc.

AnUpgrade is defined as follows:

```

Copy /// a function type for migration pubtypeMigrationFunc=fn(state:&mutFvmExecState)->anyhow::Result<()>;

pubstructUpgrade{ chain_id:ChainID, block_height:BlockHeight, new_app_version:Option, migration:MigrationFunc, }

```

Fields:

- chain_id
- : The chain id the upgrade should be applied to.
- block_height
- : The block height where the upgrade should be applied to and themigration
- function executed.
- new_app_version
- : Indicates a new application protocol version if the upgrade introduces backward-incompatible changes.
- migration
- : The migration function that is executed when applying the upgrade. The migration function is passed theFvmExecState
```

- which gives access to thestate_tree
- ,block_store
- , ability to send messages and more.
-

Fendermint supports scheduling multiple upgrades through itsUpgradeScheduler API which is defined as follows:

```

Copy structUpgradeKey(ChainID,BlockHeight);

pubstructUpgradeScheduler{ upgrades:BTreeMap, }

implUpgradeScheduler{ // add an upgrade to the upgrade schedule fnadd(&mutself, upgrade:Upgrade)->Result<()> { // implementation left out }

// check if the current chain_id has a scheduled upgrade at the given height fnget(&self, chain_id:ChainID, height:BlockHeight)->Option<&Upgrade> { // implementation left out } }

```

As an example, if we want to create anUpgradeScheduler with a singleUpgrade , we could write something like:

```

Copy // create a new upgrade scheduler letmutscheduler=UpgradeScheduler::new();

// add a new upgrade at predetermined block height letupgrade_func=|state:&mutFvmExecState|{ // implement our upgrade logic here Ok() } letupgrade=Upgrade::new(chain_name, block_height, app_version, upgrade_func); scheduler.add(upgrade);

// when initializing the FvmMessageInterpreter, specify the upgrade schedule letinterpreter=FvmMessageInterpreter:::new( ... scheduler, );

```

We have written guides showing several examples of using the UpgradeScheduler API, which you'll find on the sidebar.

Halting at predetermined height

Although theUpgradeScheduler supports multiple different types of upgrades, it can not support non-state related changes such as upgrading Fendermint dependencies (such as FVM), adding new syscalls, etc.

To support these use cases, Fendermint includes ahalt_height config which when set (is non zero), will halt and exit Fendermint with a specific exit code (2) once it reaches that block height.

This enables node operators to replace the Fendermint version at a predetermined future block height with a new Fendermint version containing the new upgrade. Optionally, the new version can specify a newUpgrade at the specifiedhalt_height in case a state migration is also required.

This ensures that all nodes run the same Fendermint version for every block height which prevents accidental forking or consensus failures.

Instructions

These instructions detail the steps for utilizing the halting mechanism to synchronize Fendermint application switching with newer versions.

NOTE: Node operators should always check the fendermint exit code. If fendermint exits with code 2, which indicates reaching the halting height, disable automatic restarts. Instead, follow steps 4 and 5 in this section. 1. Determine the future halt_height

Node operators collaborate on which future block height should be used ashalt_height . Selecting the halting height is totally up to the operators, but let say for example purposes that the halt_height was agreed to be set to10000 .

1. Update the halt_height config

Now, each operator needs to edit the Fendermint config (.fendermint/config/default.toml) and make sure it contains the following line

```
```

Copy

    cat .fendermint/config/default.toml|grephalt_height halt_height=10000

```

Note that Fendermint needs to be restarted before the config changes take effect.

1. Wait until halt_height

Once Fendermint reaches thehalt_height , it will exit and log it reached the halting height.

```

Copy ... 2024-03-19T13:57:52.904824Z INFO fendermint/app/src/app.rs:677: event=ProposalProcessed is_accepted=true block_height=9999 block_hash="55E10F55140A995CEE4B540776EF5916CBC5C9D81137FED6788C82F2ACBEA28D" num_txs=0 proposer="BA9ADDB0F298912DB80757EA743C3672AE918FF8" 2024-03-19T13:57:53.006128ZINFOfendermint/app/src/app.rs:786:event=NewBlockblock_height=449 2024-03-19T13:57:53.980298Z INFO fendermint/app/src/app.rs:677: event=ProposalProcessed is_accepted=true block_height=10000 block_hash="7B5830D4F138D4329909B64AE0A71296271AD65C83158DDED59DEB3C01DDC650" num_txs=0 proposer="BA9ADDB0F298912DB80757EA743C3672AE918FF8" 2024-03-19T13:57:54.036789ZINFOfendermint/app/src/app.rs:701:Stoppingnodeduetoreachinghaltheightheight=10000

```

The node operator can check if Fendermint exited due to reaching halting height by checking if its exit code is2 .

```

Copy

    echo ? 2

```

1. Reset the halt_height

We must change the Fenderminthalt_height to 0 (or some future block height if there is another upgrade planned).

```

Copy

    cat .fendermint/config/default.toml|grephalt_height halt_height=0

```

1. Start the new Fendermint version

We can now start the new version of Fendermint which contains the upgrade we need.
```