ChatGPT

Implementing a Private IPFS Cluster for a DID System

Overview and Key Components

This guide explains how to set up a **private IPFS network** and an **IPFS Cluster** across multiple Linux machines, tailored for a decentralized identity (DID) use-case. In a private IPFS network, nodes only connect to peers sharing a secret key (ensuring data isolation) [1]. An IPFS Cluster is a separate service that coordinates automatic pinning and replication of content across these IPFS nodes [2]. By combining a private IPFS swarm with an IPFS Cluster, we achieve controlled, secure storage for DID documents and related data, with content persistence and replication. The steps below cover installation, configuration for privacy, cluster setup, and DID-specific best practices.

1. Install IPFS (Kubo) on Each Node

Step 1: Download and install Kubo (go-IPFS) on each Linux machine. You can fetch the latest IPFS binary from the official distribution site and install it:

- Use `wget` to download a release (e.g., v0.15.0 for 64-bit Linux) [3] [4]:

```
wget https://dist.ipfs.tech/kubo/v0.15.0/kubo_v0.15.0_linux-amd64.tar.gz
tar -xvzf kubo_v0.15.0_linux-amd64.tar.gz
cd kubo && sudo bash install.sh
```

This installs the `ipfs` binary to `/usr/local/bin`. Verify the installation with `ipfs --version` [5].

- *Alternative:* On Debian/Ubuntu, you may use APT repositories or snaps, but using the official binary ensures you get the latest release. Ensure each machine has **golang** installed if you plan to build from source (not required when using pre-built binaries).

Step 2: Initialize the IPFS node on each machine:
- Run `ipfs init` (for a default config) or use the "server" profile for a headless node:

```
ipfs init --profile server
```

Using the `server` profile is recommended for servers/VMs as it disables MDNS (local peer discovery) and some GUI-related settings [6]. This creates a `~/.ipfs` directory containing the node's identity keys and config.

- Note the **Peer ID** of each node (displayed as "peer identity" after init). You can always retrieve it with `ipfs id -f="<id>\n"`.

Step 3: Run the IPFS daemon (for now, just to test):

```
ipfs daemon --init-profile server
```

This should start the node (we will stop it later to apply private config). At this point, the node would default to the public IPFS network – we will restrict it in the next steps.

2. Configure a Private IPFS Network (Swarm)

Now we'll set up a **private IPFS swarm** so that only our nodes communicate with each other, isolating them from the public IPFS network:

- **Generate a swarm key:** On **one** node (e.g., the designated bootstrap node), generate a 32-byte secret key that all cluster nodes will share. You can use the official swarm key generator utility:

  ```
  go get -u github.com/Kubuxu/go-ipfs-swarm-key-gen/ipfs-swarm-key-gen
  ipfs-swarm-key-gen > ~/.ipfs/swarm.key
  ```

  This produces a `~/.ipfs/swarm.key` file (a 32-byte pre-shared key) [7]. If you don't have Go installed to build the tool, you can use a provided script or manually create a key (the key format is documented, but using the tool is simplest).

- **Distribute the swarm key:** Copy the `swarm.key` file to the IPFS config directory on **every node** (e.g., place it at `~/.ipfs/swarm.key` for the IPFS user on each machine) [8] [9]. All nodes must have the identical key file. **Never share this key publicly** – it keeps your network private.

- **Remove public bootstraps:** By default, IPFS nodes have a list of public bootstrap peers. Remove them on every node to prevent any public network dialing:

  ```
  ipfs bootstrap rm --all
  ```

  This ensures the node won't connect to the global IPFS network [10].

- **Set private bootstrap peers:** Choose one node to act as the *bootstrap peer* for your private cluster (e.g., Node0 with IP `192.168.0.1`). On each node (including Node0), add Node0's address as the sole bootstrap peer:

  ```
  ipfs bootstrap add /ip4/<Node0_IP>/tcp/4001/ipfs/<Node0_PeerID>
  ```

Replace `<Node0_IP>` and `<Node0_PeerID>` with Node0's actual IP address and peer ID (the long Qm... or 12D3... string from `ipfs id`) [11]. This command seeds each node with the address of Node0 so they can find it on startup. When Node0 comes online, all others will connect to it (and transitively to each other). *Tip:* If you have multiple nodes, you can similarly add additional peers to each node's bootstrap list (or later use `ipfs swarm connect`), but usually one or two well-known bootstrap nodes suffice.

- **Force private network mode:** Set the environment variable `LIBP2P_FORCE_PNET=1` on each machine (or in the shell/systemd service) before running the daemon. This is a safety check – IPFS will refuse to start with a swarm key unless this variable is set, to confirm you intend to run a private network [12].

- **Update IPFS config (optional):** Open `~/.ipfs/config` on each node and adjust the API/Gateway listen addresses if needed. By default, the API and gateway bind to localhost. If you want to access the IPFS API remotely (for cluster control) or allow HTTP gateway access to DID documents:

- Set `Addresses.API` to `/ip4/0.0.0.0/tcp/5001` (or a specific network IP) [13] [14].
- Set `Addresses.Gateway` to `/ip4/0.0.0.0/tcp/8080` if you want the gateway accessible on the network [15].
  These changes let the cluster service (running on the same host) talk to the IPFS API, and allow you to fetch data via gateway from other machines if needed. (Skip gateway exposure if not required, or secure it behind a firewall since this is a private setup.)

- **Start the IPFS daemons:** Now start `ipfs daemon` on all machines. All peers should log that they are in a protected/private network. Each node will connect to the bootstrap peer(s) and form an isolated DHT. You can verify connectivity by running `ipfs swarm peers` on any node – it should list the other nodes' addresses (each node should see the others in the swarm) [16]. Also, try adding a test file on one node and cat-ing it on another:

```
echo "Hello IPFS" > hello.txt
ipfs add hello.txt   # yields a CID, e.g., QmXYZ...
ipfs cat <CID>       # run this on a *different* node
```

If the second node returns the file content, your private IPFS network is functioning [17] [18]. Additionally, confirming that the CID **cannot** be fetched via a public gateway (like `ipfs.io/ipfs/<CID>`) is a good test that your data is private to your network.

Best Practice: Consider running the IPFS daemon as a system service for resilience. For example, create a systemd service unit `/etc/systemd/system/ipfs.service` running `ExecStart=/usr/local/bin/ipfs daemon --enable-namesys-pubsub` (with `Restart=always`, run as a non-root user) [19]. The `--enable-namesys-pubsub` flag is explained later (it helps with IPNS performance). Enable and start the service with `systemctl enable ipfs && systemctl start ipfs`. This ensures the node stays online to serve DID content.

3. Install IPFS Cluster Service and CLI

Next, install the IPFS Cluster components on each machine. IPFS Cluster consists of two binaries: `ipfs-cluster-service` (the cluster peer daemon) and `ipfs-cluster-ctl` (the command-line tool to control the cluster) [20] [21].

- **Download IPFS Cluster binaries:** From the IPFS distribution site, download the latest cluster-service and cluster-ctl tarballs. For example, for version 1.0.4:

```
wget https://dist.ipfs.tech/ipfs-cluster-service/v1.0.4/ipfs-cluster-service_v1.0.4_linux-amd64.tar.gz
wget https://dist.ipfs.tech/ipfs-cluster-ctl/v1.0.4/ipfs-cluster-ctl_v1.0.4_linux-amd64.tar.gz
tar xvfz ipfs-cluster-service_v1.0.4_linux-amd64.tar.gz
tar xvfz ipfs-cluster-ctl_v1.0.4_linux-amd64.tar.gz
sudo cp ipfs-cluster-service/ipfs-cluster-service /usr/local/bin/
sudo cp ipfs-cluster-ctl/ipfs-cluster-ctl /usr/local/bin/
```

  This installs the cluster service and ctl executables system-wide. Verify by running `ipfs-cluster-service --version` and `ipfs-cluster-ctl --help` [22] [23]. *(Alternatively, you can build from source via `git clone` and `make install` [24], but using the pre-built binaries is quicker for experimentation.)*

- **Generate a Cluster secret:** Similar to the swarm key for IPFS, the cluster can use a secret key to **encrypt and isolate cluster communication**. We'll create `CLUSTER_SECRET`, a 32-byte hex string that all cluster peers share. On one node (Node0), generate it with:

```
export CLUSTER_SECRET=$(od -vN 32 -An -tx1 /dev/urandom | tr -d ' \n')
echo $CLUSTER_SECRET
```

  This will print a long hex string (e.g., `9a420e...44038f`) [25]. Copy that value and set it as an environment variable on **all** nodes (you can add `export CLUSTER_SECRET=<hex>` to `~/.bashrc` for persistence) [26] [27]. Having the same `CLUSTER_SECRET` on each peer ensures they recognize each other as part of one cluster and encrypt their RPC communication [28] [29]. *Important:* Keep this secret safe; without it, cluster peers won't talk, and conversely, any node with it could attempt to join your cluster.

4. Initialize and Launch the Cluster Daemons

With IPFS daemons running and cluster binaries installed, we can form the cluster:

- **Initialize the first cluster peer (Node0):** On Node0, run:

```
ipfs-cluster-service init
```

This generates a default cluster configuration at `~/.ipfs-cluster/`. Notably, it creates `service.json` (the cluster config file), `identity.json` (containing Node0's cluster peer ID and private key), and an empty `peerstore` file [30] [31]. Because we set `CLUSTER_SECRET`, the generated `service.json` will have that secret value (all peers must match) [32]. The cluster uses a consensus algorithm (now default is *CRDT*) to manage pins; CRDT allows dynamic peers and is suitable for most setups (no single "leader" is required) [33].

- **Start Node0's cluster service:** Still on Node0, ensure the IPFS daemon is running, then start the cluster daemon:

```
ipfs-cluster-service daemon
```

You should see logs indicating it's ready (e.g., "IPFS Cluster is ready") [34]. Node0 is now acting as the first cluster member (if you run `ipfs-cluster-ctl peers ls` now on Node0, it should list just itself).

- **Initialize other cluster peers:** For each additional node (Node1, Node2, etc.), do the following:

- **Ensure IPFS daemon is running** (and has the private swarm configured).
- **Set the** `CLUSTER_SECRET` **env** (same value as Node0).
- **Run** `ipfs-cluster-service init` on that node to generate its config and identity [35] [36]. This will create a new cluster peer ID for that node.

- **Start the cluster daemon with bootstrap:** To join the existing cluster, run:

```
ipfs-cluster-service daemon --bootstrap /ip4/<Node0_IP>/tcp/9096/ipfs/
<Node0_ClusterPeerID>
```

Replace `<Node0_IP>` with Node0's address and `<Node0_ClusterPeerID>` with Node0's *cluster* peer ID (not the IPFS peer ID). The cluster peer ID can be found in Node0's `~/.ipfs-cluster/identity.json` as the `"id"` field [37] (for example, it might look like `QmZjs...` or `12D3Koo...`). This command tells Node1 to connect to Node0's cluster listener (port 9096 by default [38]) and join the cluster [39] [40]. Repeat for all other nodes, each time bootstrapping to Node0's cluster address.

- **Verify cluster membership:** From any node, use the cluster control CLI to check peers:

```
ipfs-cluster-ctl peers ls
```

You should see all nodes listed with their cluster IDs and that each "Sees N other peers" (e.g., each node sees all others) [41] [42]. For example, a peer list might show entries for Node0, Node1, Node2 with their addresses (9096 port) and also the IPFS daemon IDs and addresses (4001 port) for each [43] [44]. This confirms the cluster is formed and aware of all members.

(Under the hood, cluster peers share pin information via the consensus (CRDT or Raft). The `peerstore` *file in* `~/.ipfs-cluster/` *gets updated with addresses of known peers [45] [46]. The cluster secret ensures*

that your cluster's communication is isolated – peers with the wrong secret cannot join or even see the cluster RPC traffic [47] [48] .)

- **Run cluster service on startup:** Similar to IPFS, you may want to run `ipfs-cluster-service` as a systemd service or in screen/tmux for experimentation. Ensure it starts *after* the IPFS daemon. You can use the sample systemd units provided by the cluster project [49] , which include dependencies on IPFS. This way, your cluster will come online automatically with the machines.

5. Pinning and Replicating DID Documents

With the cluster up, you can use it to **store and replicate content** (such as DID documents, schemas, or other identity data):

- **Adding content to the cluster:** Use the `ipfs-cluster-ctl` tool to add and pin files across the cluster. For example, to add a DID document JSON file:

  ```
  ipfs-cluster-ctl add did_document.json
  ```

  This command imports the file into IPFS on one of the cluster nodes and instructs the cluster to **pin** it on the desired number of replicas (by default, all cluster peers will pin it, since the default replication factor is -1 = "pin everywhere") [50] . The output will show the Content ID (CID) of the added file.

- **Verify pinning status:** To check the status of the content on all nodes, run:

  ```
  ipfs-cluster-ctl status <CID>
  ```

  This will show each cluster peer and the pin status (`PINNED`) for that CID [51] . A healthy cluster will report the CID as pinned on each node (or at least on the number of replicas you specify). Now, even if one node goes offline, the content remains available on the others – a key benefit for persistent DID data.

- **Accessing the content:** You can retrieve the pinned content from any node's IPFS, either via CLI (`ipfs cat <CID>`) or via an IPFS gateway URL (e.g., `http://<NodeIP>:8080/ipfs/<CID>` if gateways are enabled). In a DID scenario, this CID could represent a DID document. The cluster ensures that the CID stays available in the network (persistent storage).

- **IPNS for DID document updates:** Many DID systems require that a DID can be updated (without changing the DID string itself). Since IPFS CIDs are content-addresses (immutable once created), you can use **IPNS (Inter-Planetary Name System)** to create a stable DID reference that can point to the latest CID. In practice, a DID method like *did:ipid* uses IPNS: the DID contains a public key (or peer ID) which corresponds to an IPNS record for the DID document [52] [53] . To implement this:

- **Generate an IPNS key for the DID:** You can either use the IPFS node's default identity key or create a new keypair. For example: `ipfs key gen --type=rsa --size=2048 mydidkey` . This will output a new Peer ID (say `QmABC...` or in Base36 `k51...` format). That peer ID can

form part of the DID (e.g., `did:ipid:QmABC...` or with `12D3Koo...` prefix if using ed25519 keys) [54] [55].

- **Publish the DID document CID to IPNS:** On a node that holds the `mydidkey`, run: `ipfs name publish --key=mydidkey /ipfs/<CID>`. This signs and publishes an IPNS record linking the Peer ID (DID) to the content's CID. In a private network, the IPNS record propagates to your private DHT or through pubsub (see below). Now the DID can be resolved by looking up that IPNS key in your network. Any time the DID document changes (new CID), you'd publish a new IPNS record to update the mapping.

- **Enable IPNS PubSub for speed:** Normally, IPNS (even in a small network) can take many seconds because of DHT lookup. Enabling IPNS over PubSub on all nodes makes IPNS updates instantaneous within the cluster. We already started the IPFS daemons with `--enable-namesys-pubsub` (or you can set `Ipns.UsePubsub` in the config). With this, when one node publishes a new IPNS record, it's immediately gossiped to the others [56]. This drastically improves DID update responsiveness – changes propagate in real-time instead of within a minute. (Ensure all nodes run with this enabled, so they subscribe to IPNS topics.)

- **Cluster considerations for IPNS:** Only the node with the IPNS key can directly publish updates for that DID. In a cluster, you might designate one node to handle all DID IPNS publications, or distribute keys and have multiple nodes publish their respective DIDs. All cluster nodes will cache and rebroadcast IPNS records they hear (especially with pubsub). The content linked by the IPNS should be pinned cluster-wide for high availability. In essence, the cluster keeps the data available, and IPNS provides the pointer for DID resolvers to fetch the latest version.

6. Networking and Security Best Practices

To support a DID system, we must ensure the network is reliable, private, and secure. Here are additional tips:

- **DHT isolation:** By using a private swarm key, your IPFS nodes already form an isolated DHT for content routing – they will not query or announce to the public IPFS DHT [1]. This means outsiders can't discover your content via the global network. Keep the swarm key secret to prevent unauthorized nodes from joining your network. You can also set the IPFS config `Routing.Type` to "dhtserver" or "dhtclient" explicitly depending on your node's role, but in a small cluster all nodes can be DHT servers.

- **NAT traversal and relays:** If your nodes are on different networks (behind NATs or firewalls), pure peer-to-peer connections might fail. IPFS private networks do support NAT traversal, but you may need a **relay node** setup [57]. For example, make one node with a public IP act as a relay (enable `Swarm.EnableRelayHop` in its config), and enable `Swarm.EnableAutoRelay` on the other peers so they can use it. Peers that can't dial directly will then connect through the relay. *Note:* Each node should still have the bootstrap address of the others (or at least the relay) so they can find each other. If using relays, you'll see multiaddrs with `/p2p-circuit/` in `ipfs swarm peers`. Ensure your security group/firewall allows the swarm port (4001) and relay port if different. Having "a few publicly reachable nodes to use as relays (for signaling)" greatly improves connectivity for NATed nodes [57].

- **Connection encryption:** All IPFS traffic is encrypted by default (TLS over libp2p), and the private swarm key adds an extra layer of enforced encryption for swarm traffic. Similarly, cluster RPC traffic is encrypted by its secret. This minimizes metadata leakage – eavesdroppers can't see

what you're transferring, and nodes outside the swarm can't handshake or learn about your nodes. That said, some metadata (like a node's existence or listening ports) could be exposed if a node is on a public address. Use firewall rules to only allow connections from your known peers if possible.

- **Avoiding external leakage:** Double-check that **no default bootstrap or DNS** peers remain in your configs. Also consider disabling MDNS in IPFS (`Discovery.MDNS.Enabled=false` in the config) if the nodes are in different networks (MDNS is only useful on local LAN and might be irrelevant or even a privacy concern in some cases). Since we used the server profile at init, MDNS is likely already off. You might also disable the public circuit relay listener (`Swarm.EnableRelayHop=false` on nodes that don't intend to relay others) so that your nodes aren't inadvertently acting as relays to unrelated traffic. In short, **limit your node's peer connectivity to just your trusted cluster peers**.

- **Persistent identifiers (CIDs and DIDs):** One challenge in DID systems is persistence of identifiers. IPFS CIDs are content-addresses (if content changes, the CID changes). To maintain persistent **DID URIs**:

- Use the DID itself (with IPNS or key-based lookup) as the stable identifier and treat IPFS CIDs as versioned content. The cluster will store all versions if you pin them, but the DID method (like IPID) will typically fetch the latest CID via IPNS [58] [56].

- If you need older versions of DID documents (for auditing or recovery), you can either keep them pinned or use IPFS's built-in versioning (e.g., mutable file system / MFS or IPLD links for "previous" document versions as in the IPID spec) [55] [59]. The cluster doesn't automatically version content, but you can add version links in the DID document itself or maintain a separate index.

- **Node identity security:** Each IPFS node and cluster peer has its own identity keys (stored in `~/.ipfs` and `~/.ipfs-cluster/identity.json` respectively). Treat these like credentials. For experimentation you might use the default identities, but in a real DID deployment:

- Run the services under an unprivileged user (e.g., the guide above created an `ipfs` user on Debian [60]). This limits damage if the process is compromised.
- Protect the `.ipfs` and `.ipfs-cluster` directories (set proper file permissions so that only the service account can read them).
- Backup keys if necessary (especially if an IPNS key is the authoritative DID identifier – losing it means losing control of that DID).

- Avoid sharing private keys across nodes; instead, let each node have unique identity. The cluster's shared secret and IPNS pubsub will handle coordination. Only share an IPNS private key to another node if you need a fallback publisher for a DID (and understand the trust implications).

- **Cluster API security:** The cluster exposes an HTTP API (default on port 9094) and an IPFS proxy API (9095) [61]. By default these listen on all interfaces (as configured in `service.json`). If your cluster runs in a closed environment, this is fine. Otherwise, **restrict or secure these APIs**. You can bind them to localhost or a private network interface, and/or enable basic authentication on the cluster API [62]. For example, set `"user":"<user>", "password":"<pass>"` in the `basic_auth_credentials` section of `service.json` if

exposing the API. This prevents unauthorized control of your cluster. Likewise, the IPFS HTTP API (5001) should be firewalled from public access since it can add/get pins.

By following these steps and precautions, you now have a private IPFS cluster suited for DID applications. All DID documents and related data added to the cluster will be replicated across your nodes, ensuring high availability and immutability. The use of IPNS (or other naming layers) allows DIDs to point to dynamic content while the cluster guarantees that content persists (solving the persistence problem of pure P2P storage) [63] [64] . This setup is ideal for experimentation: you can observe how DID resolution works in a closed environment, test key rotations or document updates, and scale out by adding more nodes to the cluster as needed. Happy experimenting with decentralized identity on your IPFS cluster!

Sources:

- IPFS Private Networks and Cluster Setup – ELEKS Labs [65] [66] [67] [68]
- IPFS Cluster Documentation – IPFS Cluster site [31] [32]
- DID on IPFS (IPID Method) – DID IPID Spec & Community discussions [52] [56]
- IPFS Forums and StackExchange – IPNS PubSub and Private Network Tips [57] [69]

---

[1] [2] [7] [8] [9] [10] [11] [12] [13] [14] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [34] [38] [39] [40] [41] [42] [43] [44] [50] [51] [61] [63] [64] [65] [66] [67] [68] Building Private IPFS Network with IPFS-Cluster for Data Replication | ELEKS: Enterprise Software Development, Technology Consulting
https://eleks.com/research/ipfs-network-data-replication/

[3] [15] Why do we need IPFS and how to create private IPFS network | by piash.tanjin | Medium
https://medium.com/@piash.tanjin/why-do-we-need-ipfs-and-how-to-create-private-ipfs-network-c595bf00afc6

[4] [5] [6] [35] [36] [37] [60] Setting Up a Private IPFS Network with IPFS and IPFS-Cluster – GEEKDECODER
https://www.geekdecoder.com/setting-up-a-private-ipfs-network-with-ipfs-and-ipfs-cluster/

[28] [29] [47] [48] Multi node IPFS Cluster on Docker | by (λx.x)eranga | Effectz.AI | Medium
https://medium.com/rahasak/multi-node-ipfs-cluster-on-docker-596085bd07e0

[30] [31] [32] [33] [45] [46] [49] Download and setup - Pinset orchestration for IPFS
https://ipfscluster.io/documentation/deployment/setup/

[52] [53] [54] [55] [59] IPID DID Method
https://did-ipid.github.io/ipid-did-method/

[56] [58] [69] p2p - What should my expectations be regarding IPFS response times? - Stack Overflow
https://stackoverflow.com/questions/63253945/what-should-my-expectations-be-regarding-ipfs-response-times

[57] Does the ipfs private network support NAT traversal ? - Help - IPFS Forums
https://discuss.ipfs.tech/t/does-the-ipfs-private-network-support-nat-traversal/16950

[62] IPFS Tutorial: IPFS Cluster 로 Private IPFS Network 구성하기
https://hihellloitland.tistory.com/89