NDN Technical Report NDN-0076. http://named-data.net/techreports.html
Revision 1: 2022

Hydra - A Federated Data Repository over NDN

Justin Presley, Xi Wang, Tym Brandel, Susmit Shannigrahi
Tennessee Tech
Xusheng Ai, F. Alex Feltus
Clemson University
Proyash Podder, Alex Afanasyev
Florida International University
Tianyuan Yu, Varun Patil, Lixia Zhang
UCLA

Abstract

Today's big data science communities manage their data publication and replication at the application layer. These communities utilize myriad mechanisms to publish, discover, and retrieve datasets - the result is an ecosystem of either centralized, or otherwise a collection of ad-hoc data repositories. Publishing datasets to centralized repositories can be process-intensive, and those repositories do not accept all datasets. The ad-hoc repositories are difficult to find and utilize due to differences in data names, metadata standards, and access methods. To address these problems, some communities use high-level frameworks such as iRODS. However, since these solutions depend on location-oriented TCP/IP protocols, they need to hide the complexity of creating location-independent services over TCP/IP, making them tightly coupled and complex to build and maintain. While storing and accessing all data from commercial cloud platforms could be another solution, such platforms are costly for storing and serving large amounts of data.

To address the problem of scientific data publication and storage, we have designed *Hydra*, a secure, distributed, and decentralized data repository made of a loose federation of storage servers (nodes) provided by user communities. Hydra runs over Named Data Networking (NDN) and utilizes the State Vector Sync (SVS) protocol that lets individual nodes maintain a "global view" of the system. Hydra provides a scalable and resilient data retrieval service, with data distribution scalability achieved via NDN's built-in data anycast and in-network caching and resiliency against individual server failures through automated failure detection and maintaining a specific degree of replication. Hydra utilizes "Favor", a locally calculated numerical value to decide which nodes will replicate a file. Finally, Hydra utilizes data-centric security for data publication and node authentication. Hydra uses a Network Operation Center (NOC) to bootstrap trust in Hydra nodes and data publishers. The NOC distributes user and node certificates and performs the proof-of-possession challenges.

This technical report serves as the reference for Hydra. It outlines the design decisions, the rationale behind them, the functional modules, and the protocol specifications.

1 Introduction

This tech report describes Hydra, a distributed and decentralized data repository system built over NDN. The motivation for this work comes from the needs of scientific communities such as genomics, climate science, and high-energy particle physics that often follow a distributed collaborative model where data is published and accessed by scientists worldwide. Hydra provides an easy-to-use platform for publishing and accessing such datasets through a distributed, federated storage system where a specific community or a project provides individual storage nodes./

This tech report gathers all the Hydra design decisions and the lessons learned through the design process in one place. The report also describes the implementation of Hydra, how it provides users with secure and scalable file publishing and sharing services built using the NDN protocol stack, and our experience of Hydra's first deployment on the FABRIC testbed[1].

Hydra is designed to run over a federation of storage servers provided by different user organizations. Users can publish files to Hydra, which can be shared securely and scalably following defined access policies. Hydra also maintains a consistent "system state" among all the storage servers utilizing the NDN State Vector Sync (SVS) protocol[2]. Hydra automatically replicates files to maintain a desired degree of replication even in the face of individual server failures.

Specifically, Hydra is designed to achieve the following goals.

(1) Reducing scientific communities' data publication and management burden. Currently, data is published through ad hoc systems or centralized repositories, none of

which work well with the rapidly exploding distributed datasets. Hydra provides a decentralized framework that allows scientists to publish datasets easily and scalably.

(2) Enabling fast and reliable data replication across distributed nodes. Once a node fails, Hydra automatically replicates files to the available nodes, and the system can continue to run after the failure without operator intervention.

(3) Improving data repositories' performance and supporting more users and applications by distributing read and write requests to different data nodes.

(4) Improving data findability and reusability of data by addressing data by their names. This approach will allow the communities to spend less time curating data and tracking their locations once they agree on a project-specific namespace.

(5) Incorporating data-centric security in the design. All data in Hydra are signed and publicly verifiable.

Hydra uses the data-sharing model in the genomics community as the primary use case. In this community, data is generated in a distributed manner by various research facilities. The researchers then send the data to central facilities (e.g., repositories hosted by National Center for Biotechnology Information or NCBI) that make them available publicly. Apart from the centralized model, this approach has two problems that impede efficient data sharing. First, data publication through a central repository includes a manual component (checking and processing for quality and conformity to naming standards) and can take a long time (weeks or months) before data is available. Second, these central repositories accept specific types of datasets. Datasets that are not accepted by central repositories have to be published in ad hoc ways, making them very difficult to find and utilize in research. Hydra makes this process automated and simpler through the expressive naming of content. Once datasets are named according to community-accepted standards, they can be immediately published through Hydra and discovered and utilized by the users and workflows.

2 The Design of Hydra

2.1 Design Goals

We aim to build a federated, distributed data repository system over NDN, where individual storage servers are contributed by the users in the community and are geographically distributed. Here are the design goals of Hydra:

- All Hydra operations should use a data-centric security model. Our design secures Hydra as a distributed system and can secure the files stored in Hydra. To ensure the security of the Hydra federation, each Hydra server goes through security bootstrapping process to obtain its security credentials before being deployed from a Network Operation Center (NOC). All the files stored in Hydra are cryptographically signed for authenticity and can be encrypted for confidentiality.
- Hydra should enable nodes to operate under different administrative domains. A Hydra federation should be able to operate despite differences in the amount of resources and policies.
- Hydra should support file insertions with automatic replication and scalable file retrieval.
- The Hydra federation should perform semi-autonomously. Operators can perform infrequent operations such as configuring trust and managing catastrophic failures manually, but normal operations such as nodes joining the federation, recovering from node failure, and replication of data should not need user involvement or operator intervention.
- Other desired performance measures for Hydra include the ability to support (a) publications of large datasets (GB to TB in size), and (b) disk-to-disk throughput of 1Gbps or more.

2.2 Design Assumptions

- The number of nodes in a Hydra system is expected to be tens of nodes in the initial deployment; we hope to gain further insights through experimentation on how well Hydra can scale in terms of the number of nodes.
- In its initial trial deployment, Hydra does not hold the master copy of the data. Instead, it used the storage as a scratch space for publishing short-term data. All data that has not been used in a month will be automatically deleted. This assumption will be re-visited as Hydra becomes mature.
- As a collection of user-contributed storage servers, we assume some nodes may malfunction (that lead to file errors and node unavailability), but there is no malicious node in the federation.
- Each User file insertion request is not an interactive step, which is made infeasible by the potentially long delay in carrying out a file insertion.[1] Instead, the notification regarding success/failure will be provided to the user via a status URI.
- Hydra has a central identity manager (NOC). NOC is where the trust originates. Hydra defines trust schemes based on names and previously defined permissions. Hydra uses Google OAuth to establish identities and

[1] The size of files could be multiple gigabytes, thus uploading files to Hydra may take multiple minutes.

enforce trust schemas. Nodes are pre-authenticated. They use a bundled certificate distributed with the code.
- In Hydra, we assume that the NOC is trusted, and each Hydra node is also trusted after it completes the bootstrapping process.

2.3 Design Approaches

Here are the design approaches for Hydra:

- For all new and existing data under our control, we adopt the principle in naming. For all existing data using URLs as unique names, we gradually remove the "location" semantic in those names, treating them simply as unique data identifiers.
- All user interfaces will be kept simple.
- Operations such as file insertions, deletions, and fetching will be based on the file names.
- The users will not be exposed to the internal workings of Hydra, such as file replication and location information of the storage servers. This is not needed in Hydra since NDN routing allows users to reach the "best" node.
- We take on a decentralized approach in Hydra design, even though there may be an extra cost of decentralization - communication overhead.

3 Hydra Functional Details

Hydra is a federated storage system where participants from a big data community (e.g., genomics) contributes storage. Hydra utilizes NDN to create a federation of storage servers. The Hydra federation allows users to publish data into the system using data names agreed-upon by the community, making them readily available and lowering the barrier of data publication.

Figure 1 outlines Hydra's general structure along with an overview of the functions performed by Hydra. In its simplest form, a Hydra federation has several geographically distributed nodes. The Hydra federation also relies on a NOC that is not part of the federation but distributes certificates to the nodes and publishers.

The process of inclusion of a node into Hydra begins by creating a default route to one of the Hydra nodes and requesting a certificate from the NOC. The NOC authenticates the requests and returns a certificate that the node uses for signing further communication. This completes node bootstrapping. Once the security parameters are bootstrapped a node goes through system boostrapping where it creates its own global view. The command is routed to a Hydra node (decided by NDN routing) that verifies the command and ingests the file. Once the file is ingested, a group message is triggered. Other nodes subscribing receive the notification, update their global view, and some of the nodes start to replicate the file. Each successful replication publishes a group message that allows the nodes to keep track of the degree of replication. The nodes also send out periodic heartbeats to the other nodes so that failure recovery can begin when a node goes offline. Finally, a user or a workflow retrieves a file by sending an Interest with the name of the file. This Interest is routed to Hydra nodes that can return the file.

3.1 Function Overview

As Figure 1 outlines, a Hydra federation has four types of functions:

- Bootstrapping
- System Management
- Storage Management
- User Facing Functions

Figure 2 outlines each of these categories.

3.1.1 *Bootstrapping* can be divided into security bootstrapping and system bootstrapping. Security bootstrapping focuses on getting proper key and certificate information that the node can present to the federation before it is allowed to join. We discuss bootstrapping in Section 6. System bootstrapping focuses on the functionality needed for a new node to join the federation and sync the node's state to the system's state.

3.1.2 *System Management* modules provide the necessary systems level functionality for Hydra. These include maintaining states in the network, publishing and applying updates to the system state, logging, node failure detection, and failure recovery. These functions are at the core of how Hydra operates. Management of system is discussed further in Section 6.

3.1.3 *Storage Management* functions oversee file replication and local file storage. The file replication module replicates files across Hydra nodes and maintains the degree of replication (currently 3). This ensures data loss does not occur even when individual nodes fail. The local file storage module handles the storage functions of a node. These include interacting with the underlying database or filesystem, and freeing up storage by deleting unused files. Further discussion on file replication can be found in Section 7 and local storage can be found Section 3.2.

3.1.4 *User Facing functions* provide user interfaces for various user and application interactions. These functions include file insertion, deletion, retrieval, and system status checks. Hydra exposes name-based APIs for these functions that can be directly used by the users or integrated into separate applications. More details on user facing functions can be found in Section 7.

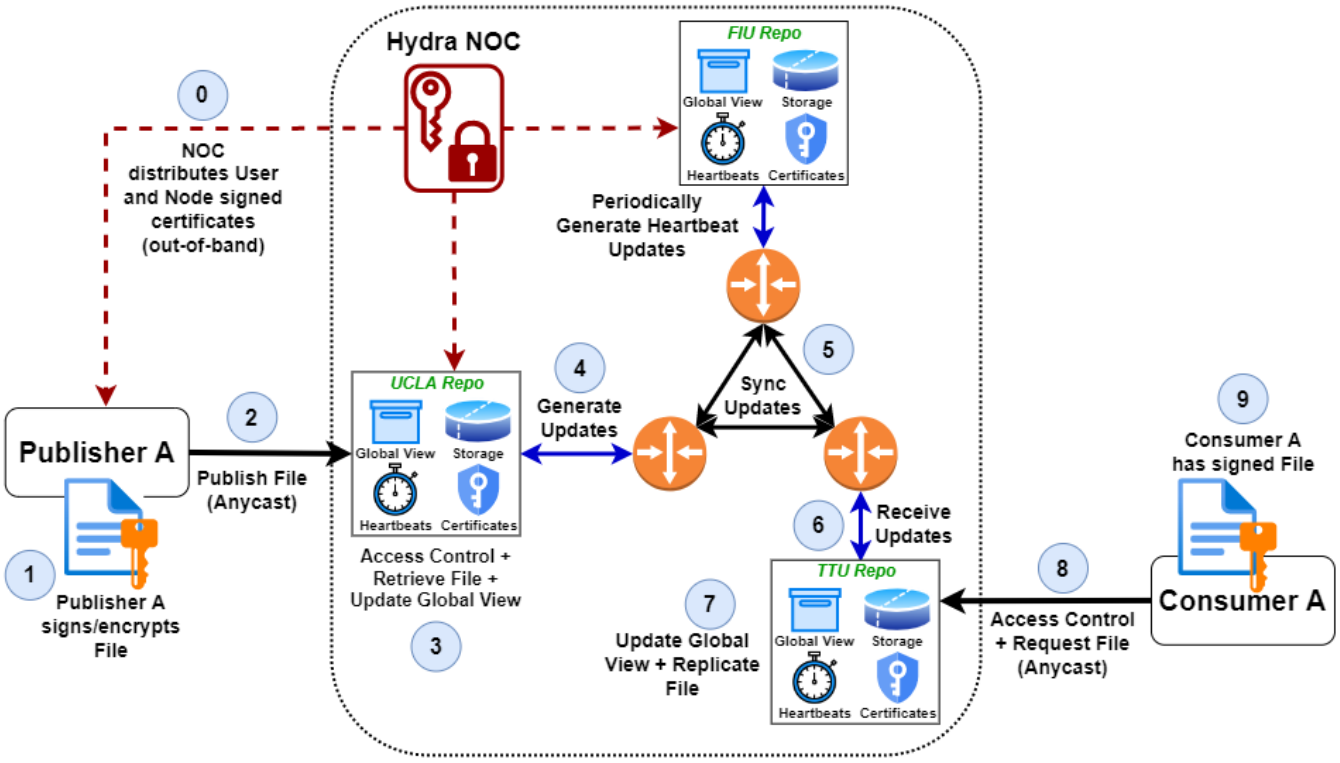

Figure 1: A High-level overview of Hydra

Security	Security Bootstrapping	Certificate Creation + Distribution	Trust Schema	Node and User Access Control	
System Management	Publish Updates	System Bootstrapping	Maintaining State	Logging	Node Failure Detection
Storage Management	File Replication	Local File Storage			
User Facing Actions	File Insertion	File Deletion	File Retreival	File Queries	

Figure 2: Necessary Functions of a Hydra Node

3.2 Modules

To provide the functions described in Subsection 3.1, a Hydra node is made up of several modules as illustrated in Figure 3. Nine modules make up a Hydra node, and the following list briefly describes each of them.

3.2.1 StateVectorSync (SVS) The SVS module uses the SVS protocol to publish data (e.g., state update messages) that is

Figure 3: Modules that make up a Hydra Node

received by all other nodes within Hydra. SVS allows Hydra to maintain a consistent view across the nodes.

3.2.2 Global View The Global View is a local database to a Hydra node. It represents a node's view of the Hydra federation. The global view contains information related to the Hydra system such as files' metadata, node information, and replication information.

It also contains the heartbeat tracker sub-module that is used to send and track a heartbeat message. The heartbeat message establishes whether a node is alive or unavailable. If the nodes in the federation do not hear a heartbeat message from a node for a certain time, it presumes the node is unreachable and sets the node in the heartbeat tracker to "Unreachable". Once *n* heartbeats are missed from a specific node, the nodes initiate automatic replication.

3.2.3 Group Message Handler All Hydra nodes join a "group" via SVS and synchronize the global view by publishing group messages. Group messages posted by a node are visible to all nodes. This module handles Group Messages - messages that are sent to the group of Hydra nodes. The group message handler interacts with SVS to receive and apply the new updates to Global view and publish new group messages.

3.2.4 Global View Monitor The global view monitor monitors the global view database for any changes. If a change is detected, it initiates an outgoing update message. It also applies update messages received by the node to the Global view database.

3.2.5 User-Facing Modules These public-facing modules allow users to interact with the Hydra system, providing data insertion, deletion, and retrieval functionality. They also allow the user/applications to query the system for file availability.

3.2.6 Network Operation Center (NOC) NOC is a centralized entity that acts at the root of trust for a deployment. NOC provides signed certificates to the nodes and users.

3.2.7 Logging Module This module logs all node records (data operations and errors) to monitor Hydra and provide information to the administrator.

3.2.8 Local File Storage Local file storage is the Database on a node that holds the Data packets corresponding to a published file. The file storage module also performs garbage collection and deletes any file not accessed for a certain duration.

4 Hydra Design Specifications

This section outlines the design specifications of Hydra.

4.1 Naming

Names in Hydra are important as they are the primary construct of the system.

Hydra uses several types of names:

- The Hydra Prefix: decided by the operator(s). Example: "/Hydra"
- File (content) Names: decided by data publisher(s). Example: "/human/genome/dna/hg38"
- Node Names: decided by the operator(s). Example: "/hydra/node-X1"
- The NOC Prefix: decided by operator(s). Example: "/hydra/NOC".

4.1.1 Hydra Prefix The <hydra-prefix> acts as the primary namespace for Hydra operations. An example Hydra prefix can be "/hydra" or "/genomics". This hydra prefix utilizes SVS to actually distribute messages sent on this prefix to all nodes in the federation.

Below are a few more specific uses of sub-namespaces under the Hydra prefix.

- "/<hydra-prefix>/<function>/<function-info>" is utilized to conduct functions that are user-facing. Users use this prefix for inserting, deleting, and any other interaction that affects data directly within the network. An example might be "/hydra/insert/<filename>".
- "/<hydra-prefix>/group" is the prefix for group communications. This prefix is utilized for sending messages through State Vector Sync between nodes. The group prefix is under the Hydra prefix since it is used to notify all the nodes of changes in the system.

4.1.2 Node prefixes The node prefix is utilized for unicast operations which allow Interests to be sent between a specific node and a specific target node within Hydra. The main

use case of the node prefix is to enable communication between Hydra nodes. The node prefix is also used for directing a user request to a specific node if the contacted node does not have the content. It is also used for replication. The node prefix is in the form of "/<node-name>/<hydra-prefix>/fetch/...". This can be appended by operation or content specific names.

4.1.3 ForwardingHints When data is not present in the node that a user contacted, it returns the name of the node that holds the data. This forwarding hint allows the user to express a subsequent Interest that is steered to a specific node name. One example might be "/human/genome/dna/hg38" with a forwarding hint to "hydra/node-X1".

4.1.4 Name Prefixes for NOC NOC is considered the trust anchor of Hydra. While it is not part of the Hydra federation, it uses NDN names for receiving requests. The name prefixes for NOC would be "/hydra/NOC/...".

4.1.5 Node Naming Nodes names are required to be unique. Hydra does not add any extra components to a node name when using it for communication. Example names can be "'/TnTech/CSC/hydra'" or "'/hydra/node-xx'", or "'/normlabs/hydra-5'".

4.1.6 Certificate names Hydra uses a hierarchy of certificate names to provide trust.

An example Hydra root key (hydra root = hydra trust anchor) would be "/hydra/KEY/...".

Node cert names can be "/hydra/32=nodes/host.ucla/KEY/... (key for UCLA node)" or "/hydra/32=nodes/fabric-hostXYZ/KEY/... (key for hostXYZ)".

The user cert names can be "/hydra/32=users/XX@gmail.com/32=ns/FIU/experiments/KEY/..." or "/hydra/32=users/YY@gmail.com/32=ns/FIU/experiments/2022/KEY/..."

4.1.7 Content naming There are two ways we can name files in Hydra. First, we can utilize the content names created by the publisher. Example of such a name would be "/human/genome/dna/hg38". The other option is to utilize Hydra specific names such as "/hydra/human/genome/dna/hg38". While both are acceptable, there are a few trade-offs. With the first option, we assume the owner of the name prefix (i.e., "/human/genome" will allow Hydra to announce the prefix. Second, if publishers publish content using different name prefixes, a Hydra node must announce all those name prefixes into the routing system (e.g. "/human/genome" and "/kidney"). Finally, creating an easy-to-utilize trust schema might be difficult with this model. On the other hand, appending the "/Hydra" prefix to each name makes trust relations and routing announcements simpler. A node can simply announce the "/Hydra" prefix into the routing system. However, this requires changing the content names (e.g./ "/human/genome/dna/hg38" becomes "/hydra/human/genome/dna/hg38", which makes the content specific to the Hydra framework. For our implementation, we use the first naming model.

4.2 Security

A Network Operation Center (NOC) acts as the system trust anchor for Hydra deployments, and issues security certificates. A new Hydra node fetches a certificate by sending an Interest. Next the node verifies the returned self-signed certificate out-of-band. Then the node signs the Interest and requests a certificate from the NOC. Finally, NOC signs the certificate using the trust anchor and returns it in the replied data.

In order to authenticate the Hydra group communications and interactions with users, Hydra's security model includes three roles in the system.

- Network Operator Center (NOC): serving as the system trust anchor for each Hydra deployment, adding or deleting Hydra nodes from the federation, and managing the security policies for the Hydra system.
- Hydra Node: a server that joins the federated repo system. One trusted server might start a new process if the old one terminates.
- Client: users who use the Hydra system by sending Insertion and Deletion commands or data requests.

In Hydra's security design, we assume the NOC is trusted, and each Hydra node is also trusted once added. To bootstrap one Hydra node to the system, out-of-band verification (e.g., pre-shared passcode, email verification) is needed to authenticate the initial communication between the Hydra node and NOC. A new Hydra node fetches the trust anchor by sending Interest /hydra/bootstrap/anchor, and verifying the returned self-signed certificate out-of-band. Then it uses the out-of-band pre-shared material to sign Interest /hydra/bootstrap/cert and request a certificate from the NOC. Verifying the certificate requester's authenticity, the NOC uses the trust anchor to sign a certificate and returns it in the replied Data. The new Hydra node learns its node name from the certificate via certificate installation. After installing the trust anchor and node certificate, the Hydra node keeps its security policies up-to-date by periodically send Interest /hydra/bootstrap/schema and fetch new policies.

The Hydra group communication should also be protected from invalid message senders. Therefore, Hydra requires each

node to sign the Interest and Data in group communication and enforce the system membership checking through verifying the signer's certificate with the trust anchor.
Besides that, users' commands to the Hydra system should also be authenticated. Different from authenticating the group communication among Hydra nodes, Hydra users do not share the trust anchor with the Hydra nodes. The identity of users can only be verified along the certificate chain when the Hydra nodes have users' trust anchors. Hydra achieves this via security policy distribution, where individual Hydra nodes fetch the trusted anchors and corresponding signing rules under those namespaces.

4.3 Files

Hydra uses the term *file* to describe the data unit of insertion, deletion, retrieval, or replication. However, a file is not necessarily a file in the UNIX system; it is just a BLOB of data that is identified by a unique name. See Section 4.1 for more details on file names.

A file consists of NDN Data packet(s); however, the size of a file directly impacts the performance of the following operations: replication, failure recovery, insertion, and retrieval. These NDN Data packet(s) also determine the unit size of a particular file.

4.4 Favor

Due to Hydra being a federation of storage nodes with different storage capacities and policies, it is necessary to have a mechanism that can express a node's local conditions and preferences for storing or replicating additional files.

Favor can be thought of as "how suitable a node is to carry file(s)". Every node is responsible for calculating a numeric value for node X where X is each node within the system including itself. The range of this numeric value is determined by the formula used. As implicitly indicated, favor is an entirely local calculation. However, all parameters of the calculation are announced by every node giving each node complete control over what and which files are stored in their respective storage. Essentially, favor protects node independence within the federated system.

As it stands, favor is calculated per node and all parameters are included in every GM in order to be as up to date as possible. To limit network traffic, Hydra does not send a GM when these parameters change. See Section 4.6 for more details on GMs.

Favor calculations may incorporate the following aspects:

- storage capacity (current usage and max usage)
- network stability / traffic
- location
- prefix preferences, file origin, node's labels, etc
- local policy

Favor affects Hydra's overall performance, usability, and stability. For our initial implementation, we simplified favor's calculation to be based only on storage capacity.

In the future, additional parameters and functions of favor may be necessary for different applications. For example, a baseline favor value may be necessary for high-volume storage environments to reserve some capacity for file takeovers essentially protecting Hydra's resiliency. See Section **??** for more details on future work.

4.5 Storage

A Hydra node utilizes several databases for maintaining state and for storage of different types of data.

Each of these databases is described briefly below.

- SVS Database: A database of all published messages over the Sync group, implemented as a key-value store.
- Global View: A Hydra node's state – a relational database containing all information relating to the entire system.
- Local File Storage: A key-value database holding files that the Hydra node is storing.
- Local Reserved Storage: A sectioned off part of Local File Storage to only be used to help with special operations such as data ingestion. This space is not used for storing replicated files or Hydra's metadata.
- Command Table: A table holding the progress and status of the commands being currently executed.
- Logs: These may either be stored locally or inside a distributed logging system or both.
- Certificate and Key Database: A key-value database that holds key and certificate information required for secured publication of messages, data validation, and user authentication.

Currently, we do not impose any limit on these storages. However, operational deployments will need to define the storage limit for these databases. file insertions, deletions or failures to all other nodes. Hydra uses the State Vector Sync (SVS) protocol[3] to achieve efficient and loss resilient global view synchronization. SVS is further discussed in Subsection **??**.

While GMs help exchange states, we still need a way to transfer and synchronize GMs between hosts. For this purpose, Hydra utilizes the State Vector Sync (SVS) protocol. Note that this sync protocol only provides eventual consistency guarantees. Note that SVS only provides a synchronization capability for message exchange. Hydra is able to utilize any other synchronization protocol for message exchange.

4.6 Group Messages

Hydra utilizes Group Messages (GMs) to exchange updates among nodes. A published Group Message goes out to every node within Hydra such that every node receives every Group Message. Understanding all past actions in Hydra allows the nodes to act in a more precise and knowledgeable manner.

There are six Group Message types with each performing different actions.

- Insert: a GM containing metadata of a new file that is inserted in Hydra.
- Delete: a GM containing information about a file to delete from Hydra.
- Claim: a GM stating the node is going to fetch or is unable to fetch a certain file.
- Store: a GM stating a node has stored a file in its local storage.
- Heartbeat: a GM that is periodically published by nodes to let others know that it is alive. In our current implementation, the interval of heartbeats is set to 30 seconds.
- Leave: a GM send by a leaving node in order to help speed up replication

To keep network overhead low, we treat all messages as heartbeat messages. If there is no GM withing a specified time, a specific heartbeat message is announced.

4.7 Global View

The Global View can be thought of as a node's view of the entire system.

As nodes exchange messages (Group Messages, see next section), a Hydra node creates and stores the state of the system in a light-weight local database known as the 'Global View'.

Note that this Global View is not stored once in a global storage accessible by every node. Instead, each node has its own version of this global view that they maintain. Throughout the duration of operation, a Hydra node synchronizes this database with the other nodes through continuously published group messages.

Every node in Hydra has a global view of:

- All node information
- Each file's specifics
  - which nodes are in possession of the file (the "on list")
  - which nodes can step up to take over the file (the "backup list")
  - meta-info (size, origin node, copies, and etc)

The Global View also has the 'state vector' that made up the Global View. This state vector is the same type found in SVS and indicates the sequence of messages that were incorporated into the state.

Global View layout example (capital alphabetical letters are placements for node names):

```
{
"state_vector": {"A":123,"B":120,"C":100,"D":150,"E":123},

"nodes": [
    {"name": "D", "favor": 50, “alive”: True},
    {"name": "C", "favor": 20, “alive”: True},
    {"name": "B", "favor": 15, “alive”: True},
    {"name": "A", "favor": 10, “alive”: True},
    ...
    ],

"files":[
    {
        "name": "/genomics/fileA",
        "size": 128,
        "contact": "B",
        "copies": 3,
        "on": ["B","A","D"],
        "on_history": ["C"]
    },
    ...
    ]
}
```

4.8 PubSub Protocol

Hydra's interface for data insertion and data deletion takes inspiration from the previous incarnation of standalone NDN repo[]. A PubSub-like protocol is used for exchanging messages with clients such as file insertion commands and updates. These operations are further outlined in Section 7.

4.9 Status Codes

Users can receive multiple status codes (like a server error + redirection for instance), the following is an overview of all status codes in Hydra.

5 Hydra User Operations

5.1 User Bootstrapping

In order to bootstrap a Hydra user into the system, each Hydra user needs to obtain trust anchor, trust policies of Hydra and and get certified by the Hydra NOC

In Hydra, each user obtains the Hydra trust anchor and initial trust policies out-of-band. After that, Hydra NOC authenticates each user on their email addresses. This requires Hydra NOC knowing trustworthy email addresses via initial out-of-band trust relations. We rely on the human trust relations between users and the NOC operator to realize the user authentication. The NOC operator maintains a list

Table 1: Hydra Command Status Codes

Status	Code
Informational	
STAND_BY	100
FETCHING	101
Success	
OK	200
Redirection	
Client Error	
BAD_NAME	400
BAD_REQUEST	401
UNAUTHENTICATED	402
UNAUTHORIZED	403
NOT_FOUND	404
NO_COMMAND	405
Server Error	
RESOURCE_LIMIT	500
TRAFFIC_OVERLOAD	501
NODE_DISCONNECT	502
UNKNOWN_ERROR	503

of trustworthy email addresses and configures the it to the NOC application.

Hydra user utilizes NDNCERT [4] to perform the email authentication and obtains certificate from NOC. Specifically, Hydra NOC as the NDNCERT CA does the following steps upon receiving a certificate request from a Hydra user. It first checks the Hydra user email address membership, then verifies the email address by sending a PIN and requesting sending back in NDNCERT message. Upon successful email membership and possession verification, Hydra NOC uses the user naming convention Tianyuan ▶*where's the user naming convention? Susmit - also what do we use this name for?*◀ to assign the user an NDN name based on its email and certifies it.

As the final step, a Hydra user completes its security bootstrapping by obtaining and installing the issued certificate.

6 Hydra Node Operations

Aside from handling incoming data operations, a Hydra node has other responsibilities partly due to being in a distributed, federated system and partly due to managing its own resources.

6.1 Node Security Bootstrapping

Similar to the user bootstrapping (Section 5.1), each Hydra node need to obtain its trust anchor, trust policies and certificate.

In Hydra, each node obtains the trust anchor and initial trust policies out-of-band. After that, Hydra NOC authenticates and authorizes each node on their node names and public keys. This requires Hydra NOC knowing trustworthy <node name, public keys> bindings via initial out-of-band trust relations (e.g., ssh, email).

For simplicity, Hydra uses email to establish the initial trust relations. Before establishing a new node, the node operator emails the <node name, public key> binding to the NOC operator who configures the NOC application to authenticate the received binding.

Hydra node utilizes NDNCERT to perform the public key authentication and obtains certificate from NOC. The Hydra NOC first verifies if the hydra node has the membership in the system by checking its public key in the pre-configured trusted bindings, and it asks the node to perform the Proof-of-Possession Challenge where the node uses private key to sign a nonce to prove its public key possession. Upon successful signature verification, Hydra NOC issues the new node with a certificate with the node name and public key obtained from trusted name-key bindings.

A Hydra node completes its security bootstrapping by obtaining and installing the issued certificate.

6.2 Federated Node Responsibilities

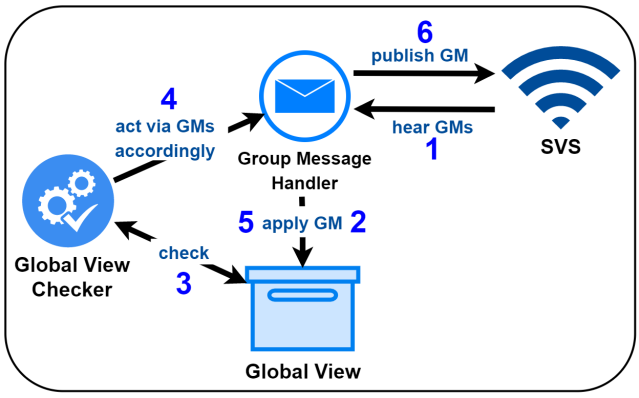

Figure 4: Modules Involved with Checking the Global View

One of the major responsibilities that a node actively fulfills is checking if any federated actions are required to be taken. This is done by the Global View Checker. The Global View Checker consistently checks the Global View as seen in figure 4.

6.3 Storage Management

Each node storage space contains two components: Local File Storage and Local Reserved Storage. Local File Storage

provides sufficient space to support valid replication requests. The function of Local Reserved Storage is to reserve computational resources for system processes, this space is not used to store replication files or metadata for Hydra.

6.4 Joining the federation and membership management

To join the federation a node must install hydra and begin broadcasting a heartbeat. Once the broadcasted heartbeat has been detected between the new node and one of the pre-existing nodes within the federation. The pre-existing node will update the global view for the federation to acknowledge the new node. At the same time, the new node is still broadcasting its heartbeat to other pre-existing nodes within the federation, and they are following suit with the previously discussed pre-existing node's actions. This will increase the propagation of the new node's acknowledgment within the global view. Once this is complete it will be listed as a node within the network or federation, and the others can begin to interact with the new node. The NOC serves as the centralized node management system within the hydra federation. Node management is inherently done through limiting the behavior of valid node operations to the operations discussed within this section, providing pooled storage, and updates through heartbeat and global view messages to the wider federation. If these node requirements are found to be insufficient for any given node then it is determined by the NOC, which manages adding and deleting nodes from the federation. The root admin with access to the NOC is responsible for the final approval and deletion of a node into the federation.

6.5 Nodes becoming unresponsive

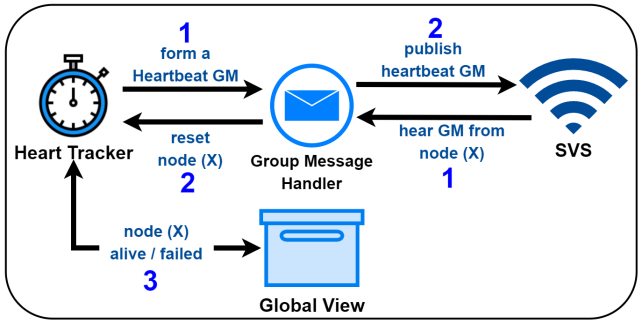

Figure 5: Modules Involved with Tracking and Maintaining Node States

A node becomes unresponsive for the following reasons:

- To leave the system a node must remove the hydra infrastructure or shut down. It will become unresponsive and be dropped from the system over time providing the same availability to rejoin that is discussed in 6.5.
- An incident occurs that limits communication between two or more nodes, or completely shut down the node.

To label a node as unresponsive, a total of three failed heartbeats must be noticed by another nodes heartbeat tracker. The Heartbeats system including the tracker is shown in figure 5. All nodes are responsible for their own detection of unresponsive nodes. Once an unresponsive node has been detected, the investigating node will move the unresponsive node to a suspended record of nodes that have gone offline. This status allows the node to update its record for other operations such as managing heartbeats and file replication. No message through State Vector Sync (SVS) or the Global View updating the status of the unresponsive node is sent out over the network by the investigating node.

This is done for two reasons:

- First, it allows unresponsive nodes during some disruptions to continue their communication with the rest of the network in the case of a partition. It is also assumed that all other nodes will recognize that a node is unresponsive too if it had truly gone offline. Once the disruption is over the unresponsive node will follow the reestablishment process with the investigating node(s) that have labeled it unresponsive.
- Second, if a node has completely gone offline; it can be assumed all other nodes will soon detect this because the offline node will no longer send heartbeats.

These two simple assumptions for handling unresponsive nodes cover all situations that could be reached within the system and allow for more accurate detection of disruptions within the system by handling them locally. This is done to avoid situations where node A may communicate normally with the rest of the network except for node B. If we did not handle unresponsiveness locally in this case, node B may alert the rest of the network that node A is unresponsive, cutting it off from communicating with the entire network due only to a minor communication problem between node A and B.

6.6 Nodes leaving the system

Nodes at anytime may leave the system as they wish and must simply state it as any other update To leave the system, a node can publish the leave GM or become unresponsive as previously discussed in section 6.5. The leave GM allows the rest of the system to begin the replication procedure outlined within section 6.5. This simplifies determining what data needs to be replicated as it can be determined from the

leaving node itself, rather the the federation of nodes determining which files. Once the leaving node has determined which files need to be replicated it will publish the leave GM containing this list for the other nodes. Once a leave GM is published the other federation nodes can replicate this list of files received. However, a leaving node is not required to stay for this process, since the federation can recover from it becoming unresponsive. A leaving node deciding to remain throughout the leaving procedure does provide performance benefits to replication as replicas can be obtained from the leaving node.The leaving node will begin replicating files to them based on the favor found within all GMs.

In this formal leaving process it is not necessary for the other nodes to need to determine what files to replicate. They will allow the leaving node to determine this has it already has the knowledge of how many copies of its own files exist, and which need to be replicated. Other federation nodes are on stand by until the leaving node becomes unresponsive or completes the formal leaving process. Once this process is determined complete from store GMs a second leave message will originate from the leaving node to alert the federation of this, and it will complete leaving the system. During this stand by all other federation nodes will move the leaving nodes on list records to the 'on history' records. The leaving node will be listed in the 'on history' for a duration of one month in case the leaving node rejoins the federation. When this month duration expires all records of the left node will be removed from the 'on history'. This allows for temporarily leaving the system but if a node is gone for to long treating it as a new node upon rejoining.

6.7 Reestablishing a Node's State / Node State Reinstatement

For reinstatement of a node the same name of the node's previous state must be used. This is done to re-link with the records other nodes within the network hold for the previous iteration of our current node. Along with this a local recovery file representing the state in which said node was at the time it became unresponsive or offline is utilized. This file is used locally to determine any differences that have occurred on the system since it last communicated with the Hydra network, and then communicate these differences to the wider network. The purpose of using the differences in system states is that it is more likely by only correcting what is different few corrections will be needed overall, this assumption does dwindle overtime. This process is to allow the node to reclaim its previous status within the network while also updating any nodes that need to know about changes. For example, if a data set was removed during the lapse in communication then other nodes expecting our reinstated node to contain that data need to be alerted to the fact it no longer contains that data. This allows for quick and accurate recovery of a node even if files were deleted or inserted from its system during the lapse in communication.

7 Hydra Data Operations

To demonstrate how Hydra performs data operations, we will utilize an initial deployment on FABRIC. We assume all these operations happen after the data publisher is authenticated by the Hydra node.

Imagine the scenario where Clemson's Genomics and Bioinformatics facility decides to use Hydra to store pre-processed (i.e. indexed) genomes. The main goal of this exercise is to allow data access for the researchers and their collaborators in a reliable manner.

A user can download the correct set of files and insert into downstream analytic workflows simply by asking for these datasets by name.

7.1 Data Insertion

7.1.1 Scenario Alice, a graduate student who is doing research at the Clemson's Genomics and Bioinformatics facility, has produced a pre-processed (i.e. indexed) axolotl genome that she believes is critical in understanding an axolotl's ability to regenerate tissue. She desires to publish this data in the form of a File within Hydra. To satisfy this scenario, she performs the following interactions with Hydra.

7.1.2 User-to-Node Interaction The process of how a user will interact with a Hydra node via NDN is described in Figure 6. Any Hydra node can be contacted by the user for data insertion, and this interaction can be summarized as a PubSub-like interaction: it uses a notification Interest with a component /notify and data retrieval via /msg. This interaction between user (A) and node (X) goes as follows:

(1) User (A) will send an Interest notification using the prefix "/<hydra-prefix/insert", announcing that it has a command for any Hydra node to process.
(2) Upon hearing this, node (X) will send an Interest to fetch this command using user (A)'s prefix (stated in user (A)'s FirstContact structure found in the Interest notification's app parameters).
(3) Node (X) will begin to process this command; and because the command is an insertion, node (X) will begin to fetch file (F) using user (A)'s fetch path (found in the InsertCommand structure).
(4) Once file (F) is retrieved, node (X) will send a notification Interest stating that the command that user (a) sent has a updated status.
(5) User (A) can then fetch the status of the command using node (X)'s prefix (stated in node (X)'s FirstContact structure found in a previous interest).

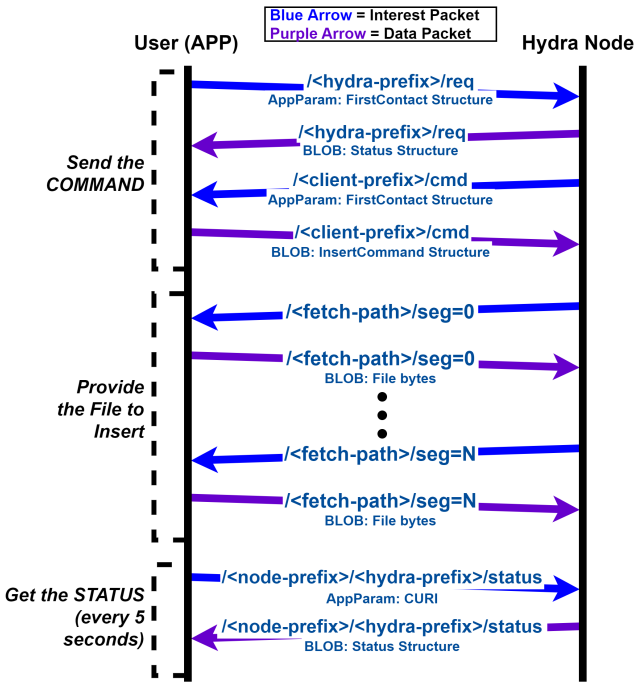

Figure 6: NDN Interaction of A User and A Hydra Node During Data Insertion

There are a few key points throughout this process.

(1) User (A) can fetch the status at any time as it has the necessary info; however if the command is not ready, node (X) will tell user (A) to wait.
(2) The switch from anycast to unicast is necessary to ensure a proper response as command information is not directly shared between Hydra nodes.
(3) Throughout the interaction, a command unique resource identifier (curi) is used. This allows Hydra nodes to process more than one command from a user and allows users to send more than one command.

7.1.3 Module Interaction The User-to-Node interaction leads to several interactions within Hydra.Figure 7 shows these interactions. When node (X) receives an Insertion command for file (F) from user (A):

(1) Node (X) properly authenticates the command, checks to see if the command can be executed, and then immediately starts fetching file (F) if the command satisfies all requirements.
(2) After storing file (F), node (X) updates the status for user (A) allowing user (A) to go offline.
(3) Node (X) then forms an Insert GM which includes all file metadata such as file name, size, etc. and states that node (X) will be storing this file.
(4) Node (X) applies this GM to its Global View.

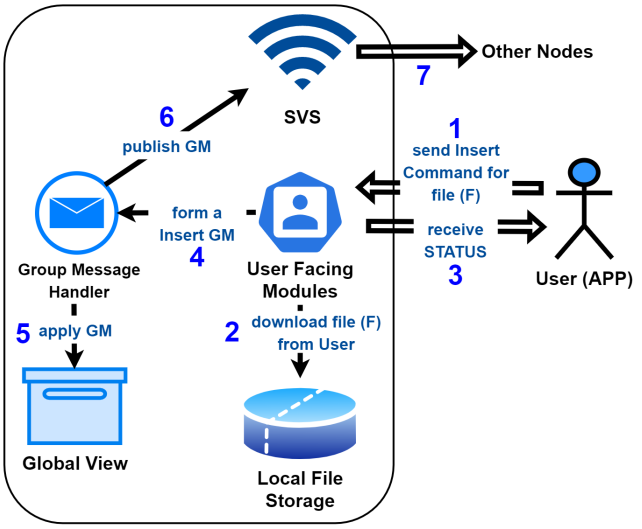

Figure 7: Module Interaction to Fulfill a User's Insertion Command

(5) Node (X) publishes this GM using SVS, our distributed synchronization protocol.

Every node will receive the Insert GM. When node (Y) receives the Insert GM sent by node (X) containing file (F)'s info, it performs the following operations:

(1) Node (Y) applies the GM to its Global View: file (F) information is added.
(2) Node (Y) sees that there is a replication need as file (F) does not meet the replication degree of 3 (stated in Hydra's base policy). For how replication is done, see Subsection 7.2

7.1.4 Data Structure Formats There are several structures that are used for the entire data insertion process. For the group message structures used in Module interaction, please refer back to Section 4.6.

The structures used in User-to-Node interaction include the following:

(1) FirstContact: This includes the preferred name prefix that the sender wants to use for further interaction and a sender command unique resource identifier (curi) that the sender will use to refer to this interaction.
(2) InsertCommand: This include all necessary info that a Hydra node needs for a publication which includes a fetch path, file name, and other metainfo about the file.
(3) NotificationSpecification: This simply includes a command unique resource identifier (curi) of the receiver that the sender is referring to.
(4) CommandStatus: This just contains a Status code that can give simple feedback to the user. Theses codes are

very simple and based on HTTP response status codes, please see Table 1.

7.2 Data Replication

7.2.1 Scenario The FABRIC platform that the Hydra instance is running on is seeing a dramatic increase in popularity. As such, new sites have been installed and more users have been added. With tens of sites and users, site failures can happen. Hydra needs to have a fast replication mechanism in order to quickly recover from Hydra node failures. Hydra should be able to copy Alice's file that she inserted in Subsection 7.1 several times (3 is Hydra's base policy) to prevent permanent data loss. To satisfy these scenarios, the following interactions are conducted.

7.2.2 Module Interaction A replication need is noticed by the Global View Checker (see Figure 4 for more details).

This need is a result of one of the following:

(1) An Insert GM is heard.
(2) A Node has become unresponsive.

When node (X) sees replication needs, it creates a list of the files that:

(1) Do not meet the necessary degree of replication counting nodes that are currently fetching the file and not counting unresponsive nodes
(2) Calculates Favor of each node based on the parameters it received via sync messages.
(3) If it is among the highest favor at this time, it starts the replication.

After deciding on replication, the node does the following:

(1) Node (x) decides to replicate the file.
(2) Node (X) starts to fetch the selected files by sending interests following the format /<node-name>/<hydra-prefix>/fetch/<file-name>/<segment-no> where the node name is a selected node that has the file already.
(3) Node (X) updates its global view and publishes a GM using SVS

7.3 Data Retrieval

7.3.1 Scenario Bob in Dallas has heard people talk about the research Alice from Subsection 7.1 has been doing on the Hydra instance on FABRIC. So much so that Bob wants to see for himself what the data looks like. Of course, Bob (being a new user of the Hydra instance) has no idea where the data replicas are located. The replicas could not be on Dallas, the Hydra node he is most close to. Therefore, Bob needs a way to fetch Alice's data regardless of where the data is located within Hydra. To satisfy this scenario, the following interactions are conducted.

7.3.2 User-to-Node Interaction The process of how a user fetches a file from Hydra can be described as a series of interests and data packets with the only difference being the segment number component. An important note is that any Hydra node can handle a user's fetch requests regardless of whether the contacted node has the file or not.

The user starts out by sending an interest who's name follows the form "/<file-name>/<segment-no>" with the filename being the name found within Hydra and the segment number starting at 0. The user will automatically assume the file spans multiple packets and always add the segment component to its interest. If the file only spans 1 packet, the user will find that out via the final block id within the first data packet. While fetching any a file starts the same, it does not end the same: there are 3 situations that can occur after expressing that interest.

The user's interest for file (F) can be fulfilled from the following:

(1) NACK: File (F) does not exist on Hydra.
(2) BLOB: File (F) exists on the contacted node, data is returned.
(3) ForwardingHint: File (F) exists in Hydra, but not on the contacted node. This acts as a redirect.

To minimize traffic and be more precise with data retrieval, the user can first send an Interest and see how the Interest will be fulfilled which can tell the user how to proceed. After finding out the correct fetch path either by getting data or by a redirect, a user can send Interests to get the rest of the file. If a NACK is received after sending the first Interest, the user knows that the file is not within Hydra and to stop further interactions.

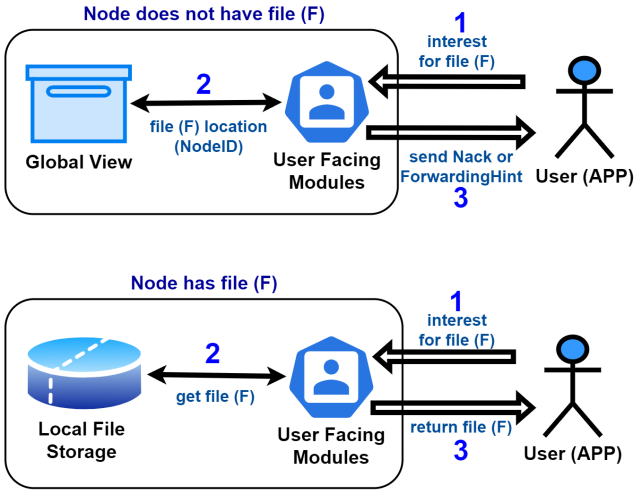

Figure 8: Module Interaction to Fulfill a User's Retrieval Requests

7.3.3 Module Interaction Figure 8 shows two different situations. The Global View is used to indicate where the requested file is. After finding this information, the Hydra node responds to the user in the already-list 3 ways. In the case that the contacted node has the file, the local file storage is used to provide the file data. If a ForwardingHint is required, the node selects a random Hydra node that has the file and provides a forwarding hint (using the selected node's name) following the format "/<hydra-prefix>/<node-name>" for the user to use to fetch the file. Please note that this is the same format that a node uses to fetch a file from another node.

7.4 Data Querying

7.4.1 Scenario It has been two weeks since Alice from Subsection 7.1 published the processed (i.e. indexed) axolotl genome. In fact, she completely forgot that she did insert this data into the Hydra instance on FABRIC. When Alice received an error for trying to insert new data with the same name, she remembered about her forgotten inserted file. From Alice's perspective, having some way of querying to see what files are in Hydra and what file metadata is under a certain name is extremely useful. To satisfy this scenario, the following interactions are conducted.

7.4.2 User-to-Node Interaction The process of how a user sends a query to a Hydra node can be described as a single Interest and a corresponding data packet. Any Hydra node can handle a query sent by the user. The naming of a query interest follows the format "/<hydra-prefix>/query/<query-type>".

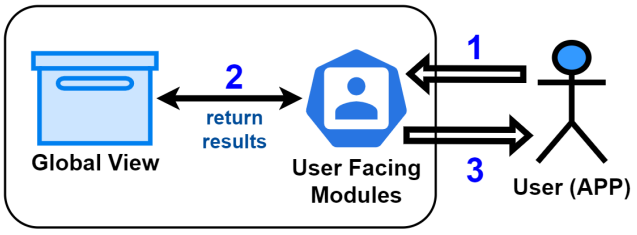

Figure 9: Module Interaction to Fulfill a User's Query Request

7.4.3 Module Interaction The Interest by the user gets processed in a simple way which is described by Figure 9. The Hydra node simply checks the Global View for the query type information and returns it via a data packet.

7.4.4 Types There are different query types that are defined. It is important to note that more queries can be added to better suit the environment, but also any queries can be disabled via Hydra's base policy. This gives Hydra instances more flexibility in what they want to expose to the users.

The query types are the following:

(1) /files: This allows users to see what files are within a Hydra instance.
(2) /nodes: This allows operators to see what nodes are part of a Hydra instance.
(3) /prefix/<prefix>: This allows users to search Hydra for files under a certain prefix.
(4) /file/<filename>: This allows users to see information about a certain file.

7.5 Data Deletion

7.5.1 Scenario The same Alice found in Subsection 7.1 finds out that she made a grave mistake in pre-processing (i.e. indexing) the axolotl genome three weeks after she had inserted the data into Hydra. While Alice can wait a few more weeks to let the data naturally remove itself due to Hydra's base policy, she fears that anyone using her data before it gets removed will get false hope and possibly base future research on false data. This is a major concern for Clemson's Genomics and Bioinformatics facility as it can harmfully affect the facility's goal with using Hydra. Therefore, Alice needs a way to delete a file. To satisfy this scenario, the following interactions are conducted.

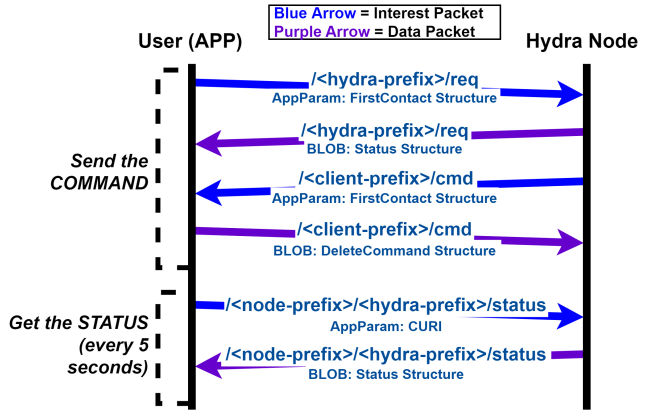

Figure 10: NDN Interaction of a User and a Hydra Node During Deletion

7.5.2 User-to-Node Interaction The process of how a user will interact with a Hydra node via NDN is described in Figure 10 and shares great similarities with data insertion. Any Hydra node can be contacted by the user for data deletion, and this interaction can be summarized as a PubSub-like interaction: it uses a notification interest with a component /notify and a data retrieval interest with a component /msg.

This interaction between user (A) and node (X) goes as follows:

(1) User (A) will send a interest notification using the prefix /<hydra-prefix/delete, announcing that it has a command for any Hydra node to process.
(2) Upon hearing this, node (X) will send an interest to fetch this command using user (A)'s prefix (stated in user (A)'s FirstContact structure found in the interest notification's app parameters).
(3) Node (X) will process this command and than send a notification interest stating that the command that user (A) sent has a updated status.
(4) User (A) can then fetch the status of the command using node (X)'s prefix (stated in node (X)'s FirstContact structure found in a previous interest).

The exact same key points found in Subsection 7.1 for User-to-Node interaction applies for this process as well.

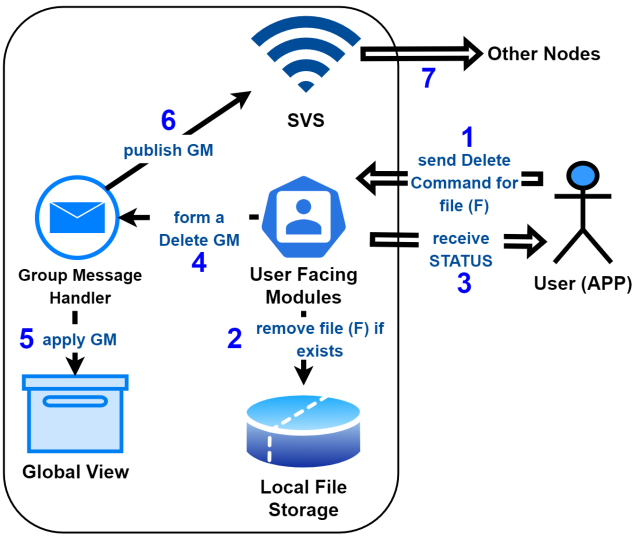

Figure 11: Module Interaction to Fulfill a User's Deletion Command

7.5.3 Module Interaction The User-to-Node interaction leads to Module interaction where a Hydra node will act out the given command. As seen in Figure 11, when node (X) receives a Deletion command for file (F) from user (A):

(1) Node (X) properly authenticates the command.
(2) Node (X) deletes any local copies of file (F).
(3) Node (X) updates the status for user (A) allowing user (A) to go offline.
(4) Node (X) forms a Delete GM which includes the name of file (F).
(5) Node (X) applies this GM to its Global View: all file (F) information is removed.
(6) Node (X) publishes this GM using SVS, our distributed synchronization protocol.

Every node will receive the Delete GM. When node (Y) receives the Delete GM sent by node (X) containing file (F)'s name:

(1) Node (Y) deletes any local copies of file (F).
(2) Node (Y) applies the GM to its Global View: all file (F) information is removed.

7.5.4 Data Structure Formats There are several structures that are used for the entire data deletion process. For the group message structures used in Module interaction, please refer back to Section 4.6. In addition, most of the other structures are the same ones described in Subsection 7.1. The only missing structure is DeleteCommand which holds the filename of the desired-to-be-removed file.

8 Deployment on FABRIC

This section describes how we deployed Hydra on the NSF FABRIC testbed[1].

8.1 Hydra Deployment Overview

As it shown in Figure 12, We provisioned nodes on the NSF FABRIC (https://fabric-testbed.net) testbed as Hydra's initial "soft" deployment. One node is a client node and others are hydra nodes. We choose five nodes because four nodes is the minimum number of nodes to demonstrate the Hydra function and one node as a client node.

Our base FABRIC resource "slice" has the following properties:

- Layer 2 connectivity (i.e. MAC address) to communicate, completely connected network graph, manual setup / routing.
- Easily deployed and destroyed to enable full system rebuilds for reproducibility testing.
- High network bandwidth (100Gbps) links between nationally dispersed sites.
- Scalable resource allocation and modern hardware (e.g. NVME, GPUs, etc.) for the creation of a data lake of sufficient scale for data-intensive scientific workflows.

8.2 Hydra Deployment on the FABRIC Testbed

To deploy the program on Fabric, first we have to become Fabric users by simply signing up at http://portal.fabric-testbed.net. The second step is to join a project. Users can join an existing project or they can create a new one. Fabric has its own JupterHub, which makes users easily integrate with the Fabric environment. The next step is setting up two SSH keypairs: bastion keypaire and sliver keypaire. The bastion keypair can be generated from the Fabric portal but has

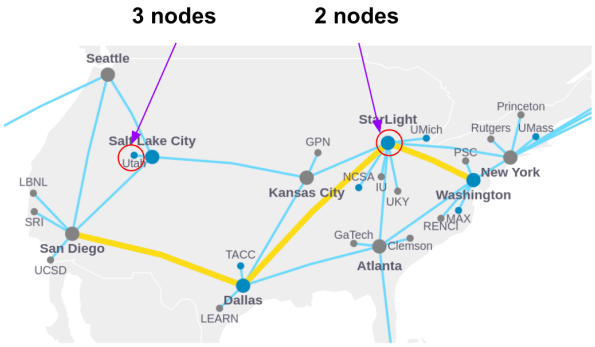

Figure 12: Hydra deployment on the FABRIC testbed.

a finite lifetime. Users need to re-generate their bastion keypairs every 6 months. The sliver keypairs are long lived and installed into Fabric VMs. The environment configuration is completed when users add the path of these two keypairs into the Juyperhub. In our experiment, we used the FABRIC API (https://github.com/fabric-testbed/fabrictestbed-extensions) to request a slice with five nodes with three nodes in Utah and two nodes in Illinois. The five nodes were set up with default components: two CPU cores, 8Gb RAM, and 10Gb SSD disk space. A layer 2 network was built between these five nodes as shown in Figure 13.

After the deployment of the slice with basic VMs, we installed the Hydra packages on each node as python3-pip, libndn-cxx-dev, nfd, ndnping, ndnpeek, ndn-dissect, ndnchunks and ndnsec. Then, we established Hydra python libraries by running a command:

```
node:$ pip3 install python-ndn ndn-storage ndn-
    svs ndn-hydra
```

Next we added the NDN face and route between the client node and the Hydra nodes as in Figure 13. The following command shows how to create face and add route between two hydra nodes (node 2 and node 3) :

```
node2:$ nfd-start
node2:$ nfdc face create remote ether://["MAC
    address of node 3"] local dev://"Ethernet
    interface of node 2 with node 3"
node2:$ nfdc route add /hydra/group ether://["MAC
    address of node 3"]
node2:$ nfdc route add /hydra/node/node3 ether://["
    MAC address of node 3"]
node2:$ nfdc route add /node3 ether://["MAC address
     of node 3"]
```

The following command shows how to create face and add route between a hydra node and a client node (node 2 and node 1) :

```
node2:$ nfd-start
node2:$ nfdc face create remote ether://["MAC
    address of node 1"] local dev://"Ethernet
    interface of node 2 with node 1"
node2:$ nfdc route add /client ether://["MAC
    address of node 1"]
```

Finally we moved data into and out of the Hydra deployment using 1MB, 5MB and 10 MB files for testing purposes.

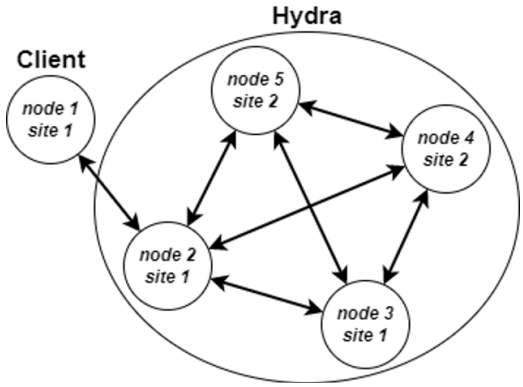

Figure 13: FABRIC topology overview of the client and Hydra nodes.

8.3 Proof of Principle Hydra Experiment

In our proof of principle experiment, we achieved four functionalities: *query*, *insert*, *fetch*, and *delete*. Once all the Hydra nodes came online, we were able to validate internode communication and performance with *iperf*.

First, the client node sent the *query* to the Hydra node by command:

```
node1:$ ndn-hydra-client query -r REPONAME -q QUERY
    [-n NODENAME]
```

This command helps the client check whether the required files are stored on Hydra nodes.

The second test command is the *insert* command:

```
node1:$ ndn-hydra-client insert -r REPONAME -f
    FILENAME -p PATH
```

The client node could then publish data to Hydra nodes. Note that the data file was separated into several small chunks based on the file sizes and stored on Hydra nodes.

Next, we *fetch* the file from the Hydra node by using the command:

```
node1:$ ndn-hydra-client fetch -r REPONAME -f
    FILENAME [-p PATH]
```

Through this command, Hydra linked the client node with the node with the request file and the request file was forwarded to the client node.

file size	client		rep1		rep2	
	avg	std	avg	std	avg	std
10 MB	12.50	2.26	27.28	0.53	19.52	7.41
100 MB	16.86	0.22	135.65	6.52	61.72	61.85
1 GB	11.29	0.01	62.17	60.58	27.26	0.24

Table 2: Hydra insertion speed: We recorded the completion time of the insert command on the client node and the two hydra nodes (rep1 and rep2), and calculated the transfer speed (Mbps). We ran the insert command 3 times for each file size and took the average and standard deviation.

After testing these three commands, the *delete* command was used to remove all files on hydra nodes by using this command:

```
node1:$ ndn-hydra-client delete -r REPONAME -f
    FILENAME
```

This basic procedure is being used to validate Hydra releases as well as vary testbed topology, slice hardware, dataset size, and data name schemas.

8.4 Initial Evaluation

In our proof of concept experiment, we achieved four functionalities: *query*, *insert*, *fetch*, and *delete*. Once all the Hydra nodes came online, we were able to validate inter-node communication and performance with Hydra. Table 2, it shows the transfer speed for inserting a file into Hydra nodes. We used 10 MB files, 100 MB files, and 1 GB files. For each file size, we ran the insert command three times and took the average value and standard deviation. We tested transfer speed on the client node side, on the first hydra node (rep1) storing the inserting file and on the second hydra node (rep2) replicating the inserting file.

Table 3 shows the transfer speed for fetching a file from Hydra nodes. The file sizes are the same as the inserting file sizes. For each file size, we ran the fetch command three times and took the average value and standard deviation. For fetching process, we only want to know how fast the client node receives the request file, so we only consider the time consuming on the client node. For Hydra, we think the same file might serve different users. Therefore, when a client first fetches a file, the file is first fetched from disk to the cache and then from the cache to the client node. Once the first fetch command is finished, the file will stay in the cache. In the future fetch, if users request the same file, the file is fetched from the cache to the client node, which means the transfer speed will be faster than the first fetch.

Table 3: Hydra fetching speed (Mbps): We recorded the completion time of the fetching command on the client node with and without caching.

File Size	No Caching(Mbps)		Caching(Mbps)	
	avg	std	avg	std
10 MB	13.23	0.29	183.92	2.11
100 MB	13.85	0.11	185.82	5.12
1 GB	13.37	0.30	13.51	0.21

However, we set the cache size to 500 MB for these experiments. When the file size is greater than the cache size (e.g. 1 GB), there would not be a considerable difference between cached and uncached retrieval.

9 Conclusions and Future Work

There are several important future work we are working on. First, Favor needs to be explored further. We need to answer questions such as what should be the weights of various parameters, what is the efficiency and overhead of replication based on favor as compared to traditional methods?

Second, we continue to enhance the current data fetching module to reliably handle large files. Third, we are working on improving the replication mechanism to reduce replication latency while maintaining a desired degree of replication. Fourth, reduce the latency of SVS, maintain a more consistent global view to improve system efficiency. Fifth, in order to better maintain storage limits and further improve the performance of the garbage collection process, a better estimate of the time limit for purging expired data is required. We are also working on defining the trust schemas for genomics use cases. Last but not least, we plan to improve Hydra's usability by addressing some of our assumptions.

References

[1] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S. Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[2] Tianxiang Li, Wentao Shang, Alex Afanasyev, Lan Wang, and Lixia Zhang. A brief introduction to ndn dataset synchronization (ndn sync). In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pages 612–618, 2018.

[3] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. Vectorsync: Distributed dataset synchronization over named data networking. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, ICN '17, page 192–193, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Zhiyi Zhang, Alexander Afanasyev, and Lixia Zhang. Ndncert: Universal usable trust management for ndn. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, ICN '17, page 178–179, New York, NY, USA, 2017. Association for Computing Machinery.