HSDS:
A REST Service for HDF5

John Readey
Overview

• Why a HDF Service?
• What’s REST?
• HSDS features
• Architecture
• Security
• Demo

Also…

This talk will focus on the service, but if you missed yesterday’s talk on h5pyd (Python client library for HSDS), it should be available as a video soon.

And tomorrow I’ll be talking about the REST VOL (C client for HSDS)
Introducing HSDS

HSDS – Highly Scalable Data Service -- is a REST-based web service for HDF data

Design criteria:

• Performant – good to great performance
• Scalable – Run across multiple cores and/or clusters
• Feature complete – Support (most) of the features provided by the HDF5 library
• Utilize POSIX or object storage (e.g. AWS S3, Azure Blob Storage)

Note: HSDS was originally developed as a NASA ACCESS 2015 project: https://earthdata.nasa.gov/esds/competitive-programs/access/hsds
HSDS Platforms

HSDS is implemented as a set of containers and can be run on common container management systems:

- docker
- kubernetes
- Amazon EKS
- Azure Kubernetes Service (AKS)
- DC/OS

Using different supported storage systems:

- POSIX Filesystem
HSDS Features

- **HDF5 Feature Support**
  - Groups, Links (including multi-link), Attributes, Datasets, Committed Datatypes
  - Simple and Compound datatypes
  - Hyperslab and Point Selections (also SQL-style queries)
  - Support for compression
    - Standard HDF5 shuffle and deflate filters
    - Support for BLOSC compressors

- **Container based**
  - Run in Docker or Kubernetes or DC/OS

- **Scalable performance:**
  - Can cache recently accessed data in RAM
  - Can parallelize requests across multiple nodes
  - More nodes ➔ better performance
  - Cluster based – any number of machines can be used to constitute the server
  - Multiple clients can read/write to same data source
  - No limit to the amount of data that can be stored by the service
Why an HDF Service?

Before talking about HSDS, let’s ask why a service might be a handy thing to have. Some reasons why this might be of interest…

- Allow remote access to large datasets (the inertia of big data)
- Provide language-neutral interface to HDF
- Enable web-based applications
- Facilitate container-based applications (Docker, Kubernetes, Mesos)
- Explore alternative implementations of HDF – object-storage, asyncio, non-MPI parallelism, etc.
**What is REST?**

- REST is a (loose) standard for creating web-based APIs
- Typically built on top of HTTP
- Uses the 4 most common HTTP operations: GET, POST, PUT, DELETE
- Stateless – one operation doesn’t depend on another
- URI based – every object has a unique identifier
- Language Neutral

**The HDF REST API**

- The HDF REST API is a specification for a web API that enables the HDF data model
- Used by HSDS (and also h5serv – an earlier prototype)
- Other implementations are free to adopt it as well
What makes it RESTful?

- Client-server model
- Stateless – (no client context stored on server)
- Cacheable – clients can cache responses
- Resources identified by URIs (datasets, groups, attributes, etc)
- Standard HTTP methods and behaviors:

<table>
<thead>
<tr>
<th>Method</th>
<th>Safe</th>
<th>Idempotent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Y</td>
<td>Y</td>
<td>Get a description of a resource</td>
</tr>
<tr>
<td>POST</td>
<td>N</td>
<td>N</td>
<td>Create a new resource</td>
</tr>
<tr>
<td>PUT</td>
<td>N</td>
<td>Y</td>
<td>Create a new named resource</td>
</tr>
<tr>
<td>DELETE</td>
<td>N</td>
<td>Y</td>
<td>Delete a resource</td>
</tr>
</tbody>
</table>
### Example URI

<table>
<thead>
<tr>
<th>scheme</th>
<th>domain</th>
<th>port</th>
<th>resource</th>
<th>Query param</th>
</tr>
</thead>
<tbody>
<tr>
<td>http://</td>
<td>kitalabhsds.hdfgroup.org:7253/datasets/34...d5e/value?select=[0:4,0:4]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Scheme:** the connection protocol
- **Endpoint:** DNS name for the server (could be a load balancer)
- **Port:** the port the server is running on
- **Resource:** identifier for the resource (dataset values in this case)
- **Query param:** Modify how the data will be returned
  - (e.g. hyperslab selection)

Request response can either be:
- **JSON** – for metadata
- **Binary** – for dataset reads
HSDS Architecture

Legend:

- Client: Any user of the service
- Load balancer – distributes requests to Service nodes
- Service Nodes – processes requests from clients (with help from Data Nodes)
- Data Nodes – responsible for partition of Object Store
- Object Store: Base storage service (e.g. AWS S3)
HDF Sharded Schema

Why a sharded data format?

• Limit maximum size of any object
• Support parallelism for read/write
• Only data that is modified needs to be updated
• Multiple clients can be reading/updating the same “file”
• Don’t need to manage free space

Legend:
• Dataset is partitioned into chunks
• Each chunk stored as an object (file)
• Dataset meta data (type, shape, attributes, etc.) stored in a separate object (as JSON text)

Big Idea: Map individual HDF5 objects (datasets, groups, chunks) as Object Storage Objects

Each chunk (heavy outlines) get persisted as a separate object
Client-side support

Client Software Stack

C/Fortran Applications  Web Applications

NetCDF4 Lib  Browser

HDF5 Lib  REST VOL

hSpyd  REST Backend

Python Applications  CMD Line Tools

HDF Services

HDF REST API (http)

Note: Clients don't need to know what's going on inside this box!
A word about Python…

*HSDS is implemented in Python which is not thought of as a high performance language. In practice though it’s worked out quite well based on the following factors:*

- HSDS utilizes Python packages (e.g. BLOSC, NumPy) that are wrappers around optimized C (Fortran?) code
- HSDS uses Numba (basically a just-in-time compiler for Python) to speed up critical code blocks
- Heavy use of asyncio (see next two slides) makes efficient use of CPU for IO based workloads
Python async in HSDS

- HSDS relies heavily on Python’s new asyncio module
  - Concurrency based on tasks (rather than say multithreading or multiprocessing)
  - Task switching occurs when process would otherwise wait on I/O

Example:

```python
async def my_func():
    a_regular_function_call()
    await a_blocking_call()
```

- Control will switch to another task when await is encountered
- Result is the app can do other useful work vs. blocking
- Supporting 1000’s of concurrent tasks within a process is quite feasible
Parallelizing data access with asyncio

- SN node invoking parallel requests on DN nodes

```python
tasks = []
for chunk_id in my_chunk_list:
    task = asyncio.ensure_future(read_chunk_query(chunk_id))
    tasks.append(task)
await asyncio.gather(*tasks, loop=loop)
```

- Read_chunk_query makes a http request to a specific DN node
- Set of DN nodes can be reading from S3, decompression and selecting requested data in parallel
- Asyncio.gather waits for all tasks to complete before continuing
- Meanwhile, new requests can be processed by SN node
Security – authentication and authorization

In a web service it’s important to verify who’s who (authentication) and only allow permitted actions (authorization)

- Authentication - HSDS supports several authentication protocols:
  - HTTP Basic Auth
  - Azure Active Directory – (OAuth 2.0)
  - Google OpenID (also Oauth 2.0)
- Authorization – Access Control Lists (ACLs)
  - Per domain list of which users can perform which actions (read, update, delete, etc)
  - Role Base Access Control (RBAC) – enable permission based on user groups
Questions?
Try it out!

Get the software here:

- HSDS: [https://github.com/HDFGroup/hsds](https://github.com/HDFGroup/hsds)
- H5pyd: [https://github.com/HDFGroup/h5pyd](https://github.com/HDFGroup/h5pyd)
- REST VOL: [https://github.com/HDFGroup/vol-rest](https://github.com/HDFGroup/vol-rest)
- REST API documentation: [https://github.com/HDFGroup/hdf-rest-api](https://github.com/HDFGroup/hdf-rest-api)
- Example programs: [https://github.com/HDFGroup/hdflab_examples](https://github.com/HDFGroup/hdflab_examples)