# HSDS New Features:
# AWS Lambda and Direct Access

John Readey

The HDF Group

# Overview



- HSDS Overview
- HSDS Serverless with AWS Lambda
- HSDS Serverless with Direct Access
- Performance Comparison

# HDF as a Service - HSDS

The HDF Group



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

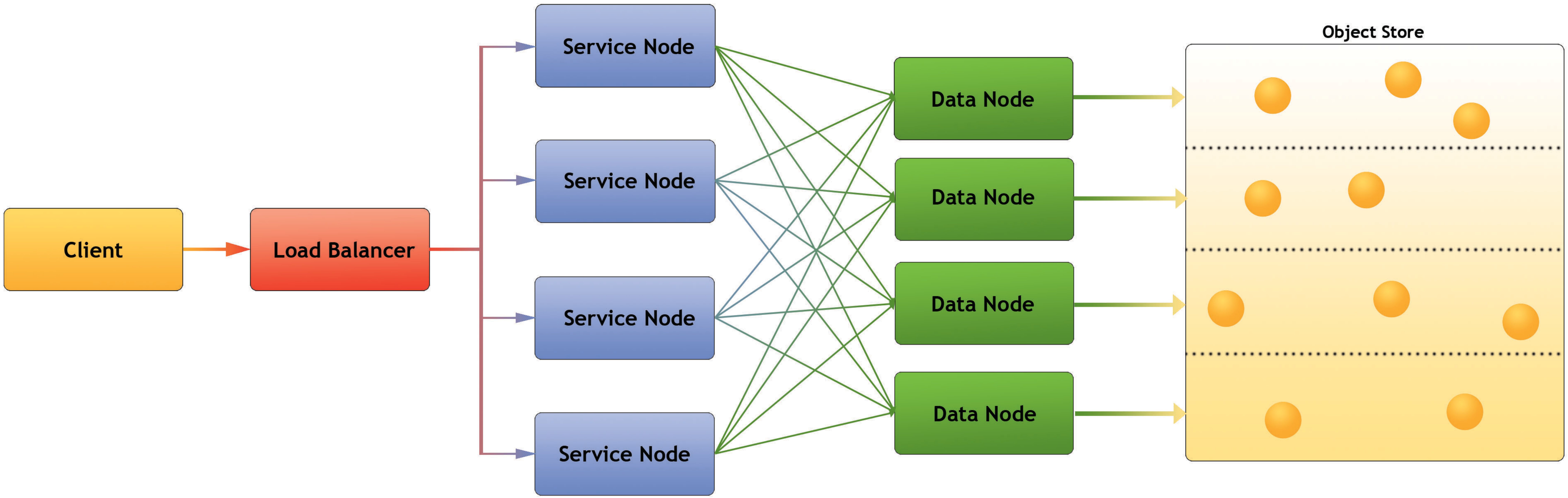
Design criteria:

- 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 Architecture






- 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 or Posix Disk)

# HSDS Platforms

HSDS is implemented as a set of containers and can be run on common container management systems (both cloud & on-prem):

docker

kubernetes

Amazon EKS

Azure Kubernetes Service (AKS)

DC/OS

Using different supported storage systems:

POSIX Filesystem

amazon webservices S3

Microsoft Azure Blob Storage

OpenIO

ceph

# Pros and cons of running a service

The HDF Group



- Accessing a sharded data store via a service (HSDS) is great:
  - Server mediates access to the storage system
  - Server can speed things up by caching recently accessed data
  - Minimizes data I/O between client & server (e.g. remote clients)
  - HSDS running on a large server or cluster can provide more processing capacity and memory than a client might have
  - HSDS serves as a synchronization point for multi-writer/multi-reader algorithms
- Unless it’s not:
  - Don’t want to bother setting up, running service
  - Challenge to scale capacity of service to clients
  - Cloud charges for running server 24/7

# HSDS sans Service



- HSDS  now provide two new (though related) capabilities:
  - AWS Lambda support - HSDS implemented as a Lambda Function
  - Direct Access Model – HSDS implemented entirely on the client
- Both of these enable developers to take advantage of HSDS capabilities without the need to run a server

Example hsinfo cmd:

```
$ hsinfo –e {lambda_dns}
server name: HSDS on AWS Lambda
server state: READY
endpoint: https://vpv89pff5.execute-api.us-west-2.amazonaws.com
username: hslambda
password: ******
home: NO ACCESS
server version: 0.7.0beta
node count: 1
up: 2 sec
h5pyd version: 0.9.0
```

# AWS Lambda



- AWS Lambda is an AWS service that enables function to be executed on demand without the need to provision any infrastructure
- Price (with 1GB memory) is: $0.0000000167/ms
- Beyond potential cost savings, Lambda enables “infinite elasticity”
  - Can support widely varying workloads without need to spin up/down servers
  - By default, 1000 instances of a function can be run simultaneously
- No infrastructure management required
  - i.e. os patches, server backup, monitoring, etc.
- Clients can invoke Lambda directly are utilize AWS API Gateway
  - API Gateway provides the same REST API as HSDS

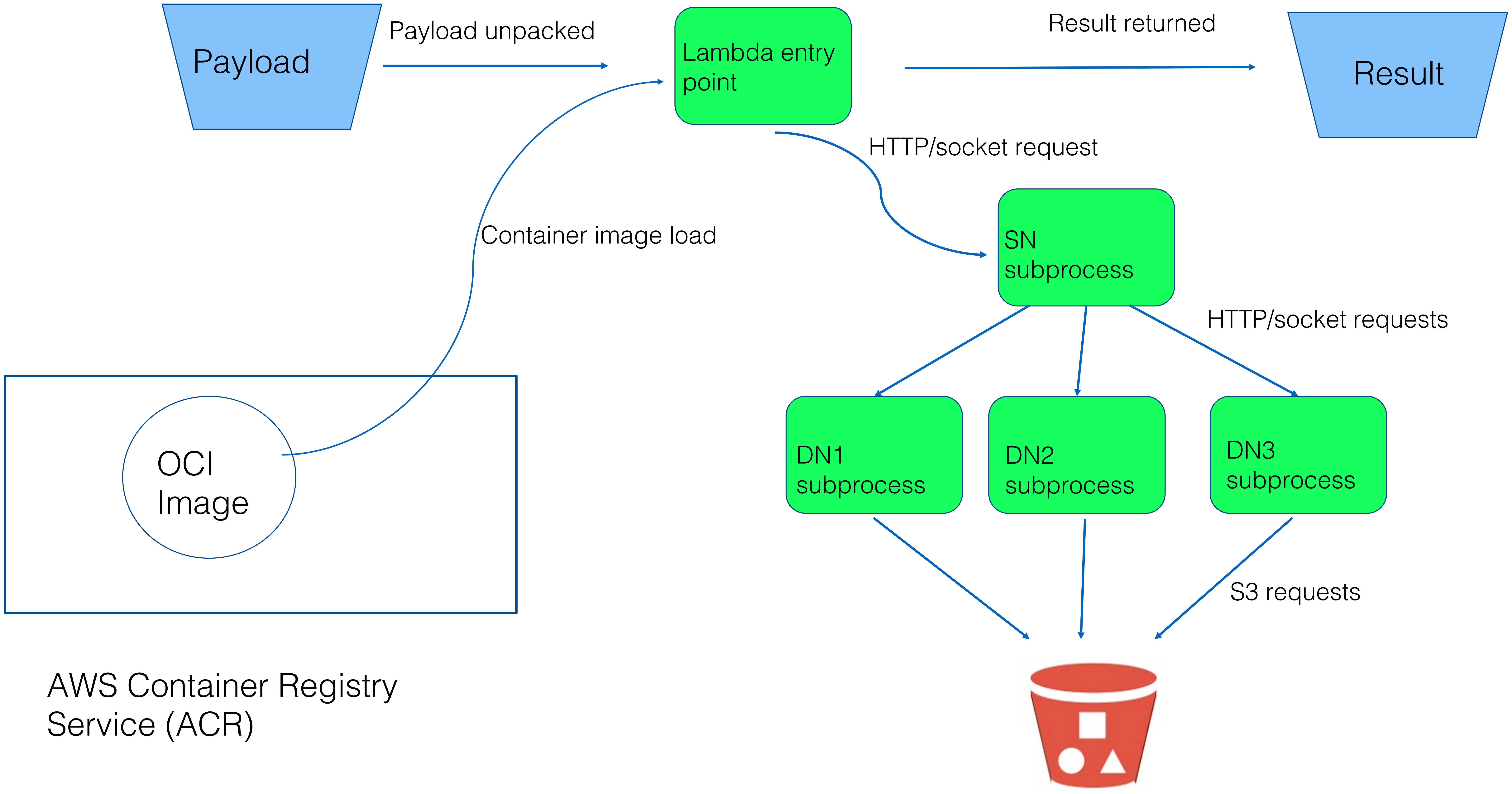
# AWS Lambda Challenges

The HDF Group



- Adapting existing container-based server to run on Lambda is not trivial…
  - Software needs to initialize as quickly as possible (to minimize latency)
  - Unable to take advantage of caching
  - Limited to max 6 VCPUs per function
  - Lambda runtime environment is restricted:
    - No equivalent to docker to manage multiple containers
    - TCP not allowed (which is how HSDS containers talk to each other)
    - Shared memory not allowed

# HSDS with Lambda Architecture




1. Image loaded
2. Subprocesses run
3. Payload unpacked
4. HTTP/socket request sent to SN
5. SN multiplex req to DN nodes
6. DN reads/write to/from S3 bucket
7. DN returns response to SN
8. SN returns response to parent process
9. Result packed to Lambda result
10. Lambda exits

# HSDS Lambda Design



- To minimize the need for special purpose code, HSDS is implemented on Lambda as follows:
  - On startup SN node and DN nodes are run as sub-processes (vs containers)
    - Number of DN nodes based on available VCPUs
    - Nodes communicate via Unix Domain sockets (vs TCP)
  - Payload is unpacked to a HTTP request and sent to the SN node
  - SN node distributes work over DN nodes
  - DN nodes read/write to S3, return result to SN node
  - Response is returned as the Lambda result

# HSDS Lambda Performance Constraints



- Compared with HSDS running on a dedicated server, the response time will be 2-100x slower
- Performance challenges:
  - 2-4 seconds for a ”cold” function to startup
  - ~0.5 seconds for HSDS code to initialize
  - All data must be fetched from S3 (no cache to utilize)
  - Limited number of cores (number of DN nodes) available

# HSDS Direct Access



- HSDS Direct Access enables client code to incorporate HSDS functionality without the need of a server
- As with HSDS Lambda, enables “serverless” operation
- Direct Access is best used for applications that run “close” to the storage and don’t require close multi-client synchronization
- To use, just set endpoint to local. E.g:

```
$ hsinfo --endpoint local
server name: Direct Connect (HSDS)
server state: READY
endpoint: local
username: jreadey@hdfgroup.org
password: ********************************
server version: 0.7.0beta
node count: 6
up: 3 sec
h5pyd version: 0.9.0
```

# HSDS Direct Access Architecture

The HDF Group



As with HSDS Lambda:

- SN code would run in a sub-process
- DN code would run in one or more sub-processes (e.g. based on number of cores)
- Communication between parent processes and sub-processes would be via Unix Domain sockets
- Sub-processes shutdown when last file is closed

But all code executes on local system

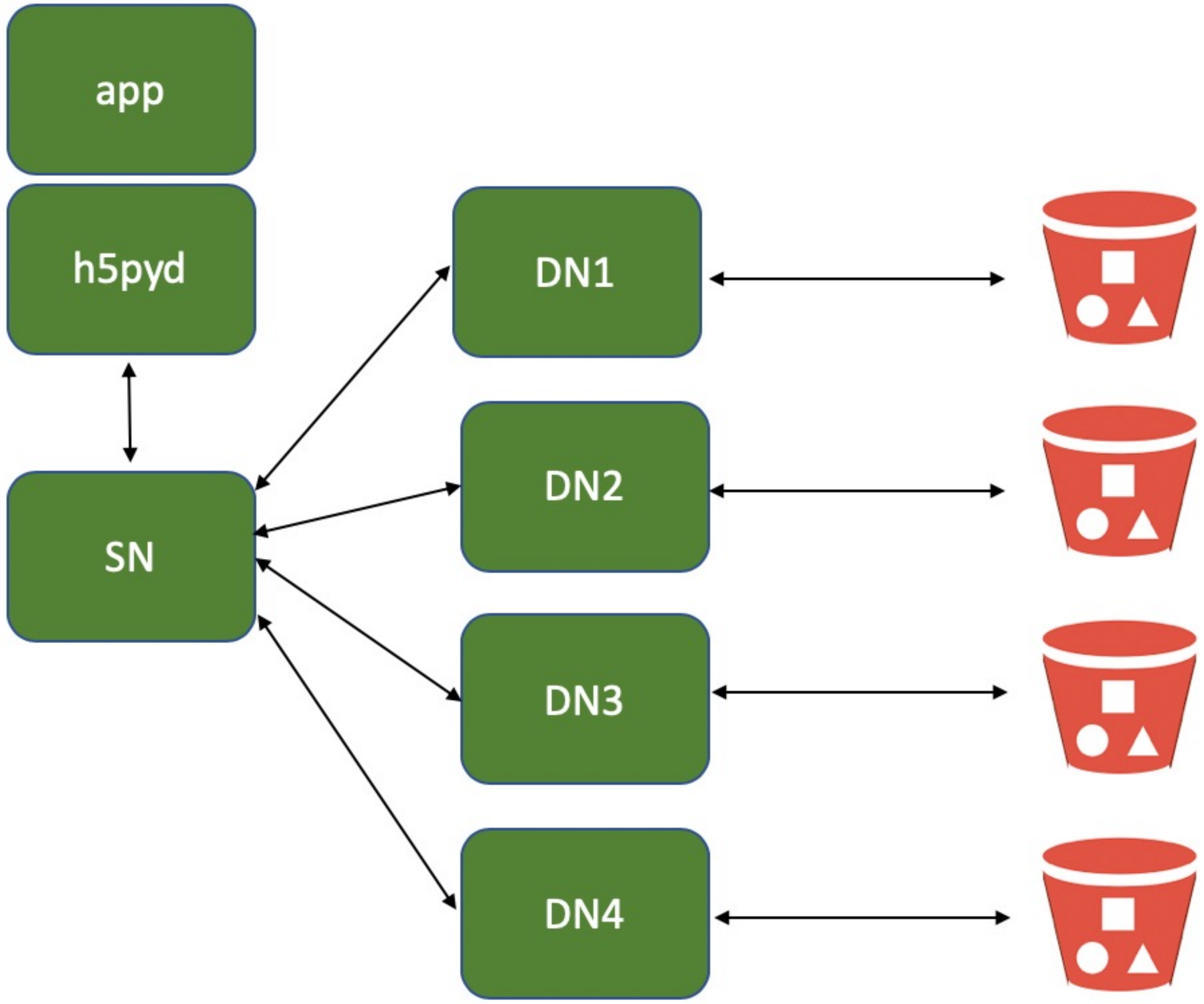
Otherwise, the application will function in same manner as with server

Note: application needs to have authority to access storage system (AWS S3, Azure Blob, Posix Disk, etc.)

# Direct Access Architecture



Number of DN nodes is set to number of cores




All green boxes run as processes on client system

S3, Azure Blob, or Posix storage

# Benchmark Shootout!



- Let’s compare performance among different approaches for a typical task
- The Challenge: read one column from a NREL NSRDB dataset
  - The file: s3://nrel-pds-nsrdb/v3/nsrdb_2000.h5, ~1.5TB
  - The dataset: ”wind_speed” – dimensions: (17568, 2018392), ~66GB
  - Choose column index between 0 and 17567 randomly (to discount any caching effects)
  - The dataset is chunked in such a way that reading one column requires accessing 5425 chunks or ~10GB of data

Disclaimer: Performance depends greatly on how the data is organized, system hardware, application code, phase of the moon, etc. YMMV!

# Contenders

- HDF5 Library reading from Posix Disk
- HDF5 Library w/ ros3 VFD, reading from S3
- HSDS on Docker
- HSDS on Kubernetes with 4 machine cluster
- HSDS with Direct Connect
- HSDS with Lambda

# Hardware



- For Kubernetes:
  - AWS m4.2xlarge – 1 to 16 machines in cluster
    - 32 GB Ram
    - 8 VCPU (VCPU ~= Intel hyperthreading cores)
    - “High” networking
- Everything Else:
  - AWS m5.8xlarge
    - 128 GB Ram
    - 16 VCPU
    - 10 Gb networking
- Both running in same region as S3 Bucket with NREL data

# The code



- Source code for the test is here:
  https://github.com/HDFGroup/hsds/blob/master/tests/perf/nsrdb/nsrdb_test.py
- Usage: python nsrdb_test.py –option
- Where --option is one of:
  - --hdf5: use HDF5 library with Posix File
  - --ros3: Use HDF5 library S3 VFD, S3 file
  - --hsds: Use HSDS
- For HSDS, Direct Access vs. Lambda vs. Docker vs Kubernetes determined by which endpoint is used

# Results



| Contestent | Time (seconds) | Throughput (MB/s) | Notes |
|---|---|---|---|
| HDF5 Lib | 80 | 135 | +2 hour penalty for copying from S3 |
| HDF5 Lib w/ros3 | 328 | 33 | Performance would improve w/ Paged Allocation |
| HSDS Docker 16 node | 16 | 678 | HSDS Config override* |
| HSDS Kubernetes 16 node | 28 | 387 | HSDS Config override* |
| HSDS direct connect | 19 | 571 | Using 16 DN sub-processes |
| HSDS Lambda | DNF | | 500 errors running test |
| | | | |

Conclusions:

- HDF5 Library penalized by having to read each chunk sequentially
- Using HDF5 lib with S3 VFD is slow, but requires no setup (performance improvements coming)
- Direct Connect performance similar to using service
- HSDS Lambda not yet ready for handling large requests

HSDS Config Overrides*:

- max_task_count: 400
- max_chunks_per_request: 6000

These improve performance in situations like this with a few clients and relatively large requests

# Scaling up HSDS

The HDF Group



| HSDS on Docker | Time (seconds) | Throughput (MB/s) | Notes |
|---|---|---|---|
| 1 node | 105 | 103 | |
| 2 nodes | 56 | 193 | |
| 4 nodes | 32 | 339 | |
| 8 nodes | 23 | 471 | HSDS Config override |
| 16 nodes | 16 | 678 | HSDS Config override |
| | | | |

Conclusions:

- Performance scales fairly well as number of nodes increases
- Not advisable to run more nodes than CPU cores
- (Not shown) performance with Posix rotating disk did not scale at all
- At some point performance will be network bandwidth limited
- (Not shown) performance with Direct Connect or Kubernetes scaled similarly

# Scaling up HSDS by number of clients



- In the previous slide we added more nodes but had just one client sending requests
- How does HSDS perform if we have more clients sending smaller requests?
- HSDS keeps track of number of inflight requests per node and responds with a 503 (Server too Busy) error when that is exceeded
- Polite clients will back off a bit when they see of 503 response
- The nsrdb_async test can simulate an arbitrary number of clients sending requests continuously to server
- How many tasks can we run for a given number of HSDS nodes?
- You can find code for the test here:

https://github.com/HDFGroup/hsds/blob/master/tests/perf/nsrdb/nsrdb_async.py

# Number of Clients - Results



| HSDS on Docker | Task Count | Success rate | Notes |
|---|---|---|---|
| 4 nodes | 10 | 100% | |
| 4 nodes | 12 | 94% | |
| 8 nodes | 20 | 100% | |
| 8 nodes | 25 | 98% | HSDS Config override |
| 8 nodes | 30 | 83% | “ |
| 16 nodes | 40 | 100% | “ |
| 16 nodes | 50 | 98% | “ |
| 16 nodes | 79 | 85% | “ |
| | | | |

Conclusions:

- Number of clients scales linearly with number of nodes
- Performance will degrade if server is over-subscribed
- Kubernetes (not shown) performed similarly
- Lambda or Direct connect has benefit of not requiring matching scaling of client/server

# Next Steps

- More work is needed for AWS Lambda to improve performance
  - Example: currently it sends data in JSON rather than as binary
- Simply config settings needed for Direct Connect  (e.g. ROOT_DIR)
- Remove requirement to extract meta for HDF5 with “hsload –-link”
  - Instead acquire dynamically when file is accessed
- Adding Direct Connect functionality as a VOL for C/C++ clients
- Support Azure Functions (Azure’s version of AWS Lambda)
- Streaming support – process data as bytes are received. Benefits;
  - Remove limit on size of requests
  - Lower latency
  - Reduce memory pressure

# Try it out!



Get the software here:

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

# Questions?