Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 7 May 2026

Z. Shen
CNIC, Chinese Academy of Sciences

3 November 2025

DACP: Data Access and Collaboration Protocol
draft-shenzhihong-dacp-01

Abstract

This document describes the Data Access and Collaboration Protocol (DACP), a communication protocol designed to support cross-node, cross-process data access in scientific and distributed computing environments. DACP provides standardized streaming-based data interactions over the Arrow Flight protocol and defines a unified Streaming DataFrame (SDF) model, which acts as a high-performance abstraction for accessing and processing both structured and unstructured data.

Table of Contents

1. Introduction

Modern data processing, particularly in scientific and distributed computing, requires unified and low-latency data access that spans across different domains, nodes, and processes. However, the inherently fragmented, heterogeneous, and siloed nature of scientific data impedes effective data sharing and collaboration.

The *Data Access and Collaboration Protocol (DACP)* is designed to enable secure, high-performance, and auditable streaming of data across distributed systems. DACP builds upon *Apache Arrow Flight* [Apache-Arrow-Flight] to provide zero-copy, columnar data transfer, while introducing an end-to-end provenance tracking mechanism for data access.

Traditional data access methods, such as REST over JSON, not only incur high serialization overhead but also lack native support for both data stream transport across processes and nodes and multi-hop auditing of data access. To address these limitations, DACP defines Streaming DataFrame (SDF) as a standardized data unit.

Key features include:

* Unified & consistent data representation
* High-performance stream-framed data transport
* Strict authentication and access control
* End-to-end tracking of data flow paths and provenance

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

* *DACP*: Data Access and Collaboration Protocol
* *Streaming DataFrame (SDF)*: The fundamental data unit in DACP.
* *Dataset*: A logical container used to organize and manage multiple named SDFs.
* *DACP Server*: A server that hosts datasets/SDFs and responds to DACP requests.

* *DACP Client*: A client that initiates a DACP connection to request data or metadata.

* *DACP Proxy*: An optional but recommended intermediary service that acts as a secure gateway, enabling clients from an external network (e.g., the public internet) to access DACP Servers located within a private network.

3. Architecture Overview

3.1. Protocol

DACP is an application-layer protocol that governs the communication between a *DACP Client* and *DACP Server*. Its primary purpose is to provide a standardized way to request, query, and stream data across different processes and network nodes.

To achieve this, DACP introduces the *Streaming DataFrame (SDF)* as the unified model for all data streams transmitted by the protocol. The SDF provides a consistent, structured representation for data. The *Dataset* acts as an optional logical container for organizing and sharing related SDFs. While the Dataset (an optional logical container) serves as a key unit for organized data sharing, the SDF remains the fundamental unit for analysis and processing.

Instead of defining a new transport layer from scratch, DACP is built directly on top of *Apache Arrow Flight*. It maps its application-level concepts onto Arrow’s proven capabilities.

3.2. URI Format

DACP resources use this URI scheme:

dacp://host:port/[dataset_name]/path

Where:

* host: Domain name or IP address (required)

* port: Optional port number

* dataset_name: Optional unique name within the host, identifying a specific dataset

* path: Slash-separated path used to identify a specific SDF within the context of the specified Dataset (may include file extensions if applicable; required)

Examples:

* dacp://10.0.0.1:8587

* dacp://10.0.0.1:8587/weather_db

* dacp://10.0.0.1:8587/weather_db/sensors/temperature.csv

3.3. Communication Model

DACP uses a client-initiated request-response model with streaming support:

1. Client establishes a secure (TLS) connection.

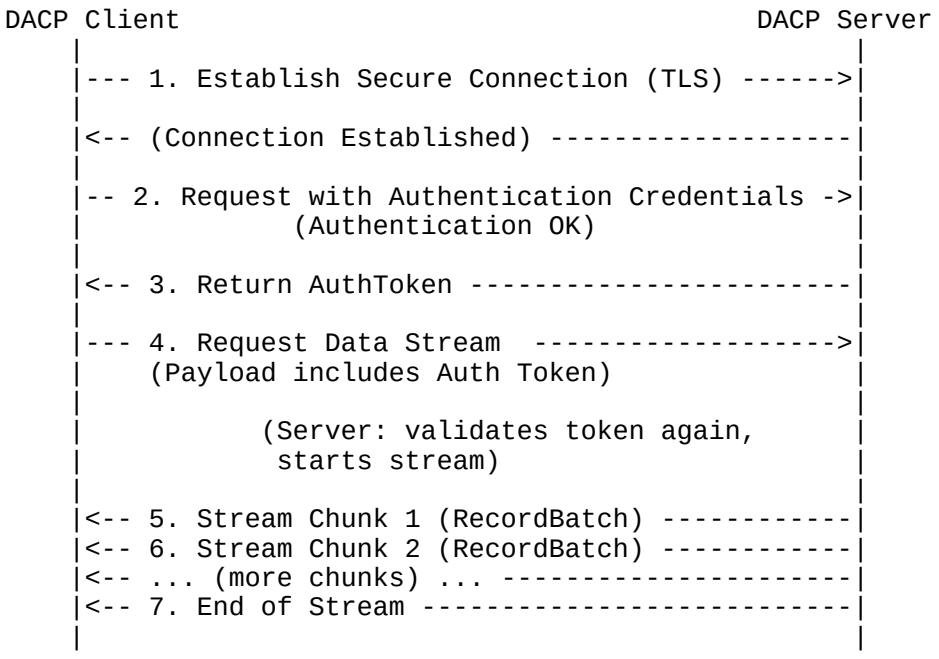
2. Client establishes connection using DACP URI, presenting its long-term credentials (e.g., username/password).

3. Server validates the credentials and, if successful, returns a short-lived Authentication Token (AuthToken). This token serves as proof of identity for all subsequent requests.

4. Client sends a data request to the server. The DACP Application Payload of this request must contain the AuthToken.

5. Server validates the token on every incoming request to ensure the client is authenticated. Large SDFs use streaming in chunks. Link information is updated at each hop.

6. The connection is maintained until closed by the client or server.



4. Streaming DataFrame (SDF)

The Streaming DataFrame (SDF) is the fundamental logical data unit in the DACP protocol. It represents a large-scale, two-dimensional table of columnar data, which is not necessarily derived from a single file.

Key Characteristics:

* *Schema-Driven and Ordered*: Every SDF is defined by a strict schema that includes column names and their corresponding data types. The positions of columns within the schema are fixed, allowing columns to be accessed by their positional index (e.g., _1, _2, _3). To ensure semantic interoperability across different systems, schema registration (via the DACP server's metadata interface, including fields like schema ID, creation time, and type compatibility rules) is *RECOMMENDED*. If a schema is not registered, the server will default to Apache Arrow's native type mapping for interoperability.

* *Immutable*: SDFs are immutable by design. A transformation operation does not change the original SDF but instead defines a new logical SDF as its result.

* *Data Representation and Types*: All values within an SDF conform to the Apache Arrow specification, which defines their type and binary on-the-wire representation.
  - *Null* is a valid and supported value for any data type.
  - The Binary data type is explicitly supported, allowing an SDF to act as a container for unstructured or semi-structured data (e.g., images, documents, blobs).
* *Streaming Native*: An SDF is not a monolithic in-memory object but a logical stream composed of one or more sequential data chunks, typically represented as Arrow RecordBatches. This design is fundamental to DACP, allowing it to handle datasets that vastly exceed the memory capacity of any single node.
* *Addressability and Naming*: SDFs are addressable at two levels:
  - *Host-Local Identifier*: Each SDF has an identifier (e.g., a00001 or ds0001/a/b/1.cdf) that is unique within the scope of its host DACP server. The naming convention is determined by the host.
  - *Globally Unique URI*: A host-local identifier is combined with the server's address to form a globally unique DACP Uniform Resource Identifier (URI), such as dacp://10.0.0.1/a00001. This URI is the canonical way for clients to request a specific SDF.
* *Rich Metadata*: Each SDF can be accompanied by a metadata block that provides critical context about the data. This *MAY* include standardized fields such as rows, totalBytes, lastModified, etc.

5. SDF Operations

The DACP protocol defines a computational model for processing SDFs based on the principles of immutability and lazy evaluation. This allows clients to build complex queries that are executed efficiently on the server.

5.1. Core Principles

* *Immutability*: A source SDF is never modified. A *transformation* operation does not change the original data but instead defines a new, logical SDF as its result.

* *Deferred Execution (Lazy Evaluation)*: Transformations are not executed immediately. They are accumulated client-side into a logical execution plan called a *Transformation Chain*. The entire computation is triggered only when a result-triggering *Action* is invoked. This model naturally supports method chaining, such as df.select("xx").limit(10).collect().

5.2. Operation Categories

SDF operations are categorized by how they interact with the server.

5.2.1. Transformations

Transformations are operations that create a new logical SDF. Common transformations supported by the protocol *SHOULD* include:

* select: Selects a subset of columns.
* map: Applies a function to each row.
* union: Combines two DataFrames.
* sample: Returns a random sample of the data.
* limit: Restricts the result to the first N rows.
* filter: Selects rows that satisfy a given predicate. The filter operation *SHOULD* support two distinct types of predicate expressions, allowing for flexibility in how conditions are specified:
  1. *SQL-style String*: A string containing a standard SQL WHERE clause expression.
     - _Example_: filter("AAA > 1 and BBB = 2")
  2. *Functional Expression*: A string representing a functional or lambda-style expression that evaluates to a boolean for each row.
     - _Example_: filter("row => row(’AAA’) > 1 && row(’BBB’) == 2")

5.2.2. Actions

Actions trigger the execution of the Transformation Chain and produce a final result. Common actions *SHOULD* include:

* collect: Gathers all rows (or the first N rows) of the result and returns them to the client.
* count: Returns the total number of rows in the result.
* first: Returns the first row of the result.
* reduce: Aggregates the elements of the DataFrame.
* foreach: Applies a function to each row (typically for side effects on the server).
* write: Saves the result to a storage system.

5.3. RPC Mapping

Transformations *MUST NOT* trigger network requests. They only append a new entry to the actions array in the client's local DataFrame state representation. Actions trigger server-side execution. The choice of RPC is determined by the action's return type.

* *Data-Returning Actions (via DoGet)*
  - *Operations*: collect, first, reduce, foreach, write.
  - *Result*: The server executes the Transformation Chain and streams back the resulting SDF as an Arrow IPC Stream.
* *Computation-Returning Actions (via DoAction)*
  - *Operations*: sum, mean, count.
  - *Result*: The server executes the query and returns a small result (e.g., a JSON object) in the Result stream.
* *Metadata-Returning Actions (via GetFlightInfo)*
  - *Operations*: schema, num_rows, shape.
  - *Result*: The server plans the query, determines the properties of the result (schema, row count), and returns them in a FlightInfo message without sending any data.

6. Dataset

To facilitate logical organization and data sharing, DACP introduces the optional concept of a *Dataset*. A Dataset is not a physical container but a logical grouping that functions like a dictionary or namespace for one or more related SDFs.

Key characteristics of a Dataset are:

* *Logical, Not Physical*: A Dataset is a metadata construct. An SDF is not required to belong to a Dataset. The fundamental unit of analysis remains the SDF.

* *Flat Namespace*: A Dataset contains a collection of named SDFs. This namespace is flat, not hierarchical; a tree-like folder structure *SHOULD NOT* be assumed. This simplifies data access, as each SDF can be directly addressed.

* *Naming*: A Dataset has a name that is unique within the scope of its host DACP server (e.g., human_face_images).

* *Metadata Inheritance*: One of Dataset’s roles is to provide descriptive metadata. Metadata defined at the Dataset level can be inherited by the SDFs it contains, providing a convenient way to apply common context.

* *Unit of Sharing vs. Unit of Analysis*: In a typical workflow, the *Dataset serves as the basic unit for data sharing*, providing a complete, context-rich package of related data. The *SDF remains the basic unit for data analysis*, representing the actual data to be processed.

7. Data Stream Framing

Data Stream Framing is the process by which a DACP server maps files, folders, and databases into the DACP’s Streaming DataFrame (SDF) model. This process makes diverse data accessible through a unified protocol.

7.1. Framing Strategies

The framing strategy depends on the nature of the source data.

7.1.1. Framing Structured and Semi-Structured Data

Sources with inherent tabular or relational structure are mapped directly into one or more SDFs.

* *Single Tabular File (e.g., CSV)*: Mapped to a single SDF where rows and columns correspond to the file's content.

* *Hierarchical/Scientific File (e.g., NetCDF)*: Mapped to one SDF representing the variables and dimensions within the file.

* *Relational Database (RDBMS)*: Mapped to a Dataset where each table or view becomes a distinct SDF.

* *Knowledge Graph (e.g., RDF)*: Mapped to a Dataset containing separate SDFs for vertices and edges.

7.1.2. Framing Files and File Collections

Not all files can be meaningfully parsed into a multi-row, multi-column SDF (e.g., a JPEG image). For these "opaque" files, or for any collection of files, DACP uses a powerful alternative: *framing the file list itself*.

* *File List Framing*: A list of files (e.g., from a folder or a ZIP archive) is mapped into a single, structured SDF. In this SDF:

  - Each *row* represents a single file.

  - *Columns* describe the file's metadata, such as name, path, suffix, type, size, and modification_time. The schema for this is often referred to as a FileListSchema.

  - A special column, typically named blob and of type Binary, is included. This column represents the file's raw binary content.

  - A url column provides a direct, navigable link to access the content of the item.

* *Lazy Loading of Binary Content*: A critical feature of this model is that the blob column *MUST* be lazy-loaded. The binary content of a file is only read from storage and streamed to the client when that specific column for that specific row is explicitly accessed. This prevents the costly operation of loading all file contents into memory upfront.

7.2. Example: A Mixed-Content Folder

Consider a folder containing results.csv and plot.png. A DACP server could frame this folder as a single SDF with the following structure:

name	path	type	size	time	url	blob
results.csv	/results.csv	File	10240	(datetime)	dacp://.../dataset-x/results.csv	(...)
plot.png	/plot.png	File	204800	(datetime)	dacp://.../dataset-x/plot.png	(...)
logs	/logs	Dir	4096	(datetime)	dacp://.../dataset-x/logs	null

Table 1

The url column is the key to interacting with the folder's contents. Accessing a URL triggers a new DACP request, and the server's response depends on the type of the item:

1. *Accessing a Directory URL*:

    * If a client accesses the URL for the logs directory (dacp://.../dataset-x/logs), the server responds with a *new SDF*.

    * This new SDF represents the contents of the logs/ sub-folder, having the exact same structure (name, type, size, url, etc.). This enables recursive, drill-down navigation through the file system.

2. *Accessing a File URL*:

    * The behavior depends on whether the file is structured or opaque.

    * *Structured File (e.g., results.csv)*: When accessing the URL for results.csv, the server responds with an *SDF representing the parsed content of the CSV file*. The client receives a ready-to-use, multi-row, multi-column DataFrame.

    * *Opaque File (e.g., plot.png)*: When accessing the URL for plot.png, which cannot be meaningfully parsed into a tabular SDF, the server returns a raw binary stream, which can be saved to a file or loaded into an image library.

Lazy Loading of Binary Content:

A critical feature of this model is that the blob column in the initial folder listing *MUST* be lazy-loaded. The binary content of a file is only read from storage and streamed to the client when that specific column for that specific row is explicitly requested (e.g., df[1]['blob']).

8. Authentication and Authorization

Security is a fundamental aspect of the DACP protocol. The security model is designed around the principles of unified authentication and distributed, request-based authorization.

8.1. Authentication

DACP mandates a unified authentication system for all actors (users and services) to ensure a consistent identity layer across the entire ecosystem.

* *Unified Authentication Service*: All hosts *MUST* integrate with a central, unified authentication service. This service is responsible for verifying user credentials and issuing identity tokens.

* *Authentication Flow (OAuth 2.0)*: The standard mechanism for authentication *MAY* be an OAuth 2.0 flow. End-users authenticate with the central service to obtain a short-lived *Bearer Token* (e.g., a JWT).

* *Token Transmission*: This Bearer Token *MUST* be included in the Authorization header of DACP requests sent to a server or proxy.

  - *Format*: Authorization: Bearer <token>

* *Token Validation*: Upon receiving a request, every DACP endpoint (Proxy or Server) *MUST* validate the token. This includes verifying its signature, checking that it has not expired, and confirming that it was issued by the trusted central authentication service.

8.2. Authorization and Access Control

While authentication is centralized, authorization is distributed and managed by the administrators of each individual host. This allows data owners to have full control over their resources.

The authorization model follows a dynamic, request-and-approval workflow:

1. *Default State*: The host administrator explicitly defines which resources are publicly accessible and which require specific authorization.

2. *Access Request*: To access a restricted resource, a user must submit an access request. This request specifies the user, the target dataset(s), and the expiration time.

3. *Administrative Approval*: The administrator of the host that owns the resource reviews the request. If approved, the administrator's action triggers the issuance of a new, permission-scoped token for the user.

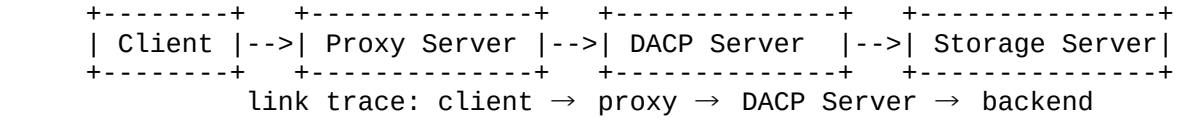
4. *Scoped Token Issuance*: Upon approval, the central system issues a new token to the user. This token contains scopes that explicitly grant the approved permissions (e.g., dacp://host-a/dataset-b).

5. *Policy Enforcement*: When a user makes a request using this new token, the DACP server validates that the token's scopes (e.g., dacp://host-a/dataset-b) match the requested resource and operation (e.g., read, stream), ensuring the user has the necessary permissions for the requested operation on the target resource. Access is granted only if all these checks pass.

9. Provenance Tracking

DACP maintains a provenance trail that records critical information for each hop in the connection chain, including machine identifiers, timestamps, and traffic metrics. This is particularly crucial for auditability in multi-hop environments involving proxies and gateways, tracking the full request path (e.g., client_ip -> proxy_server_ip -> server_ip -> storage_server_ip).

Each intermediate node MUST append its own entry to the provenance trail. The trail ensures auditability even in untrusted network environments.



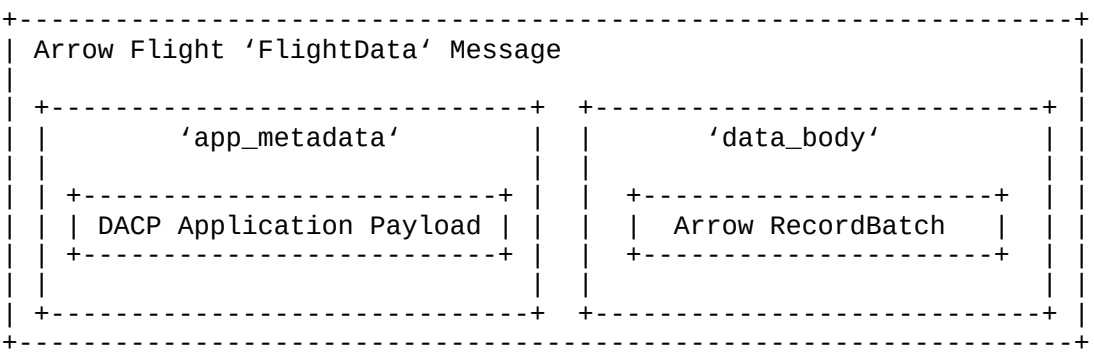
10. DACP Message Structure

10.1. The Message Transport: FlightData

DACP defines a single, unified application payload that encapsulates all control, security, and metadata information. All DACP data and metadata are transported within standard Arrow Flight FlightData messages. A FlightData message logically separates application-level metadata from the raw data payload, consisting of two primary components:

* *app_metadata*: A byte field intended for use by higher-level protocols. DACP places its *application payload*, including headers and provenance information, into this field. The Arrow Flight framework treats this data as opaque and simply transports it from sender to receiver.

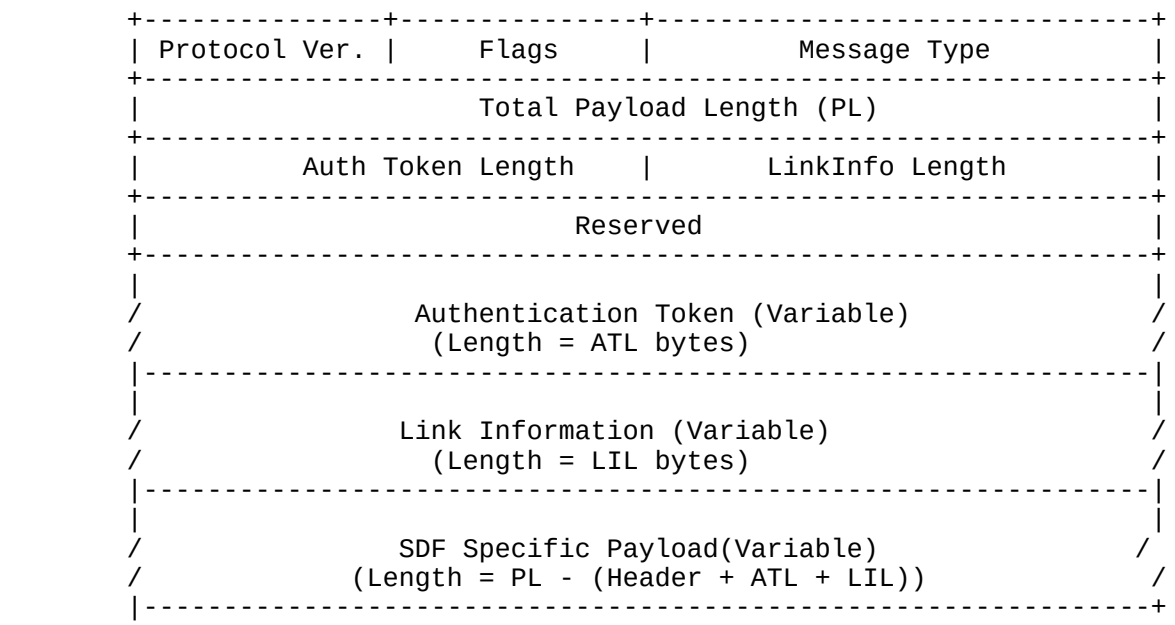
* *data_body*: A byte field that contains the raw, serialized Arrow *columnar data (RecordBatch)*. This design enables zero-copy data access on the receiving end, as the data can be used directly without parsing or transformation.



10.2. DACP Application Payload Structure

A DACP payload is structured as follows:

The DACP Application Payload is a single, contiguous block of bytes that contains header, auth token and link information for a DACP message. It is designed to be self-describing, allowing any recipient to parse it by first reading a fixed-size header.



DACP Header fields:

* *Protocol Version (1 byte)*: Allows evolution (e.g. 0x01).

* *Flags (1 byte)*: 8 bits for control flags (e.g., 0x02 for End of Stream).

* *Message Type (2 bytes)*: An identifier for the DACP operation (e.g., SDF Request/Response).

* *Total Payload Length (4 bytes)*: The total size in bytes of the entire DACP Application Payload.

* *Auth Token Length (2 bytes)*: The size in bytes of the Authentication Token block.

* *Link Info Length (2 bytes)*: The size in bytes of the Link Information block.

* *Reserved (4 bytes):* Reserved for future use, such as a token expiry timestamp.

Authentication Token: A variable-length field containing the security token required for authentication. This is the first component to be processed by a receiving node.

Link Information: A variable-length field containing the serialized provenance trail. The trail is a serialized LinkInfo message, which is a list of HopInfo entries. A HopInfo SHOULD contain the following fields:

* *proxy/server id*: A unique identifier for the DACP proxy/server that processed the message (e.g., hostname).

* *proxy ip*: The IP address of the proxy.

* *timestamp*: A high-precision UTC timestamp.

* *authenticated_user*: The identity of the user as authenticated by this hop.

* *bytes_transferred*: The number of bytes of the primary payload (e.g., SDF data) processed or forwarded.

SDF Specific Payload: A JSON object that encapsulates a complete SDF query. The SDF Specific Payload MUST contain the following keys:

* *id*: A string representing the URI of the source SDF upon which the operations are based.

* *actions*: An ordered array representing the chain of transformations. Each element in the array is a tuple [operation_name, parameters_object].

Example SDF Specific Payload:

```
{
  "id": "dacp://10.0.0.1/weather_db/sensors",
  "actions": [
    ["filter", {"expression": "temperature > 25.0"}],
    ["select", {"columns": ["location", "temperature"]}],
    ["limit", {"n": 100}]
  ]
}
```

11. IANA Considerations

This document requests that IANA perform the following actions.

This document requests the registration of the "dacp" Uniform Resource Identifier (URI) scheme in the "Uniform Resource Identifier (URI) Schemes" registry.

* *Scheme name:* dacp

* *Status:* Permanent

* *Applications/protocols that use this scheme:* Data Access and Collaboration Protocol (DACP).

* *URI Scheme Syntax:* The syntax is specified in Section 3.2 of this document.

* *Reference:* This document.

This document requests the assignment of a TCP port number in the "Service Name and Transport Protocol Port Number Registry".

* *Service Name:* dacp

* *Port Number:* 8587 (Suggested)

* *Transport Protocol(s):* TCP

* *Description:* Data Access and Collaboration Protocol

* *Reference:* This document.

This document requests the registration of the application/dacp+arrow media type in the "Media Types" registry.

* *Type name:* application

* *Subtype name:* dacp+arrow

* *Required parameters:* None

* *Optional parameters:* None

* *Encoding considerations:* binary

12. Security Considerations

The DACP protocol relies on the security of the underlying transport, Apache Arrow Flight, which in turn uses TLS (typically TLS 1.3 [RFC8446]) for connection security. All security considerations applicable to TLS and gRPC apply to DACP. Implementations MUST support TLS 1.3 or a subsequent version.

Authentication is managed via Bearer Tokens, as described in Section 8.1. Implementers must be aware of the security risks associated with Bearer Tokens, such as interception and replay attacks. The use of short-lived tokens and secure storage on the client side is strongly RECOMMENDED.

Authorization policies are enforced at each DACP server, as described in Section 8.2. Administrators of DACP hosts are responsible for correctly configuring access control policies to prevent unauthorized data access.

The provenance tracking mechanism described in Section 9 is designed for auditability but does not, by itself, prevent malicious actors from tampering with the provenance trail if they compromise an intermediate node. The integrity of the trail relies on the security of each hop in the chain.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

13.2. Informative References

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[Apache-Arrow-Flight] Apache Arrow Community, "Apache Arrow Flight", 2023, <https://arrow.apache.org/docs/format/Flight.html>.

Appendix A. Appendix 1. Python Client Usage Example

This appendix provides a practical example of how to use the DACP Python sdk (DacpClient) to connect to a server, explore datasets, and perform common data manipulation tasks on a Streaming DataFrame (SDF).

```
import logging
from dacp_client import DacpClient, Principal

# Configure logging for demonstration purposes
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

print("--- 1. Connecting to the dacp Service ---")

# Connect using username and password via OAuth
url = "dacp://60.245.194.25:8587"
username = "your_username"  # Replace with actual username
password = "your_password"  # Replace with actual password
conn = DacpClient.connect(url, Principal.oauth(username, password))

print("\n--- 2. Discovering Datasets and DataFrames ---")
# List all available datasets
datasets = conn.list_datasets()
print(f"Found datasets: {datasets}")

# Get metadata for the first dataset
dataset_metadata = conn.get_dataset(datasets[0])
print(f"Metadata for '{datasets[0]}': {dataset_metadata}")

# List all DataFrames within that dataset
dataframes = conn.list_dataframes(datasets[0])
print(f"DataFrames in '{datasets[0]}': {dataframes}")

# Get the name of the first DataFrame to work with
dataframe_name = dataframes[0]['dataframeName']
print(f"Opening DataFrame: '{dataframe_name}'")

# Open the DataFrame to create a client-side handle
df = conn.open(dataframe_name)

print("\n--- 3. Inspecting DataFrame Properties ---")
logger.info(f"Schema: {df.schema}")
logger.info(f"Shape (rows, cols): {df.shape}")
logger.info(f"Number of rows: {df.num_rows}")  # Or len(df)
logger.info(f"Number of columns: {df.num_cols}")
logger.info(f"Column names: {df.column_names}")
```

```
   logger.info(f"Total size in bytes: {df.total_bytes}")

   print("\n--- 4. Previewing and Printing Data ---")
   # Generate a formatted string representation of
   # the DataFrame's head and tail
   data_preview_str = df.to_string(head_rows=5, tail_rows=5,
                                   first_cols=3, last_cols=3)
   logger.info(f"DataFrame Preview:\n{data_preview_str}\n")
   # Note: Simply printing the object might also provide a summary
   # logger.info(df)

   print("\n--- 5. Streaming Data in Chunks ---")
   # Iterate over the data in chunks of up to 100 rows at a time
   for chunk in df.get_stream(max_chunksize=100):
       logger.info(f"Processing chunk with {chunk.num_rows} rows...")
       # logger.info(chunk) # You can print or process the chunk here

   print("\n--- 6. Selecting and Filtering Data ---")
   ## Column Selection
   logger.info(f"Selecting a single column:
                   {df['col1']}\n")
   logger.info(f"Selecting multiple columns:
                   {df.select('col1', 'col2', 'col3')}\n")

   ## Row Selection and Slicing
   logger.info(f"Selecting the first row (index 0):
                   {df[0]}\n")
   logger.info(f"Selecting a specific cell (row 0, column 'col1'):
                   {df[0]['col1']}\n")
   logger.info(f"Selecting the first 10 rows:
                   {df.limit(10)}\n")
   logger.info(f"Selecting rows from index 2 up to
                   (but not including) 4:{df.slice(2, 4)}\n")

   ## Conditional Filtering using expressions
   # Example 1: Filter rows where 'col1' is less than or equal to 30
   expression1 = "col1 <= 30"
   # Example 2: Filter rows where 'col2' equals a specific string
   expression2 = "col2 == 'example_string'"
   # Example 3: Combine conditions on multiple columns
   expression3 = "(col1 > 10) & (col3 < 50)"
   # Example 4: Filter rows where 'col4' is in a list of values
   expression4 = "col4.isin([1, 2, 3])"
   # Example 5: Filter rows where 'col5' is not null
   expression5 = "col5.notnull()"
   # Example 6: A more complex combination of conditions
   expression6 = "((col1 < 10) | (col2 == 'example_string'))
                                & (col3 != 0)"
```

```
   logger.info(f"Result of filtering with '{expression1}':
                  {df.filter(expression1)}\n")

   print("\n--- 7. Performing SQL Queries ---")
   sql_query = (
       "SELECT OBJECTID, start_l, end_l "
       "FROM dataframe "
       "WHERE OBJECTID <= 30 "
       "ORDER BY OBJECTID DESC"
   )
   sql_result = df.sql(sql_query)
   logger.info(f"SQL query result:\n{sql_result}")
```

Appendix B. References

B.1. Normative References

[RFC2119]
[RFC8174]
[RFC8446]

B.2. Informative References

[RFC6838]
[Apache-Arrow-Flight]

Author's Address

Zhihong Shen
CNIC, Chinese Academy of Sciences
Email: bluejoe@cnic.cn