Workgroup: Network Working Group
Internet-Draft: draft-ietf-ppm-dap-10
Published: 29 February 2024
Intended Status: Standards Track
Expires: 1 September 2024
Authors: T. Geoghegan
ISRG
C. Patton
Cloudflare
B. Pitman
ISRG
E. Rescorla
Mozilla
C. A. Wood
Cloudflare

Distributed Aggregation Protocol for Privacy Preserving Measurement

Abstract

There are many situations in which it is desirable to take measurements of data which people consider sensitive. In these cases, the entity taking the measurement is usually not interested in people's individual responses but rather in aggregated data. Conventional methods require collecting individual responses and then aggregating them, thus representing a threat to user privacy and rendering many such measurements difficult and impractical. This document describes a multi-party distributed aggregation protocol (DAP) for privacy preserving measurement (PPM) which can be used to collect aggregate data without revealing any individual user's data.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://ietf-wg-ppm.github.io/draft-ietf-ppm-dap/draft-ietf-ppm-dap.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-ppm-dap/.

Discussion of this document takes place on the Privacy Preserving Measurement Working Group mailing list (mailto:ppm@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/ppm/. Subscribe at https://www.ietf.org/mailman/listinfo/ppm/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-ppm/draft-ietf-ppm-dap.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute

working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 September 2024.

Copyright Notice

Table of Contents

1. Introduction

This document describes the Distributed Aggregation Protocol (DAP) for privacy preserving measurement. The protocol is executed by a large set of clients and two aggregator servers. The aggregators' goal is to compute some aggregate statistic over the clients' inputs without learning the inputs themselves. This is made possible by distributing the computation among the aggregators in such a way that, as long as at least one of them executes the protocol honestly, no input is ever seen in the clear by any aggregator.

1.1. Change Log

(*) Indicates a change that breaks wire compatibility with the previous draft.

10:

- *Editorial changes from httpdir early review.
- *Poll collection jobs with HTTP GET instead of POST. (*)
- *Upload reports with HTTP POST instead of PUT. (*)
- *Clarify requirements for problem documents.
- *Provide guidance on batch sizes when running VDAFs with non-trivial aggregation parameters.
- *Bump version tag from "dap-09" to "dap-10". (*)

09:

- *Fixed-size queries: make the maximum batch size optional.
- *Fixed-size queries: require current-batch queries to return distinct batches.
- *Clarify requirements for compatible VDAFs.
- *Clarify rules around creating and abandoning aggregation jobs.
- *Recommend that all task parameters are visible to all parties.
- *Revise security considerations section.
- *Bump draft-irtf-cfrg-vdaf-07 to 08 [VDAF]. (*)
- *Bump version tag from "dap-07" to "dap-09". (*)

08:

* Clarify requirements for initializing aggregation jobs.

* Add more considerations for Sybil attacks.

* Expand guidance around choosing the VDAF verification key.

* Add an error type registry for the aggregation sub-protocol.

07:

* Bump version tag from "dap-06" to "dap-07". This is a bug-fix revision: the editors overlooked some changes we intended to pick up in the previous version. (*)

06:

* Bump draft-irtf-cfrg-vdaf-06 to 07 [VDAF]. (*)

* Overhaul security considerations (#488).

* Adopt revised ping-pong interface in draft-irtf-cfrg-vdaf-07 (#494).

* Add aggregation parameter to AggregateShareAad (#498). (*)

* Bump version tag from "dap-05" to "dap-06". (*)

05:

* Bump draft-irtf-cfrg-vdaf-05 to 06 [VDAF]. (*)

* Specialize the protocol for two-party VDAFs (i.e., one Leader and One Helper). Accordingly, update the aggregation sub-protocol to use the new "ping-pong" interface for two-party VDAFs introduced in draft-irtf-cfrg-vdaf-06. (*)

* Allow the following actions to be safely retried: aggregation job creation, collection job creation, and requesting the Helper's aggregate share.

* Merge error types that are related.

* Drop recommendation to generate IDs using a cryptographically secure pseudorandom number generator wherever pseudorandomness is not required.

* Require HPKE config identifiers to be unique.

* Bump version tag from "dap-04" to "dap-05". (*)

04:

* Introduce resource oriented HTTP API. (#278, #398, #400) (*)

* Clarify security requirements for choosing VDAF verify key. (#407, #411)

* Require Clients to provide nonce and random input when sharding inputs. (#394, #425) (*)

* Add interval of time spanned by constituent reports to Collection message. (#397, #403) (*)

* Update share validation requirements based on latest security analysis. (#408, #410)

* Bump draft-irtf-cfrg-vdaf-03 to 05 [VDAF]. (#429) (*)

* Bump version tag from "dap-03" to "dap-04". (#424) (*)

03:

* Enrich the "fixed_size" query type to allow the Collector to request a recently aggregated batch without knowing the batch ID in advance. ID discovery was previously done out-of-band. (*)

* Allow Aggregators to advertise multiple HPKE configurations. (*)

* Clarify requirements for enforcing anti-replay. Namely, while it is sufficient to detect repeated report IDs, it is also enough to detect repeated IDs and timestamps.

* Remove the extensions from the Report and add extensions to the plaintext payload of each ReportShare. (*)

* Clarify that extensions are mandatory to implement: If an Aggregator does not recognize a ReportShare's extension, it must reject it.

* Clarify that Aggregators must reject any ReportShare with repeated extension types.

* Specify explicitly how to serialize the Additional Authenticated Data (AAD) string for HPKE encryption. This clarifies an ambiguity in the previous version. (*)

* Change the length tag for the aggregation parameter to 32 bits. (*)

* Use the same prefix ("application") for all media types. (*)

* Make input share validation more explicit, including adding a new ReportShareError variant, "report_too_early", for handling reports too far in the future. (*)

* Improve alignment of problem details usage with [RFC7807]. Replace "reportTooLate" problem document type with "repjortRejected" and clarify handling of rejected reports in the upload sub-protocol. (*)

* Bump version tag from "dap-02" to "dap-03". (*)

02:

* Define a new task configuration parameter, called the "query type", that allows tasks to partition reports into batches in different ways. In the current draft, the Collector specifies a "query", which the Aggregators use to guide selection of the batch. Two query types are defined: the "time_interval" type captures the semantics of draft 01; and the "fixed_size" type allows the Leader to partition the reports arbitrarily, subject to the constraint that each batch is roughly the same size. (*)

* Define a new task configuration parameter, called the task "expiration", that defines the lifetime of a given task.

* Specify requirements for HTTP request authentication rather than a concrete scheme. (Draft 01 required the use of the DAP-Auth-Token header; this is now optional.)

* Make "task_id" an optional parameter of the "/hpke_config" endpoint.

* Add report count to CollectResp message. (*)

* Increase message payload sizes to accommodate VDAFs with input and aggregate shares larger than 2^16-1 bytes. (*)

* Bump draft-irtf-cfrg-vdaf-01 to 03 [VDAF]. (*)

* Bump version tag from "dap-01" to "dap-02". (*)

* Rename the report nonce to the "report ID" and move it to the top of the structure. (*)

* Clarify when it is safe for an Aggregator to evict various data artifacts from long-term storage.

1.2. Conventions and Definitions

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.

Aggregate result: The output of the aggregation function computed over a batch of measurements and an aggregation parameter. As defined in [VDAF].

Aggregate share: A share of the aggregate result emitted by an Aggregator. Aggregate shares are reassembled by the Collector into the aggregate result, which is the final output of the aggregation function. As defined in [VDAF].

Aggregation function: The function computed over the Clients' measurements. As defined in [VDAF].

Aggregation parameter: Parameter used to prepare a set of measurements for aggregation (e.g., the candidate prefixes for Poplar1 from Section 8 of [VDAF]). As defined in [VDAF].

Aggregator: A server that receives input shares from Clients and validates and aggregates them with the help of the other Aggregators.

Batch: A set of reports (i.e., measurements) that are aggregated into an aggregate result.

Batch duration: The time difference between the oldest and newest report in a batch.

Batch interval: A parameter of a query issued by the Collector that specifies the time range of the reports in the batch.

Client: The DAP protocol role identifying a party that uploads a report. Note the distinction between a DAP Client (distinguished in this document by the capital "C") and an HTTP client (distinguished in this document by the phrase HTTP client), as the DAP Client is not the only role that sometimes acts as an HTTP client.

Collector: The party that selects the aggregation parameter and receives the aggregate result.

Helper: The Aggregator that executes the aggregation and collection interactions as instructed by the Leader.

Input share: An Aggregator's share of a measurement. The input shares are output by the VDAF sharding algorithm. As defined in [VDAF].

Output share: An Aggregator's share of the refined measurement resulting from successful execution of the VDAF preparation phase. Many output shares are combined into an aggregate share during the VDAF aggregation phase. As defined in [VDAF].

Leader: The Aggregator that coordinates aggregation and collection with the Helper.

Measurement: A plaintext input emitted by a Client (e.g., a count, summand, or string), before any encryption or secret sharing is applied. Depending on the VDAF in use, multiple values may be grouped into a single measurement. As defined in [VDAF].

Minimum batch size: The minimum number of reports in a batch.

Public share: The output of the VDAF sharding algorithm broadcast to each of the Aggregators. As defined in [VDAF].

Report: A cryptographically protected measurement uploaded to the Leader by a Client. Comprised of a set of report shares.

Report Share: An encrypted input share comprising a piece of a report.

This document uses the presentation language of [RFC8446] to define messages in the DAP protocol. Encoding and decoding of these messages as byte strings also follows [RFC8446].

2. Overview

The protocol is executed by a large set of Clients and a pair of servers referred to as "Aggregators". Each Client's input to the protocol is its measurement (or set of measurements, e.g., counts of some user behavior). Given the input set of measurements x_1, ..., x_n held by n Clients, and an aggregation parameter p shared by the Aggregators, the goal of DAP is to compute y = F(p, x_1, ..., x_n) for some function F while revealing nothing else about the measurements. We call F the "aggregation function."

This protocol is extensible and allows for the addition of new cryptographic schemes that implement the VDAF interface specified in [VDAF]. Candidates include:

* Prio3 (Section 7 of [VDAF]), which allows for aggregate statistics such as sum, mean, histograms, etc.

* Poplar1 (Section 8 of [VDAF]), which allows for finding the most popular strings uploaded by a set of Clients (e.g., the URL of their home page) as well as counting the number of Clients that hold a given string. This VDAF is the basis of the Poplar protocol of [BBCGGI21], which is designed to solve the heavy hitters problem in a privacy preserving manner.

VDAFs rely on secret sharing to protect the privacy of the measurements. Rather than sending its input in the clear, each Client shards its measurement into a pair of "input shares" and sends an input share to each of the Aggregators. This provides two important properties:

* Given only one of the input shares, it is impossible to deduce the plaintext measurement from which it was generated.

* It allows the Aggregators to compute the aggregation function by first aggregating up their input shares locally into "aggregate shares", then combining the aggregate shares into the aggregate result.

2.1. System Architecture

The overall system architecture is shown in Figure 1.

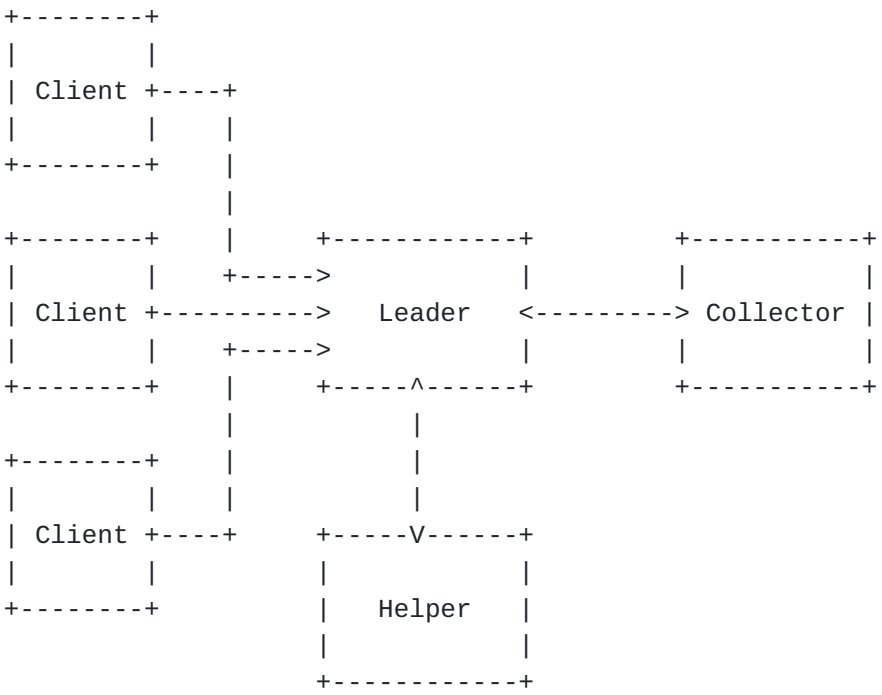

Figure 1: System Architecture

The main participants in the protocol are as follows:

Collector:
The entity which wants to obtain the aggregate of the measurements generated by the Clients. Any given measurement task will have a single Collector.

Client(s): The parties which directly take the measurement(s) and report them to the DAP protocol. In order to provide reasonable levels of privacy, there must be a large number of Clients.

Aggregator: A server which receives report shares. Each Aggregator works with its co-Aggregator to compute the aggregate result. Any given measurement task will have two Aggregators: a Leader and a Helper.

Leader: The Aggregator responsible for coordinating the protocol. It receives the reports, splits them into report shares, distributes the report shares to the Helper, and orchestrates the process of computing the aggregate result as requested by the Collector.

Helper: The Aggregator assisting the Leader with the computation. The protocol is designed so that the Helper is relatively lightweight, with most of the operational burden borne by the Leader.

The basic unit of DAP is the "task" which represents a single measurement process (though potentially aggregating multiple batches of measurements). The definition of a task includes the following parameters:

*The type of each measurement.

*The aggregation function to compute (e.g., sum, mean, etc.).

*The set of Aggregators and necessary cryptographic keying material to use.

*The VDAF to execute, which to some extent is dictated by the previous choices.

*The minimum "batch size" of reports which can be aggregated.

*The rate at which measurements can be taken, i.e., the "minimum batch duration".

These parameters are distributed to the Clients, Aggregators, and Collector before the task begins. This document does not specify a distribution mechanism, but it is important that all protocol participants agree on the task's configuration. Each task is

identified by a unique 32-byte ID which is used to refer to it in protocol messages.

During the lifetime of a task, each Client records its own measurement value(s), packages them up into a report, and sends them to the Leader. Each share is separately encrypted for each Aggregator so that even though they pass through the Leader, the Leader is unable to see or modify them. Depending on the task, the Client may only send one report or may send many reports over time.

The Leader distributes the shares to the Helper and orchestrates the process of verifying them (see Section 2.2) and assembling them into a final aggregate result for the Collector. Depending on the VDAF, it may be possible to incrementally process each report as it comes in, or may be necessary to wait until the entire batch of reports is received.

2.2. Validating Inputs

An essential task of any data collection pipeline is ensuring that the data being aggregated is "valid". In DAP, input validation is complicated by the fact that none of the entities other than the Client ever sees that Client's plaintext measurement.

In order to address this problem, the Aggregators engage in a secure, multi-party computation specified by the chosen VDAF [VDAF] in order to prepare a report for aggregation. At the beginning of this computation, each Aggregator is in possession of an input share uploaded by the Client. At the end of the computation, each Aggregator is in possession of either an "output share" that is ready to be aggregated or an indication that a valid output share could not be computed.

To facilitate this computation, the input shares generated by the Client include information used by the Aggregators during aggregation in order to validate their corresponding output shares. For example, Prio3 includes a zero-knowledge proof of the input's validity (see Section 7.1 of [VDAF]). which the Aggregators can jointly verify and reject the report if it cannot be verified. However, they do not learn anything about the individual report other than that it is valid.

The specific properties attested to in the proof vary depending on the measurement being taken. For instance, to measure the time the user took performing a given task the proof might demonstrate that the value reported was within a certain range (e.g., 0-60 seconds). By contrast, to report which of a set of N options the user select, the report might contain N integers and the proof would demonstrate that N-1 were 0 and the other was 1.

It is important to recognize that "validity" is distinct from "correctness". For instance, the user might have spent 30s on a task but the Client might report 60s. This is a problem with any measurement system and DAP does not attempt to address it; it merely ensures that the data is within acceptable limits, so the Client could not report 10^6s or -20s.

3. Message Transport

Communications between DAP participants are carried over HTTP [RFC9110]. Use of HTTPS is **REQUIRED** to provide server authentication and confidentiality.

3.1. HTTPS Request Authentication

DAP is made up of several interactions in which different subsets of the protocol's participants interact with each other.

In those cases where a channel between two participants is tunneled through another protocol participant, DAP mandates the use of public-key encryption using [HPKE] to ensure that only the intended recipient can see a message in the clear.

In other cases, DAP requires HTTP client authentication as well as server authentication. Any authentication scheme that is composable with HTTP is allowed. For example:

* [OAuth2] credentials are presented in an Authorization HTTP header, which can be added to any DAP protocol message.

* TLS client certificates can be used to authenticate the underlying transport.

* The DAP-Auth-Token HTTP header described in [I-D.draft-dcook-ppm-dap-interop-test-design-04].

This flexibility allows organizations deploying DAP to use existing well-known HTTP authentication mechanisms that they already support. Discovering what authentication mechanisms are supported by a DAP participant is outside of this document's scope.

3.2. Errors

Errors can be reported in DAP both as HTTP status codes and as problem detail objects [RFC9457] in the response body. For example, if the HTTP client sends a request using a method not allowed in this document, then the server **MAY** return HTTP status 405 Method Not Allowed.

When the server responds with an error status code, it **SHOULD** provide additional information using a problem detail object. If the response body does consist of a problem detail object, the HTTP status code **MUST** indicate a client or server error (the 4xx or 5xx classes, respectively, from Section 15 of [RFC9110]).

To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field (within the DAP URN namespace "urn:ietf:params:ppm:dap:error:"):

Type	Description
invalidMessage	A message received by a protocol participant could not be parsed or otherwise was invalid.
unrecognizedTask	A server received a message with an unknown task ID.
unrecognizedAggregationJob	A server received a message with an unknown aggregation job ID.
outdatedConfig	The message was generated using an outdated configuration.
reportRejected	Report could not be processed for an unspecified reason.
reportTooEarly	Report could not be processed because its timestamp is too far in the future.
batchInvalid	The batch boundary check for Collector's query failed.
invalidBatchSize	There are an invalid number of reports in the batch.
batchQueriedTooManyTimes	The maximum number of batch queries has been exceeded for one or more reports included in the batch.
batchMismatch	Aggregators disagree on the report shares that were aggregated in a batch.
unauthorizedRequest	Authentication of an HTTP request failed (see Section 3.1).
missingTaskID	HPKE configuration was requested without specifying a task ID.
stepMismatch	The Aggregators disagree on the current step of the DAP aggregation protocol.
batchOverlap	A request's query includes reports that were previously collected in a different batch.

Table 1

This list is not exhaustive. The server **MAY** return errors set to a URI other than those defined above. Servers **MUST NOT** use the DAP URN namespace for errors not listed in the appropriate IANA registry

(see Section 8.3). The "detail" member of the Problem Details document includes additional diagnostic information.

When the task ID is known (see Section 4.2), the problem document **SHOULD** include an additional "taskid" member containing the ID encoded in Base 64 using the URL and filename safe alphabet with no padding defined in Sections 5 and 3.2 of [RFC4648].

In the remainder of this document, the tokens in the table above are used to refer to error types, rather than the full URNs. For example, an "error of type 'invalidMessage'" refers to an error document with "type" value "urn:ietf:params:ppm:dap:error:invalidMessage".

This document uses the verbs "abort" and "alert with [some error message]" to describe how protocol participants react to various error conditions. This implies HTTP status code 400 Bad Request unless explicitly specified otherwise.

4. Protocol Definition

DAP has three major interactions which need to be defined:

* Uploading reports from the Client to the Aggregators, specified in Section 4.4

* Computing the results for a given measurement task, specified in Section 4.5

* Collecting aggregated results, specified in Section 4.6

Each of these interactions is defined in terms of "resources". In this section we define these resources and the messages used to act on them.

The following are some basic type definitions used in other messages:

```
/* ASCII encoded URL. e.g., "https://example.com" */
opaque Url<1..2^16-1>;

uint64 Duration; /* Number of seconds elapsed between two instants */

uint64 Time; /* seconds elapsed since start of UNIX epoch */

/* An interval of time of length duration, where start is included and (
duration) is excluded. */
struct {
  Time start;
  Duration duration;
} Interval;

/* An ID used to uniquely identify a report in the context of a DAP task
opaque ReportID[16];

/* The various roles in the DAP protocol. */
enum {
  collector(0),
  client(1),
  leader(2),
  helper(3),
  (255)
} Role;

/* Identifier for a server's HPKE configuration */
uint8 HpkeConfigId;

/* An HPKE ciphertext. */
struct {
  HpkeConfigId config_id;    /* config ID */
  opaque enc<1..2^16-1>;     /* encapsulated HPKE key */
  opaque payload<1..2^32-1>; /* ciphertext */
} HpkeCiphertext;

/* Represent a zero-length byte string. */
struct {} Empty;
```

DAP uses the 16-byte ReportID as the nonce parameter for the VDAF shard and prep_init methods (see [VDAF], Section 5). Additionally, DAP includes messages defined in the VDAF specification encoded as opaque byte strings within various DAP messages. Thus, for a VDAF to be compatible with DAP, it **MUST** specify a NONCE_SIZE of 16 bytes, and **MUST** specify encodings for the following VDAF types:

- *PublicShare

- *InputShare

*AggParam

*AggShare

*PrepShare

*PrepMessage

4.1. Queries

Aggregated results are computed based on sets of reports, called "batches". The Collector influences which reports are used in a batch via a "query." The Aggregators use this query to carry out the aggregation flow and produce aggregate shares encrypted to the Collector.

This document defines the following query types:

```
enum {
  reserved(0), /* Reserved for testing purposes */
  time_interval(1),
  fixed_size(2),
  (255)
} QueryType;
```

The time_interval query type is described in Section 4.1.1; the fixed_size query type is described in Section 4.1.2. Future specifications may introduce new query types as needed (see Section 8.2.1). Implementations are free to implement only a subset of the available query types.

A query includes parameters used by the Aggregators to select a batch of reports specific to the given query type. A query is defined as follows:

```
opaque BatchID[32];

enum {
  by_batch_id(0),
  current_batch(1),
  (255)
} FixedSizeQueryType;

struct {
  FixedSizeQueryType query_type;
  select (FixedSizeQuery.query_type) {
    case by_batch_id: BatchID batch_id;
    case current_batch: Empty;
  }
} FixedSizeQuery;

struct {
  QueryType query_type;
  select (Query.query_type) {
    case time_interval: Interval batch_interval;
    case fixed_size: FixedSizeQuery fixed_size_query;
  }
} Query;
```

The query is issued in-band as part of the collect interaction (Section 4.6). Its content is determined by the "query type", which in turn is encoded by the "query configuration" configured out-of-band. All query types have the following configuration parameters in common:

- min_batch_size - The smallest number of reports the batch is allowed to include. In a sense, this parameter controls the degree of privacy that will be obtained: the larger the minimum batch size, the higher degree of privacy. However, this ultimately depends on the application and the nature of the measurements and aggregation function.

- time_precision - Clients use this value to truncate their report timestamps; see Section 4.4. Additional semantics may apply, depending on the query type. (See Section 4.6.5 for details.)

The parameters pertaining to specific query types are described in Section 4.1.1 and Section 4.1.2.

4.1.1. Time-interval Queries

The first query type, time_interval, is designed to support applications in which reports are collected over a long period of time. The Collector specifies a "batch interval" that determines the time range for reports included in the batch. For each report in the

batch, the time at which that report was generated (see Section 4.4) **MUST** fall within the batch interval specified by the Collector.

Typically the Collector issues queries for which the batch intervals are continuous, monotonically increasing, and have the same duration. For example, the sequence of batch intervals (1659544000, 1000), (1659545000, 1000), (1659546000, 1000), (1659547000, 1000) satisfies these conditions. (The first element of the pair denotes the start of the batch interval and the second denotes the duration.) Of course, there are cases in which Collector may need to issue queries out-of-order. For example, a previous batch might need to be queried again with a different aggregation parameter (e.g, for Poplar1). In addition, the Collector may need to vary the duration to adjust to changing report upload rates.

4.1.2. Fixed-size Queries

The fixed_size query type is used to support applications in which the Collector needs the ability to strictly control the batch size. This is particularly important for controlling the amount of noise added to reports by Clients (or added to aggregate shares by Aggregators) in order to achieve differential privacy.

For this query type, the Aggregators group reports into arbitrary batches such that each batch has roughly the same number of reports. These batches are identified by opaque "batch IDs", allocated in an arbitrary fashion by the Leader.

To get the aggregate of a batch, the Collector issues a query specifying the batch ID of interest (see Section 4.1). The Collector may not know which batch ID it is interested in; in this case, it can also issue a query of type current_batch, which allows the Leader to select a recent batch to aggregate. The Leader **MUST** select a batch that has not yet been associated with a collection job.

In addition to the minimum batch size common to all query types, the configuration may include a parameter max_batch_size that determines maximum number of reports per batch.

If the configuration does not include max_batch_size, then the Aggregators can output any batch size that is larger than or equal to min_batch_size. This is useful for applications that are not concerned with sample size, i.e., the privacy guarantee is not affected by the sampling rate of the population, therefore a larger than expected batch size does not weaken the designed privacy guarantee.

Implementation note: The goal for the Aggregators is to aggregate the same number of reports in each batch. The target batch size is deployment-specific, and may be equal to or greater than the minimum

batch size. Deciding how soon batches should be output is also deployment-specific. Exactly sizing batches may be challenging for Leader deployments in which multiple, independent nodes running the aggregate interaction (see Section 4.5) need to be coordinated. The difference between the minimum batch size and maximum batch size is in part intended to allow room for error, and allow a range of target batch sizes.

4.1.3. Batch Size Considerations

If each batch will be collected only once (i.e. when using Prio3), then batch sizes **MAY** be equal to min_batch_size without issue. In the case of fixed size queries, max_batch_size **MAY** be equal to min_batch_size. Any target batch size between min_batch_size and max_batch_size may be chosen.

If each batch may be collected more than once (i.e. when using Poplar1), then batches **SHOULD** be larger than min_batch_size, to allow for the possibility that some reports may be accepted with the first aggregation parameter, but be rejected with a subsequent aggregation parameter. Once a batch has been collected for the first time, subsequent collections of the same batch must process the same set of reports, and collections can only succeed if the number of successfully aggregated reports is at least min_batch_size (see Section 4.6.5.1 and Section 4.6.5.2). Thus, if enough reports are rejected such that fewer than min_batch_size output shares are available for aggregation, then collection of that batch may not proceed. (Note that this is not an issue for the first collection of a batch, since more reports could be combined with the uncollected reports in a subsequent collection attempt.)

The target batch size may be chosen on a deployment-specific basis, based on the expected rate of invalid reports and Sybil attack defenses (Section 7.1). When using fixed size queries, max_batch_size **SHOULD** be unset or greater than or equal to the target batch size, and the Leader **SHOULD** construct batches of the target batch size. When using time interval queries, the Collector **SHOULD** adaptively choose batch intervals based on the report upload rate so that they exceed min_batch_size by enough to allow for subsequent rejections of reports.

4.2. Task Configuration

Prior to the start of execution of the protocol, each participant must agree on the configuration for each task. A task is uniquely identified by its task ID:

```
opaque TaskID[32];
```

The task ID value **MUST** be a globally unique sequence of bytes. Each task has the following parameters associated with it:

- leader_aggregator_url: A URL relative to which the Leader's API resources can be found.

- helper_aggregator_url: A URL relative to which the Helper's API resources can be found.

- The query configuration for this task (see Section 4.1). This determines the query type for batch selection and the properties that all batches for this task must have. The party **MUST NOT** configure the task if it does not recognize the query type.

- max_batch_query_count: The maximum number of times a batch of reports may be queried by the Collector.

- task_expiration: The time up to which Clients are expected to upload to this task. The task is considered completed after this time. Aggregators **MAY** reject reports that have timestamps later than task_expiration.

- A unique identifier for the VDAF in use for the task, e.g., one of the VDAFs defined in Section 10 of [VDAF].

Note that the leader_aggregator_url and helper_aggregator_url values may include arbitrary path components.

In addition, in order to facilitate the aggregation and collect protocols, each of the Aggregators is configured with following parameters:

- collector_hpke_config: The [HPKE] configuration of the Collector (described in Section 4.4.1); see Section 6 for information about the HPKE configuration algorithms.

- vdaf_verify_key: The VDAF verification key shared by the Aggregators. This key is used in the aggregation interaction (Section 4.5). The security requirements are described in Section 7.5.1.

Finally, the Collector is configured with the HPKE secret key corresponding to collector_hpke_config.

A task's parameters are immutable for the lifetime of that task. The only way to change parameters or to rotate secret values like collector HPKE configuration or the VDAF verification key is to configure a new task.

4.3. Resource URIs

DAP is defined in terms of "resources", such as reports (Section 4.4), aggregation jobs (Section 4.5), and collection jobs (Section 4.6). Each resource has an associated URI. Resource URIs are specified by a sequence of string literals and variables. Variables are expanded into strings according to the following rules:

- Variables {leader} and {helper} are replaced with the base URL of the Leader and Helper respectively (the base URL is defined in Section 4.2).

- Variables {task-id}, {aggregation-job-id}, and {collection-job-id} are replaced with the task ID (Section 4.2), aggregation job ID (Section 4.5.1), and collection job ID (Section 4.6.1) respectively. The value **MUST** be encoded in its URL-safe, unpadded Base 64 representation as specified in Sections 5 and 3.2 of [RFC4648].

For example, given a helper URL "https://example.com/api/dap", task ID "f0 16 34 47 36 4c cf 1b c0 e3 af fc ca 68 73 c9 c3 81 f6 4a cd f9 02 06 62 f8 3f 46 c0 72 19 e7" and an aggregation job ID "95 ce da 51 e1 a9 75 23 68 b0 d9 61 f9 46 61 28" (32 and 16 byte octet strings, represented in hexadecimal), resource URI {helper}/tasks/{task-id}/aggregation_jobs/{aggregation-job-id} would be expanded into https://example.com/api/dap/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA.

4.4. Uploading Reports

Clients periodically upload reports to the Leader. Each report contains two "report shares", one for the Leader and another for the Helper. The Helper's report share is transmitted by the Leader during the aggregation interaction (see Section 4.5).

4.4.1. HPKE Configuration Request

Before the Client can upload its report to the Leader, it must know the HPKE configuration of each Aggregator. See Section 6 for information on HPKE algorithm choices.

Clients retrieve the HPKE configuration from each Aggregator by sending an HTTP GET request to {aggregator}/hpke_config. Clients **MAY** specify a query parameter task_id whose value is the task ID whose HPKE configuration they want. If the Aggregator does not recognize the task ID, then it **MUST** abort with error unrecognizedTask.

An Aggregator is free to use different HPKE configurations for each task with which it is configured. If the task ID is missing from the

Client's request, the Aggregator **MAY** abort with an error of type missingTaskID, in which case the Client **SHOULD** retry the request with a well-formed task ID included.

An Aggregator responds to well-formed requests with HTTP status code 200 OK and an HpkeConfigList value, with media type "application/dap-hpke-config-list". The HpkeConfigList structure contains one or more HpkeConfig structures in decreasing order of preference. This allows an Aggregator to support multiple HPKE configurations simultaneously.

[TODO: Allow Aggregators to return HTTP status code 403 Forbidden in deployments that use authentication to avoid leaking information about which tasks exist.]

```
HpkeConfig HpkeConfigList<1..2^16-1>;

struct {
  HpkeConfigId id;
  HpkeKemId kem_id;
  HpkeKdfId kdf_id;
  HpkeAeadId aead_id;
  HpkePublicKey public_key;
} HpkeConfig;

opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeAeadId; /* Defined in [HPKE] */
uint16 HpkeKemId;  /* Defined in [HPKE] */
uint16 HpkeKdfId;  /* Defined in [HPKE] */
```

[OPEN ISSUE: Decide whether to expand the width of the id.]

Aggregators **MUST** allocate distinct id values for each HpkeConfig in an HpkeConfigList.

The Client **MUST** abort if any of the following happen for any HPKE config request:

- the GET request did not return a valid HPKE config list;
- the HPKE config list is empty; or
- no HPKE config advertised by the Aggregator specifies a supported a KEM, KDF, or AEAD algorithm triple.

Aggregators **SHOULD** use HTTP caching to permit client-side caching of this resource [RFC5861]. Aggregators **SHOULD** favor long cache lifetimes to avoid frequent cache revalidation, e.g., on the order of days. Aggregators can control this cached lifetime with the Cache-Control header, as in this example:

```
Cache-Control: max-age=86400
```

Servers should choose a max-age value appropriate to the lifetime of their keys. Clients **SHOULD** follow the usual HTTP caching [RFC9111] semantics for HPKE configurations.

Note: Long cache lifetimes may result in Clients using stale HPKE configurations; Aggregators **SHOULD** continue to accept reports with old keys for at least twice the cache lifetime in order to avoid rejecting reports.

4.4.2. Upload Request

Clients upload reports by using an HTTP POST to {leader}/tasks/{task-id}/reports. The payload is a Report, with media type "application/dap-report", structured as follows:

```
struct {
  ReportID report_id;
  Time time;
} ReportMetadata;

struct {
  ReportMetadata report_metadata;
  opaque public_share<0..2^32-1>;
  HpkeCiphertext leader_encrypted_input_share;
  HpkeCiphertext helper_encrypted_input_share;
} Report;
```

- report_metadata is public metadata describing the report.
  - report_id is used by the Aggregators to ensure the report appears in at most one batch (see Section 4.5.1.4). The Client **MUST** generate this by generating 16 random bytes using a cryptographically secure random number generator.
  - time is the time at which the report was generated. The Client **SHOULD** round this value down to the nearest multiple of the task's time_precision in order to ensure that that the timestamp cannot be used to link a report back to the Client that generated it.
- public_share is the public share output by the VDAF sharding algorithm. Note that the public share might be empty, depending on the VDAF.
- leader_encrypted_input_share is the Leader's encrypted input share.

*helper_encrypted_input_share is the Helper's encrypted input share.

Aggregators **MAY** require Clients to authenticate when uploading reports (see Section 7.2). If it is used, HTTP client authentication **MUST** use a scheme that meets the requirements in Section 3.1.

The handling of the upload request by the Leader **MUST** be idempotent as discussed in Section 9.2.2 of [RFC9110].

To generate a report, the Client begins by sharding its measurement into input shares and the public share using the VDAF's sharding algorithm (Section 5.1 of [VDAF]), using the report ID as the nonce:

```
(public_share, input_shares) = Vdaf.shard(
    measurement, /* plaintext measurement */
    report_id,   /* nonce */
    rand,        /* randomness for sharding algorithm */
)
```

The last input comprises the randomness consumed by the sharding algorithm. The sharding randomness is a random byte string of length specified by the VDAF. The Client **MUST** generate this using a cryptographically secure random number generator.

The sharding algorithm will return two input shares. The first input share returned from the sharding algorithm is considered to be the Leader's input share, and the second input share is considered to be the Helper's input share.

The Client then wraps each input share in the following structure:

```
struct {
  Extension extensions<0..2^16-1>;
  opaque payload<0..2^32-1>;
} PlaintextInputShare;
```

Field extensions is set to the list of extensions intended to be consumed by the given Aggregator. (See Section 4.4.3.) Field payload is set to the Aggregator's input share output by the VDAF sharding algorithm.

Next, the Client encrypts each PlaintextInputShare plaintext_input_share as follows:

```
enc, payload = SealBase(pk,
  "dap-10 input share" || 0x01 || server_role,
  input_share_aad, plaintext_input_share)
```

where pk is the Aggregator's public key; 0x01 represents the Role of the sender (always the Client); server_role is the Role of the intended recipient (0x02 for the Leader and 0x03 for the Helper), plaintext_input_share is the Aggregator's PlaintextInputShare, and input_share_aad is an encoded message of type InputShareAad defined below, constructed from the same values as the corresponding fields in the report. The SealBase() function is as specified in [HPKE], Section 6.1 for the ciphersuite indicated by the HPKE configuration.

```
struct {
  TaskID task_id;
  ReportMetadata report_metadata;
  opaque public_share<0..2^32-1>;
} InputShareAad;
```

The Leader responds to well-formed requests with HTTP status code 201 Created. Malformed requests are handled as described in Section 3.2. Clients **SHOULD NOT** upload the same measurement value in more than one report if the Leader responds with HTTP status code 201 Created.

If the Leader does not recognize the task ID, then it **MUST** abort with error unrecognizedTask.

The Leader responds to requests whose Leader encrypted input share uses an out-of-date or unknown HpkeConfig.id value, indicated by HpkeCiphertext.config_id, with error of type 'outdatedConfig'. When the Client receives an 'outdatedConfig' error, it **SHOULD** invalidate any cached HpkeConfigList and retry with a freshly generated Report. If this retried upload does not succeed, the Client **SHOULD** abort and discontinue retrying.

If a report's ID matches that of a previously uploaded report, the Leader **MUST** ignore it. In addition, it **MAY** alert the Client with error reportRejected. See the implementation note in Section 4.5.1.4.

The Leader **MUST** ignore any report pertaining to a batch that has already been collected (see Section 4.5.1.4 for details). Otherwise, comparing the aggregate result to the previous aggregate result may result in a privacy violation. Note that this is also enforced by the Helper during the aggregation interaction. The Leader **MAY** also abort the upload protocol and alert the Client with error reportRejected.

The Leader **MAY** ignore any report whose timestamp is past the task's task_expiration. When it does so, it **SHOULD** also abort the upload protocol and alert the Client with error reportRejected. Client **MAY**

choose to opt out of the task if its own clock has passed task_expiration.

The Leader may need to buffer reports while waiting to aggregate them (e.g., while waiting for an aggregation parameter from the Collector; see Section 4.6). The Leader **SHOULD NOT** accept reports whose timestamps are too far in the future. Implementors **MAY** provide for some small leeway, usually no more than a few minutes, to account for clock skew. If the Leader rejects a report for this reason, it **SHOULD** abort the upload protocol and alert the Client with error reportTooEarly. In this situation, the Client **MAY** re-upload the report later on.

If the Leader's input share contains an unrecognized extension, or if two extensions have the same ExtensionType, then the Leader **MAY** abort the upload request with error "invalidMessage". Note that this behavior is not mandatory because it requires the Leader to decrypt its input share.

4.4.3. Upload Extensions

Each PlaintextInputShare carries a list of extensions that Clients use to convey additional information to the Aggregator. Some extensions might be intended for both Aggregators; others may only be intended for a specific Aggregator. (For example, a DAP deployment might use some out-of-band mechanism for an Aggregator to verify that reports come from authenticated Clients. It will likely be useful to bind the extension to the input share via HPKE encryption.)

Each extension is a tag-length encoded value of the following form:

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} Extension;

enum {
  TBD(0),
  (65535)
} ExtensionType;
```

Field "extension_type" indicates the type of extension, and "extension_data" contains information specific to the extension.

Extensions are mandatory-to-implement: If an Aggregator receives a report containing an extension it does not recognize, then it **MUST** reject the report. (See Section 4.5.1.4 for details.)

4.5. Verifying and Aggregating Reports

Once a set of Clients have uploaded their reports to the Leader, the Leader can begin the process of validating and aggregating them with the Helper. To enable the system to handle large batches of reports, this process can be parallelized across many "aggregation jobs" in which small subsets of the reports are processed independently. Each aggregation job is associated with exactly one DAP task, but a task can have many aggregation jobs.

The primary objective of an aggregation job is to run the VDAF preparation process described in [VDAF], Section 5.2 for each report in the job. Preparation has two purposes:

1. To "refine" the input shares into "output shares" that have the desired aggregatable form. For some VDAFs, like Prio3, the mapping from input to output shares is some fixed, linear operation, but in general the mapping is controlled dynamically by the Collector and can be non-linear. In Poplar1, for example, the refinement process involves a sequence of "candidate prefixes" and the output consists of a sequence of zeros and ones, each indicating whether the corresponding candidate is a prefix of the measurement from which the input shares were generated.

2. To verify that the output shares, when combined, correspond to a valid, refined measurement, where validity is determined by the VDAF itself. For example, the Prio3Sum variant of Prio3 (Section 7.4.2 of [VDAF]) requires that the output shares sum up to an integer in a specific range; for Poplar1, the output shares are required to sum up to a vector that is non-zero in at most one position.

In general, refinement and verification are not distinct computations, since for some VDAFs, verification may only be achieved implicitly as a result of the refinement process. We instead think of these as properties of the output shares themselves: if preparation succeeds, then the resulting output shares are guaranteed to combine into a valid, refined measurement.

VDAF preparation is mapped onto an aggregation job as illustrated in Figure 2. The protocol is comprised of a sequence of HTTP requests from the Leader to the Helper, the first of which includes the aggregation parameter, the Helper's report share for each report in the job, and for each report the initialization step for preparation. The Helper's response, along with each subsequent request and response, carries the remaining messages exchanged during preparation.

Figure 2: Overview of the DAP aggregation interaction.

The number of steps, and the type of the responses, depends on the VDAF. The message structures and processing rules are specified in the following subsections.

In general, reports cannot be aggregated until the Collector specifies an aggregation parameter. However, in some situations it is possible to begin aggregation as soon as reports arrive. For example, Prio3 has just one valid aggregation parameter (the empty string). And there are use cases for Poplar1 in which aggregation can begin immediately (i.e., those for which the candidate prefixes/ strings are fixed in advance).

An aggregation job can be thought of as having three phases:

- Initialization: Begin the aggregation flow by disseminating report shares and initializing the VDAF prep state for each report.

- Continuation: Continue the aggregation flow by exchanging prep shares and messages until preparation completes or an error occurs.

- Completion: Finish the aggregate flow, yielding an output share corresponding to each report share in the aggregation job.

These phases are described in the following subsections.

4.5.1. Aggregate Initialization

The Leader begins an aggregation job by choosing a set of candidate reports that pertain to the same DAP task and a job ID which **MUST** be unique within the scope of the task. The job ID is a 16-byte value, structured as follows:

```
opaque AggregationJobID[16];
```

The Leader can run this process for many sets of candidate reports in parallel as needed. After choosing a set of candidates, the Leader begins aggregation by splitting each report into report shares, one for each Aggregator. The Leader and Helper then run the aggregate initialization flow to accomplish two tasks:

1. Recover and determine which input report shares are valid.

2. For each valid report share, initialize the VDAF preparation process (see Section 5.2 of [VDAF]).

The Leader and Helper initialization behavior is detailed below.

4.5.1.1. Leader Initialization

The Leader begins the aggregate initialization phase with the set of candidate reports as follows:

1. Generate a fresh AggregationJobID.

2. Decrypt the input share for each report share as described in Section 4.5.1.3.

3. Check that the resulting input share is valid as described in Section 4.5.1.4.

If any step invalidates the report, the Leader rejects the report and removes it from the set of candidate reports.

Next, for each report the Leader executes the following procedure:

```
(state, outbound) = Vdaf.ping_pong_leader_init(
    vdaf_verify_key,
    agg_param,
    report_id,
    public_share,
    plaintext_input_share.payload)
```

where:

- vdaf_verify_key is the VDAF verification key for the task
- agg_param is the VDAF aggregation parameter provided by the Collector (see Section 4.6)
- report_id is the report ID, used as the nonce for VDAF sharding
- public_share is the report's public share
- plaintext_input_share is the Leader's PlaintextInputShare

The methods are defined in Section 5.8 of [VDAF]. This process determines the initial per-report state, as well as the initial outbound message to send to the Helper. If state is of type Rejected, then the report is rejected and removed from the set of candidate reports, and no message is sent to the Helper.

If state is of type Continued, then the Leader constructs a PrepareInit message structured as follows:

```
struct {
  ReportMetadata report_metadata;
  opaque public_share<0..2^32-1>;
  HpkeCiphertext encrypted_input_share;
} ReportShare;

struct {
  ReportShare report_share;
  opaque payload<0..2^32-1>;
} PrepareInit;
```

Each of these messages is constructed as follows:

- report_share.report_metadata is the report's metadata.
- report_share.public_share is the report's public share.

*report_share.encrypted_input_share is the intended recipient's (i.e. Helper's) encrypted input share.

*payload is set to the outbound message computed by the previous step.

It is not possible for state to be of type Finished during Leader initialization.

Once all the report shares have been initialized, the Leader creates an AggregationJobInitReq message structured as follows:

```
struct {
  QueryType query_type;
  select (PartialBatchSelector.query_type) {
    case time_interval: Empty;
    case fixed_size: BatchID batch_id;
  };
} PartialBatchSelector;

struct {
  opaque agg_param<0..2^32-1>;
  PartialBatchSelector part_batch_selector;
  PrepareInit prepare_inits<1..2^32-1>;
} AggregationJobInitReq;
```

[[OPEN ISSUE: Consider sending report shares separately (in parallel) to the aggregate instructions. Right now, aggregation parameters and the corresponding report shares are sent at the same time, but this may not be strictly necessary.]]

This message consists of:

*agg_param: The VDAF aggregation parameter.

*part_batch_selector: The "partial batch selector" used by the Aggregators to determine how to aggregate each report:

-For fixed_size tasks, the Leader specifies a "batch ID" that determines the batch to which each report for this aggregation job belongs.

[OPEN ISSUE: For fixed_size tasks, the Leader is in complete control over which batch a report is included in. For time_interval tasks, the Client has some control, since the timestamp determines which batch window it falls in. Is this desirable from a privacy perspective? If not, it might be simpler to drop the timestamp altogether and have the agg init request specify the batch window instead.]

The indicated query type **MUST** match the task's query type. Otherwise, the Helper **MUST** abort with error invalidMessage.

This field is called the "partial" batch selector because, depending on the query type, it may not determine a batch. In particular, if the query type is time_interval, the batch is not determined until the Collector's query is issued (see Section 4.1).

* prepare_inits: the sequence of PrepareInit messages constructed in the previous step.

Finally, the Leader sends a PUT request to {helper}/tasks/{task-id}/aggregation_jobs/{aggregation-job-id}. The payload is set to AggregationJobInitReq and the media type is set to "application/dap-aggregation-job-init-req".

The Leader **MUST** authenticate its requests to the Helper using a scheme that meets the requirements in Section 3.1.

The Helper's response will be an AggregationJobResp message (see Section 4.5.1.2. The response's prepare_resps must include exactly the same report IDs in the same order as the Leader's AggregationJobInitReq. Otherwise, the Leader **MUST** abort the aggregation job.

[[OPEN ISSUE: consider relaxing this ordering constraint. See issue#217.]]

Otherwise, the Leader proceeds as follows with each report:

1. If the inbound prep response has type "continue", then the Leader computes

   ```
   (state, outbound) = Vdaf.ping_pong_leader_continued(agg_param,
                                                       prev_state,
                                                       inbound)
   ```

   where:

   * agg_param is the VDAF aggregation parameter provided by the Collector (see Section 4.6)

   * prev_state is the state computed earlier by calling Vdaf.ping_pong_leader_init or Vdaf.ping_pong_leader_continued

   * inbound is the message payload in the PrepareResp

If outbound != None, then the Leader stores state and outbound and proceeds to Section 4.5.2.1. If outbound == None, then the preparation process is complete: either state == Rejected(), in which case the Leader rejects the report and removes it from the candidate set; or state == Finished(out_share), in which case preparation is complete and the Leader stores the output share for use in the collection interaction Section 4.6.

2. Else if the type is "reject", then the Leader rejects the report and removes it from the candidate set. The Leader **MUST NOT** include the report in a subsequent aggregation job, unless the error is report_too_early, in which case the Leader **MAY** include the report in a subsequent aggregation job.

3. Else the type is invalid, in which case the Leader **MUST** abort the aggregation job.

(Note: Since VDAF preparation completes in a constant number of rounds, it will never be the case that some reports are completed and others are not.)

4.5.1.2. Helper Initialization

The Helper begins an aggregation job when it receives an AggregationJobInitReq message from the Leader. For each PrepareInit conveyed by this message, the Helper attempts to initialize VDAF preparation (see Section 5.1 of [VDAF]) just as the Leader does. If successful, it includes the result in its response that the Leader will use to continue preparing the report.

To begin this process, the Helper checks if it recognizes the task ID. If not, then it **MUST** abort with error unrecognizedTask.

Next, the Helper checks that the report IDs in AggregationJobInitReq.prepare_inits are all distinct. If two preparation initialization messages have the same report ID, then the Helper **MUST** abort with error invalidMessage.

The Helper is now ready to process each report share into an outbound prepare step to return to the Leader. The responses will be structured as follows:

```
enum {
  continue(0),
  finished(1)
  reject(2),
  (255)
} PrepareRespState;

enum {
  batch_collected(0),
  report_replayed(1),
  report_dropped(2),
  hpke_unknown_config_id(3),
  hpke_decrypt_error(4),
  vdaf_prep_error(5),
  batch_saturated(6),
  task_expired(7),
  invalid_message(8),
  report_too_early(9),
  (255)
} PrepareError;

struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state;
  select (PrepareResp.prepare_resp_state) {
    case continue: opaque payload<0..2^32-1>;
    case finished: Empty;
    case reject:   PrepareError prepare_error;
  };
} PrepareResp;
```

First the Helper preprocesses each report as follows:

1. Decrypt the input share for each report share as described in Section 4.5.1.3.

2. Check that the resulting input share is valid as described in Section 4.5.1.4.

For any report that was rejected, the Helper sets the outbound preparation response to

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = reject;
  PrepareError prepare_error;
} PrepareResp;
```

where report_id is the report ID and prepare_error is the indicated error. For all other reports it initializes the VDAF prep state as

follows (let inbound denote the payload of the prep step sent by the Leader):

```
(state, outbound) = Vdaf.ping_pong_helper_init(vdaf_verify_key,
                                               agg_param,
                                               report_id,
                                               public_share,
                                               plaintext_input_share.pay
```

where:

- vdaf_verify_key is the VDAF verification key for the task
- agg_param is the VDAF aggregation parameter sent in the AggregationJobInitReq
- report_id is the report ID
- public_share is the report's public share
- plaintext_input_share is the Helper's PlaintextInputShare

This procedure determines the initial per-report state, as well as the initial outbound message to send in response to the Leader. If state is of type Rejected, then the Helper responds with

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = reject;
  PrepareError prepare_error = vdaf_prep_error;
} PrepareResp;
```

Otherwise the Helper responds with

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = continue;
  opaque payload<0..2^32-1> = outbound;
} PrepareResp;
```

Finally, the Helper creates an AggregationJobResp to send to the Leader. This message is structured as follows:

```
struct {
  PrepareResp prepare_resps<1..2^32-1>;
} AggregationJobResp;
```

where prepare_resps are the outbound prep steps computed in the previous step. The order **MUST** match AggregationJobInitReq.prepare_inits.

The Helper responds to the Leader with HTTP status code 201 Created and a body consisting of the AggregationJobResp, with media type "application/dap-aggregation-job-resp".

Changing an aggregation job's parameters is illegal, so further requests to PUT /tasks/{tasks}/aggregation_jobs/{aggregation-job-id} for the same aggregation-job-id but with a different AggregationJobInitReq in the body **MUST** fail with an HTTP client error status code.

Additionally, it is not possible to rewind or reset the state of an aggregation job. Once an aggregation job has been continued at least once (see Section 4.5.2), further requests to initialize that aggregation job **MUST** fail with an HTTP client error status code.

Finally, if state == Continued(prep_state), then the Helper stores state to prepare for the next continuation step (Section 4.5.2.2). Otherwise, if state == Finished(out_share), then the Helper stores out_share for use in the collection interaction (Section 4.6).

4.5.1.3. Input Share Decryption

Each report share has a corresponding task ID, report metadata (report ID and, timestamp), public share, and the Aggregator's encrypted input share. Let task_id, report_metadata, public_share, and encrypted_input_share denote these values, respectively. Given these values, an Aggregator decrypts the input share as follows. First, it constructs an InputShareAad message from task_id, report_metadata, and public_share. Let this be denoted by input_share_aad. Then, the Aggregator looks up the HPKE config and corresponding secret key indicated by encrypted_input_share.config_id and attempts decryption of the payload with the following procedure:

```
plaintext_input_share = OpenBase(encrypted_input_share.enc, sk,
  "dap-10 input share" || 0x01 || server_role,
  input_share_aad, encrypted_input_share.payload)
```

where sk is the HPKE secret key, 0x01 represents the Role of the sender (always the Client), and server_role is the Role of the recipient Aggregator (0x02 for the Leader and 0x03 for the Helper). The OpenBase() function is as specified in [HPKE], Section 6.1 for the ciphersuite indicated by the HPKE configuration. If decryption fails, the Aggregator marks the report share as invalid with the error hpke_decrypt_error. Otherwise, the Aggregator outputs the resulting PlaintextInputShare plaintext_input_share.

4.5.1.4. Input Share Validation

Validating an input share will either succeed or fail. In the case of failure, the input share is marked as invalid with a corresponding PrepareError.

Before beginning the preparation step, Aggregators are required to perform the following checks:

1. Check that the input share can be decoded as specified by the VDAF. If not, the input share **MUST** be marked as invalid with the error invalid_message.

2. Check if the report is too far into the future. Implementors can provide for some small leeway, usually no more than a few minutes, to account for clock skew. If a report is rejected for this reason, the Aggregator **SHOULD** mark the input share as invalid with the error report_too_early.

3. Check if the report's timestamp has passed the task's task_expiration time. If so, the Aggregator **MAY** mark the input share as invalid with the error task_expired.

4. Check if the PlaintextInputShare contains unrecognized extensions. If so, the Aggregator **MUST** mark the input share as invalid with error invalid_message.

5. Check if the ExtensionType of any two extensions in PlaintextInputShare are the same. If so, the Aggregator **MUST** mark the input share as invalid with error invalid_message.

6. Check if the report may still be aggregated with the current aggregation parameter. This can be done by looking up all aggregation parameters previously used for this report and calling

   Vdaf.is_valid(current_agg_param, previous_agg_params)

   If this returns false, the input share **MUST** be marked as invalid with the error report_replayed.

   *Implementation note: To detect replay attacks, each Aggregator is required to keep track of the set of reports it has processed for a given task. Because honest Clients choose the report ID at random, it is sufficient to store the set of IDs of processed reports. However, implementations may find it helpful to track additional information, like the timestamp, so that the storage used for anti-replay can be sharded efficiently.

7. If the report pertains to a batch that was previously collected, then make sure the report was already included in all previous collections for the batch. If not, the input share **MUST** be marked as invalid with error batch_collected. This prevents Collectors from learning anything about small numbers of reports that are uploaded between two collections of a batch.

    * Implementation note: The Leader considers a batch to be collected once it has completed a collection job for a CollectionReq message from the Collector; the Helper considers a batch to be collected once it has responded to an AggregateShareReq message from the Leader. A batch is determined by query (Section 4.1) conveyed in these messages. Queries must satisfy the criteria covered in Section 4.6.5. These criteria are meant to restrict queries in a way make it easy to determine wither a report pertains to a batch that was collected.

      [TODO: If a section to clarify report and batch states is added this can be removed. See Issue #384]

8. Depending on the query type for the task, additional checks may be applicable:

    * For fixed_size tasks, the Aggregators need to ensure that each batch is roughly the same size. If the number of reports aggregated for the current batch exceeds the maximum batch size (per the task configuration), the Aggregator **MAY** mark the input share as invalid with the error batch_saturated. Note that this behavior is not strictly enforced here but during the collect interaction. (See Section 4.6.5.) If maximum batch size is not provided, then Aggregators only need to ensure the current batch exceeds the minimum batch size (per the task configuration). If both checks succeed, the input share is not marked as invalid.

9. Finally, if an Aggregator cannot determine if an input share is valid, it **MUST** mark the input share as invalid with error report_dropped. For example, if the Aggregator has evicted the state required to perform the check from long-term storage. (See Section 5.4.1 for details.)

If all of the above checks succeed, the input share is not marked as invalid.

4.5.2. Aggregate Continuation

In the continuation phase, the Leader drives the VDAF preparation of each report in the candidate report set until the underlying VDAF

moves into a terminal state, yielding an output share for both Aggregators or a rejection.

Whether this phase is reached depends on the VDAF: for 1-round VDAFs, like Prio3, processing has already completed. Continuation is required for VDAFs that require more than one round.

4.5.2.1. Leader Continuation

The Leader begins each step of aggregation continuation with a prep state object state and an outbound message outbound for each report in the candidate set.

The Leader advances its aggregation job to the next step (step 1 if this is the first continuation after initialization). Then it instructs the Helper to advance the aggregation job to the step the Leader has just reached. For each report the Leader constructs a preparation continuation message:

```
struct {
  ReportID report_id;
  opaque payload<0..2^32-1>;
} PrepareContinue;
```

where report_id is the report ID associated with state and outbound, and payload is set to the outbound message.

Next, the Leader sends a POST request to the aggregation job URI used during initialization (see Section 4.5.1.1) with media type "application/dap-aggregation-job-continue-req" and body structured as:

```
struct {
  uint16 step;
  PrepareContinue prepare_continues<1..2^32-1>;
} AggregationJobContinueReq;
```

The step field is the step of DAP aggregation that the Leader just reached and wants the Helper to advance to. The prepare_continues field is the sequence of preparation continuation messages constructed in the previous step. The PrepareContinues **MUST** be in the same order as the previous aggregate request.

The Leader **MUST** authenticate its requests to the Helper using a scheme that meets the requirements in Section 3.1.

The Helper's response will be an AggregationJobResp message (see Section 4.5.1.2). The response's prepare_resps must include exactly the same report IDs in the same order as the Leader's

AggregationJobContinueReq. Otherwise, the Leader **MUST** abort the aggregation job.

[[OPEN ISSUE: consider relaxing this ordering constraint. See issue#217.]]

Otherwise, the Leader proceeds as follows with each report:

1. If the inbound prep response type is "continue" and the state is Continued(prep_state), then the Leader computes

   ```
   (state, outbound) = Vdaf.ping_pong_leader_continued(agg_param,
                                                       state,
                                                       inbound)
   ```

   where inbound is the message payload. If outbound != None, then the Leader stores state and outbound and proceeds to another continuation step. If outbound == None, then the preparation process is complete: either state == Rejected(), in which case the Leader rejects the report and removes it from the candidate set; or state == Finished(out_share), in which case preparation is complete and the Leader stores the output share for use in the collection interaction Section 4.6.

2. Else if the type is "finished" and state == Finished(out_share), then preparation is complete and the Leader stores the output share for use in the collection flow (Section 4.6).

3. Else if the type is "reject", then the Leader rejects the report and removes it from the candidate set.

4. Else the type is invalid, in which case the Leader **MUST** abort the aggregation job.

4.5.2.2. Helper Continuation

The Helper begins each step of continuation with a sequence of state objects, which will be Continued(prep_state), one for each report in the candidate set.

The Helper awaits an HTTP POST request to the aggregation job URI from the Leader, the body of which is an AggregationJobContinueReq as specified in Section 4.5.2.1.

Next, it checks that it recognizes the task ID. If not, then it **MUST** abort with error unrecognizedTask.

Next, it checks if it recognizes the indicated aggregation job ID. If not, it **MUST** abort with error unrecognizedAggregationJob.

Next, the Helper checks that:

1. the report IDs are all distinct

2. each report ID corresponds to one of the state objects

3. AggregationJobContinueReq.step is not equal to 0

If any of these checks fail, then the Helper **MUST** abort with error invalidMessage. Additionally, if any prep step appears out of order relative to the previous request, then the Helper **MAY** abort with error invalidMessage. (Note that a report may be missing, in which case the Helper should assume the Leader rejected it.)

[OPEN ISSUE: Issue 438: It may be useful for the Leader to explicitly signal rejection.]

Next, the Helper checks if the continuation step indicated by the request is correct. (For the first AggregationJobContinueReq the value should be 1; for the second the value should be 2; and so on.) If the Leader is one step behind (e.g., the Leader has resent the previous HTTP request), then the Helper **MAY** attempt to recover by re-sending the previous AggregationJobResp. In this case it **SHOULD** verify that the contents of the AggregationJobContinueReq are identical to the previous message (see Section 4.5.2.3). Otherwise, if the step is incorrect, the Helper **MUST** abort with error stepMismatch.

The Helper is now ready to continue preparation for each report. Let inbound denote the payload of the prep step. The Helper computes the following:

```
(state, outbound) = Vdaf.ping_pong_helper_continued(agg_param,
                                                    state,
                                                    inbound)
```

If state == Rejected(), then the Helper's response is

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = reject;
  PrepareError prepare_error = vdaf_prep_error;
} PrepareResp;
```

Otherwise, if outbound != None, then the Helper's response is

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = continue;
  opaque payload<0..2^32-1> = outbound;
} PrepareResp;
```

Otherwise, if outbound == None, then the Helper's response is

```
struct {
  ReportID report_id;
  PrepareRespState prepare_resp_state = finished;
} PrepareResp;
```

Next, the Helper constructs an AggregationJobResp message (Section 4.5.1.2) with each prep step. The order of the prep steps **MUST** match the Leader's request. It responds to the Leader with HTTP status 200 OK, media type application/dap-aggregation-job-resp, and a body consisting of the AggregationJobResp.

Finally, if state == Continued(prep_state), then the Helper stores state to prepare for the next continuation step (Section 4.5.2.2). Otherwise, if state == Finished(out_share), then the Helper stores out_share for use in the collection interaction (Section 4.6).

If for whatever reason the Leader must abandon the aggregation job, it **SHOULD** send an HTTP DELETE request to the aggregation job URI so that the Helper knows it can clean up its state.

4.5.2.3. Recovering from Aggregation Step Skew

AggregationJobContinueReq messages contain a step field, allowing Aggregators to ensure that their peer is on an expected step of the DAP aggregation protocol. In particular, the intent is to allow recovery from a scenario where the Helper successfully advances from step n to n+1, but its AggregationJobResp response to the Leader gets dropped due to something like a transient network failure. The Leader could then resend the request to have the Helper advance to step n+1 and the Helper should be able to retransmit the AggregationJobContinueReq that was previously dropped. To make that kind of recovery possible, Aggregator implementations **SHOULD** checkpoint the most recent step's prep state and messages to durable storage such that the Leader can re-construct continuation requests and the Helper can re-construct continuation responses as needed.

When implementing an aggregation step skew recovery strategy, the Helper **SHOULD** ensure that the Leader's AggregationJobContinueReq message did not change when it was re-sent (i.e., the two messages must be identical). This prevents the Leader from re-winding an aggregation job and re-running an aggregation step with different parameters.

[[OPEN ISSUE: Allowing the Leader to "rewind" aggregation job state of the Helper may allow an attack on privacy. For instance, if the VDAF verification key changes, the prep shares in the Helper's response would change even if the consistency check is made. Security analysis is required. See #401.]]

One way the Helper could address this would be to store a digest of the Leader's request, indexed by aggregation job ID and step, and refuse to service a request for a given aggregation step unless it matches the previously seen request (if any).

4.6. Collecting Results

In this phase, the Collector requests aggregate shares from each Aggregator and then locally combines them to yield a single aggregate result. In particular, the Collector issues a query to the Leader (Section 4.1), which the Aggregators use to select a batch of reports to aggregate. Each Aggregator emits an aggregate share encrypted to the Collector so that it can decrypt and combine them to yield the aggregate result. This entire process is composed of two interactions:

1. Collect request and response between the Collector and Leader, specified in Section 4.6.1

2. Aggregate share request and response between the Leader and the Helper, specified in Section 4.6.2

Once complete, the Collector computes the final aggregate result as specified in Section 4.6.3.

This overall process is referred to as a "collection job".

4.6.1. Collection Job Initialization

First, the Collector chooses a collection job ID:

```
opaque CollectionJobID[16];
```

This ID value **MUST** be unique within the scope of the corresponding DAP task.

To initiate the collection job, the collector issues a PUT request to {leader}/tasks/{task-id}/collection_jobs/{collection-job-id}. The body of the request has media type "application/dap-collect-req", and it is structured as follows:

```
struct {
  Query query;
  opaque agg_param<0..2^32-1>; /* VDAF aggregation parameter */
} CollectionReq;
```

The named parameters are:

- query, the Collector's query. The indicated query type **MUST** match the task's query type. Otherwise, the Leader **MUST** abort with error "invalidMessage".

- agg_param, an aggregation parameter for the VDAF being executed. This is the same value as in AggregationJobInitReq (see Section 4.5.1.1).

Collectors **MUST** authenticate their requests to Leaders using a scheme that meets the requirements in Section 3.1.

Depending on the VDAF scheme and how the Leader is configured, the Leader and Helper may already have prepared a sufficient number of reports satisfying the query and be ready to return the aggregate shares right away. However, this is not always the case. In fact, for some VDAFs, it is not be possible to begin running aggregation jobs (Section 4.5) until the Collector initiates a collection job. This is because, in general, the aggregation parameter is not known until this point. In certain situations it is possible to predict the aggregation parameter in advance. For example, for Prio3 the only valid aggregation parameter is the empty string. For these reasons, the collection job is handled asynchronously.

Upon receipt of a CollectionReq, the Leader begins by checking that it recognizes the task ID in the request path. If not, it **MUST** abort with error unrecognizedTask.

The Leader **MAY** further validate the request according to the requirements in Section 4.6.5 and abort with the indicated error, though some conditions such as the number of valid reports may not be verifiable while handling the CollectionReq message, and the batch will have to be re-validated later on regardless.

If the Leader finds the CollectionReq to be valid, it immediately responds with HTTP status 201.

The Leader then begins working with the Helper to aggregate the reports satisfying the query (or continues this process, depending on the VDAF) as described in Section 4.5.

Changing a collection job's parameters is illegal, so further requests to PUT /tasks/{tasks}/collection_jobs/{collection-job-id}

for the same collection-job-id but with a different CollectionReq in the body **MUST** fail with an HTTP client error status code.

After receiving the response to its CollectionReq, the Collector makes an HTTP GET request to the collection job URI to check on the status of the collect job and eventually obtain the result. If the collection job is not finished yet, the Leader responds with HTTP status 202 Accepted. The response **MAY** include a Retry-After header field to suggest a polling interval to the Collector.

Asynchronously from any request from the Collector, the Leader attempts to run the collection job. It first checks whether it can construct a batch for the collection job by applying the requirements in Section 4.6.5. If so, then the Leader obtains the Helper's aggregate share following the aggregate-share request flow described in Section 4.6.2. If not, it either aborts the collection job or tries again later, depending on which requirement in Section 4.6.5 was not met. If the Leader has a pending aggregation job that overlaps with the batch and aggregation parameter for the collection job, the Leader **MUST** first complete the aggregation job before proceeding and requesting an aggregate share from the Helper. This avoids a race condition between aggregation and collection jobs that can yield trivial batch mismatch errors.

Once both aggregate shares are successfully obtained, the Leader responds to subsequent HTTP GET requests to the collection job with HTTP status code 200 OK and a body consisting of a Collection:

```
struct {
  PartialBatchSelector part_batch_selector;
  uint64 report_count;
  Interval interval;
  HpkeCiphertext leader_encrypted_agg_share;
  HpkeCiphertext helper_encrypted_agg_share;
} Collection;
```

The body's media type is "application/dap-collection". The Collection structure includes the following:

*part_batch_selector: Information used to bind the aggregate result to the query. For fixed_size tasks, this includes the batch ID assigned to the batch by the Leader. The indicated query type **MUST** match the task's query type.

[OPEN ISSUE: What should the Collector do if the query type doesn't match?]

*report_count: The number of reports included in the batch.

- *interval: The smallest interval of time that contains the timestamps of all reports included in the batch, such that the interval's start and duration are both multiples of the task's time_precision parameter. Note that in the case of a time_interval type query (see Section 4.1), this interval can be smaller than the one in the corresponding CollectionReq.query.

- *leader_encrypted_agg_share: The Leader's aggregate share, encrypted to the Collector.

- *helper_encrypted_agg_share: The Helper's aggregate share, encrypted to the Collector.

If obtaining aggregate shares fails, then the Leader responds to subsequent HTTP GET requests to the collection job with an HTTP error status and a problem document as described in Section 3.2.

The Leader **MAY** respond with HTTP status 204 No Content to requests to a collection job if the results have been deleted.

The Collector can send an HTTP DELETE request to the collection job, which indicates to the Leader that it can abandon the collection job and discard all state related to it.

4.6.2. Obtaining Aggregate Shares

The Leader must obtain the Helper's encrypted aggregate share before it can complete a collection job. To do this, the Leader first computes a checksum over the reports included in the batch. The checksum is computed by taking the SHA256 [SHS] hash of each report ID from the Client reports included in the aggregation, then combining the hash values with a bitwise-XOR operation.

Then the Leader sends a POST request to {helper}/tasks/{task-id}/aggregate_shares with the following message:

```
struct {
  QueryType query_type;
  select (BatchSelector.query_type) {
    case time_interval: Interval batch_interval;
    case fixed_size: BatchID batch_id;
  };
} BatchSelector;

struct {
  BatchSelector batch_selector;
  opaque agg_param<0..2^32-1>;
  uint64 report_count;
  opaque checksum[32];
} AggregateShareReq;
```

The media type of the request is "application/dap-aggregate-share-req". The message contains the following parameters:

* batch_selector: The "batch selector", which encodes parameters used to determine the batch being aggregated. The value depends on the query type for the task:

  - For time_interval tasks, the request specifies the batch interval.

  - For fixed_size tasks, the request specifies the batch ID.

  The indicated query type **MUST** match the task's query type. Otherwise, the Helper **MUST** abort with "invalidMessage".

* agg_param: The opaque aggregation parameter for the VDAF being executed. This value **MUST** match the AggregationJobInitReq message for each aggregation job used to compute the aggregate shares (see Section 4.5.1.1) and the aggregation parameter indicated by the Collector in the CollectionReq message (see Section 4.6.1).

* report_count: The number number of reports included in the batch.

* checksum: The batch checksum.

Leaders **MUST** authenticate their requests to Helpers using a scheme that meets the requirements in Section 3.1.

To handle the Leader's request, the Helper first ensures that it recognizes the task ID in the request path. If not, it **MUST** abort with error unrecognizedTask. The Helper then verifies that the request meets the requirements for batch parameters following the procedure in Section 4.6.5.

Next, it computes a checksum based on the reports that satisfy the query, and checks that the report_count and checksum included in the request match its computed values. If not, then it **MUST** abort with an error of type "batchMismatch".

Next, it computes the aggregate share agg_share corresponding to the set of output shares, denoted out_shares, for the batch interval, as follows:

```
agg_share = Vdaf.aggregate(agg_param, out_shares)
```

Implementation note: For most VDAFs, it is possible to aggregate output shares as they arrive rather than wait until the batch is collected. To do so however, it is necessary to enforce the batch parameters as described in Section 4.6.5 so that the Aggregator knows which aggregate share to update.

The Helper then encrypts agg_share under the Collector's HPKE public key as described in Section 4.6.4, yielding encrypted_agg_share. Encryption prevents the Leader from learning the actual result, as it only has its own aggregate share and cannot compute the Helper's.

The Helper responds to the Leader with HTTP status code 200 OK and a body consisting of an AggregateShare, with media type "application/dap-aggregate-share":

```
struct {
  HpkeCiphertext encrypted_aggregate_share;
} AggregateShare;
```

encrypted_aggregate_share.config_id is set to the Collector's HPKE config ID. encrypted_aggregate_share.enc is set to the encapsulated HPKE context enc computed above and encrypted_aggregate_share.ciphertext is the ciphertext encrypted_agg_share computed above.

The Helper's handling of this request **MUST** be idempotent. That is, if multiple identical, valid AggregateShareReqs are received, they should all yield the same response while only consuming one unit of the task's max_batch_query_count (see Section 4.6.5).

After receiving the Helper's response, the Leader uses the HpkeCiphertext to finalize a collection job (see Section 4.6.3).

Once an AggregateShareReq has been issued for the batch determined by a given query, it is an error for the Leader to issue any more aggregation jobs for additional reports that satisfy the query. These reports will be rejected by the Helper as described in Section 4.5.1.4.

Before completing the collection job, the Leader also computes its own aggregate share agg_share by aggregating all of the prepared output shares that fall within the batch interval. Finally, it encrypts its aggregate share under the Collector's HPKE public key as described in Section 4.6.4.

4.6.3. Collection Job Finalization

Once the Collector has received a collection job from the Leader, it can decrypt the aggregate shares and produce an aggregate result. The Collector decrypts each aggregate share as described in Section 4.6.4. Once the Collector successfully decrypts all aggregate shares, it unshards the aggregate shares into an aggregate result using the VDAF's unshard algorithm. In particular, let leader_agg_share denote the Leader's aggregate share, helper_agg_share denote the Helper's aggregate share, let report_count denote the report count sent by the Leader, and let

agg_param be the opaque aggregation parameter. The final aggregate result is computed as follows:

```
agg_result = Vdaf.unshard(agg_param,
                          [leader_agg_share, helper_agg_share],
                          report_count)
```

4.6.4. Aggregate Share Encryption

Encrypting an aggregate share agg_share for a given AggregateShareReq is done as follows:

```
enc, payload = SealBase(pk, "dap-10 aggregate share" || server_role || 0
  agg_share_aad, agg_share)
```

where pk is the HPKE public key encoded by the Collector's HPKE key, server_role is the Role of the encrypting server (0x02 for the Leader and 0x03 for a Helper), 0x00 represents the Role of the recipient (always the Collector), and agg_share_aad is a value of type AggregateShareAad. The SealBase() function is as specified in [HPKE], Section 6.1 for the ciphersuite indicated by the HPKE configuration.

```
struct {
  TaskID task_id;
  opaque agg_param<0..2^32-1>;
  BatchSelector batch_selector;
} AggregateShareAad;
```

- task_id is the ID of the task the aggregate share was computed in.
- agg_param is the aggregation parameter used to compute the aggregate share.
- batch_selector is the is the batch selector from the AggregateShareReq (for the Helper) or the batch selector computed from the Collector's query (for the Leader).

The Collector decrypts these aggregate shares using the opposite process. Specifically, given an encrypted input share, denoted enc_share, for a given batch selector, decryption works as follows:

```
agg_share = OpenBase(enc_share.enc, sk, "dap-10 aggregate share" ||
  server_role || 0x00, agg_share_aad, enc_share.payload)
```

where sk is the HPKE secret key, server_role is the Role of the server that sent the aggregate share (0x02 for the Leader and 0x03 for the Helper), 0x00 represents the Role of the recipient (always the Collector), and agg_share_aad is an AggregateShareAad message

constructed from the task ID and the aggregation parameter in the collect request, and a batch selector. The value of the batch selector used in agg_share_aad is computed by the Collector from its query and the response to its query as follows:

- For time_interval tasks, the batch selector is the batch interval specified in the query.

- For fixed_size tasks, the batch selector is the batch ID assigned sent in the response.

The OpenBase() function is as specified in [HPKE], Section 6.1 for the ciphersuite indicated by the HPKE configuration.

4.6.5. Batch Validation

Before a Leader runs a collection job or a Helper responds to an AggregateShareReq, it must first check that the job or request does not violate the parameters associated with the DAP task. It does so as described here. Where we say that an Aggregator **MUST** abort with some error, then:

- Leaders should respond to subsequent HTTP GET requests to the collection job with the indicated error.

- Helpers should respond to the AggregateShareReq with the indicated error.

First the Aggregator checks that the batch respects any "boundaries" determined by the query type. These are described in the subsections below. If the boundary check fails, then the Aggregator **MUST** abort with an error of type "batchInvalid".

Next, the Aggregator checks that batch contains a valid number of reports, as determined by the query type. If the size check fails, then Helpers **MUST** abort with an error of type "invalidBatchSize". Leaders **SHOULD** wait for more reports to be validated and try the collection job again later.

Next, the Aggregator checks that the batch has not been queried too many times. This is determined by the maximum number of times a batch can be queried, max_batch_query_count. If the batch has been queried with more than max_batch_query_count distinct aggregation parameters, the Aggregator **MUST** abort with error of type "batchQueriedTooManyTimes".

Finally, the Aggregator checks that the batch does not contain a report that was included in any previous batch. If this batch overlap check fails, then the Aggregator **MUST** abort with error of type "batchOverlap". For time_interval tasks, it is sufficient (but

not necessary) to check that the batch interval does not overlap with the batch interval of any previous query. If this batch interval check fails, then the Aggregator **MAY** abort with error of type "batchOverlap".

[[OPEN ISSUE: #195 tracks how we might relax this constraint to allow for more collect query flexibility. As of now, this is quite rigid and doesn't give the Collector much room for mistakes.]]

4.6.5.1. Time-interval Queries

4.6.5.1.1. Boundary Check

The batch boundaries are determined by the time_precision field of the query configuration. For the batch_interval included with the query, the Aggregator checks that:

* batch_interval.duration >= time_precision (this field determines, effectively, the minimum batch duration)

* both batch_interval.start and batch_interval.duration are divisible by time_precision

These measures ensure that Aggregators can efficiently "pre-aggregate" output shares recovered during the aggregation interaction.

4.6.5.1.2. Size Check

The query configuration specifies the minimum batch size, min_batch_size. The Aggregator checks that len(X) >= min_batch_size, where X is the set of reports successfully aggregated into the batch.

4.6.5.2. Fixed-size Queries

4.6.5.2.1. Boundary Check

For fixed_size tasks, the batch boundaries are defined by opaque batch IDs. Thus the Aggregator needs to check that the query is associated with a known batch ID:

* For a CollectionReq containing a query of type by_batch_id, the Leader checks that the provided batch ID corresponds to a batch ID it returned in a previous collection for the task.

* For an AggregateShareReq, the Helper checks that the batch ID provided by the Leader corresponds to a batch ID used in a previous AggregationJobInitReq for the task.

4.6.5.2.2. Size Check

The query configuration specifies the minimum batch size, min_batch_size, and optionally the maximum batch size, max_batch_size. The Aggregator checks that len(X) >= min_batch_size. And if max_batch_size is specified, also len(X) <= max_batch_size, where X is the set of reports successfully aggregated into the batch.

5. Operational Considerations

The DAP protocol has inherent constraints derived from the tradeoff between privacy guarantees and computational complexity. These tradeoffs influence how applications may choose to utilize services implementing the specification.

5.1. Protocol Participant Capabilities

The design in this document has different assumptions and requirements for different protocol participants, including Clients, Aggregators, and Collectors. This section describes these capabilities in more detail.

5.1.1. Client Capabilities

Clients have limited capabilities and requirements. Their only inputs to the protocol are (1) the parameters configured out of band and (2) a measurement. Clients are not expected to store any state across any upload flows, nor are they required to implement any sort of report upload retry mechanism. By design, the protocol in this document is robust against individual Client upload failures since the protocol output is an aggregate over all inputs.

5.1.2. Aggregator Capabilities

Leaders and Helpers have different operational requirements. The design in this document assumes an operationally competent Leader, i.e., one that has no storage or computation limitations or constraints, but only a modestly provisioned Helper, i.e., one that has computation, bandwidth, and storage constraints. By design, Leaders must be at least as capable as Helpers, where Helpers are generally required to:

- Support the aggregate interaction, which includes validating and aggregating reports; and

- Publish and manage an HPKE configuration that can be used for the upload protocol.

In addition, for each DAP task, the Helper is required to:

* Implement some form of batch-to-report index, as well as inter- and intra-batch replay mitigation storage, which includes some way of tracking batch report size. Some of this state may be used for replay attack mitigation. The replay mitigation strategy is described in Section 4.5.1.4.

Beyond the minimal capabilities required of Helpers, Leaders are generally required to:

* Support the upload protocol and store reports; and

* Track batch report size during each collect flow and request encrypted output shares from Helpers.

In addition, for each DAP task, the Leader is required to:

* Implement and store state for the form of inter- and intra-batch replay mitigation in Section 4.5.1.4.

5.1.3. Collector Capabilities

Collectors statefully interact with Aggregators to produce an aggregate output. Their input to the protocol is the task parameters, configured out of band, which include the corresponding batch window and size. For each collect invocation, Collectors are required to keep state from the start of the protocol to the end as needed to produce the final aggregate output.

Collectors must also maintain state for the lifetime of each task, which includes key material associated with the HPKE key configuration.

5.2. VDAFs and Compute Requirements

The choice of VDAF can impact the computation required for a DAP task. For instance, the Poplar1 VDAF [VDAF] when configured to compute a set of heavy hitters requires each measurement to be of the same bit-length which all parties need to agree on prior to VDAF execution. The computation required for such tasks increases superlinearly as a function of the chosen bit-length, as multiple rounds of collection are needed for each bit of the measurement value, and each bit of the measurement value requires computation which scales with the chosen bit-length.

When dealing with variable length measurements (e.g domain names), it is necessary to pad them to convert into fixed-length measurements. When computing the heavy hitters from a batch of such measurements, VDAF execution can be terminated early once once the

padding region is reached for a given measurement: at this point, the padded measurement can be included in the last set of candidate prefixes. For smaller length measurements, this significantly reduces the cost of communication between Aggregators and the steps required for the computation. However, malicious Clients can still generate maximum length measurements forcing the system to always operate at worst-case performance.

Therefore, care must be taken that a DAP deployment can handle VDAF execution of all possible measurement values for any tasks that the deployment is configured for. Otherwise, an attacker may deny service by uploading many expensive reports.

Applications which require computationally-expensive VDAFs can mitigate the computation cost of aggregation in a few ways, such as producing aggregates over a sample of the data or choosing a representation of the data permitting a simpler aggregation scheme.

[[TODO: Discuss explicit key performance indicators, here or elsewhere.]]

5.3. Aggregation Utility and Soft Batch Deadlines

A soft real-time system should produce a response within a deadline to be useful. This constraint may be relevant when the value of an aggregate decreases over time. A missed deadline can reduce an aggregate's utility but not necessarily cause failure in the system.

An example of a soft real-time constraint is the expectation that input data can be verified and aggregated in a period equal to data collection, given some computational budget. Meeting these deadlines will require efficient implementations of the input-validation protocol. Applications might batch requests or utilize more efficient serialization to improve throughput.

Some applications may be constrained by the time that it takes to reach a privacy threshold defined by a minimum number of reports. One possible solution is to increase the reporting period so more samples can be collected, balanced against the urgency of responding to a soft deadline.

5.4. Protocol-specific Optimizations

Not all DAP tasks have the same operational requirements, so the protocol is designed to allow implementations to reduce operational costs in certain cases.

5.4.1. Reducing Storage Requirements

In general, the Aggregators are required to keep state for tasks and all valid reports for as long as collect requests can be made for them. In particular, Aggregators must store a batch as long as the batch has not been queried more than max_batch_query_count times. However, it is not always necessary to store the reports themselves. For schemes like Prio3 [VDAF] in which reports are verified only once, each Aggregator only needs to store its aggregate share for each possible batch interval, along with the number of times the aggregate share was used in a batch. This is due to the requirement that the batch interval respect the boundaries defined by the DAP parameters. (See Section 4.6.5.)

However, Aggregators are also required to implement several per-report checks that require retaining a number of data artifacts. For example, to detect replay attacks, it is necessary for each Aggregator to retain the set of report IDs of reports that have been aggregated for the task so far. Depending on the task lifetime and report upload rate, this can result in high storage costs. To alleviate this burden, DAP allows Aggregators to drop this state as needed, so long as reports are dropped properly as described in Section 4.5.1.4. Aggregators **SHOULD** take steps to mitigate the risk of dropping reports (e.g., by evicting the oldest data first).

Furthermore, the Aggregators must store data related to a task as long as the current time has not passed this task's task_expiration. Aggregator **MAY** delete the task and all data pertaining to this task after task_expiration. Implementors **SHOULD** provide for some leeway so the Collector can collect the batch after some delay.

6. Compliance Requirements

In the absence of an application or deployment-specific profile specifying otherwise, a compliant DAP application **MUST** implement the following HPKE cipher suite:

*KEM: DHKEM(X25519, HKDF-SHA256) (see [HPKE], Section 7.1)

*KDF: HKDF-SHA256 (see [HPKE], Section 7.2)

*AEAD: AES-128-GCM (see [HPKE], Section 7.3)

7. Security Considerations

DAP aims to achieve the privacy and robustness security goals defined in Section 9 of [VDAF], even in the presence of an active attacker. It is assumed that the attacker may control the network and have the ability to control a subset of of Clients, one of the Aggregators, and the Collector for a given task.

In the presence of this adversary, there are some threats DAP does not defend against and which are considered outside of DAP's threat model. These are enumerated below, along with potential mitigations.

Attacks on robustness:

1. Aggregators can defeat robustness by emitting incorrect aggregate shares, by omitting reports from the aggregation process, or by manipulating the VDAF preparation process for a single report. DAP follows VDAF in providing robustness only if both Aggregators honestly follow the protocol.

2. Clients may affect the quality of aggregate results by reporting false measurements. A VDAF can only verify that a submitted measurement is valid, not that it is true.

3. An attacker can impersonate multiple Clients, or a single malicious Client can upload an unexpectedly-large number of reports, in order to skew aggregate results or to reduce the number of measurements from honest Clients in a batch below the minimum batch size. See Section 7.1 for discussion and potential mitigations.

Attacks on privacy:

1. Clients can intentionally leak their own measurements and compromise their own privacy.

2. Both Aggregators together can, purposefully or accidentally, share unencrypted input shares in order to defeat the privacy of individual reports. DAP follows VDAF in providing privacy only if at least one Aggregator honestly follows the protocol.

Attacks on other properties of the system:

1. Both Aggregators together can, purposefully or accidentally, share unencrypted aggregate shares in order to reveal the aggregation result for a given batch.

2. Aggregators, or a passive network attacker between the Clients and the Leader, can examine metadata such as HTTP client IP in order to infer which Clients are submitting reports. Depending on the particulars of the deployment, this may be used to infer sensitive information about the Client. This can be mitigated for the Aggregator by deploying an anonymizing proxy (see Section 7.3), or in general by requiring Clients to submit reports at regular intervals independently of the measurement value such that the existence of a report does not imply the occurrence of a sensitive event.

3. Aggregators can deny service by refusing to respond to collection requests or aggregate share requests.

4. Collection requests may leak information beyond the collection results. For example, the Poplar1 VDAF [VDAF] can be used to compute the set of heavy hitters among a set of arbitrary bit strings uploaded by Clients. This requires multiple evaluations of the VDAF, the results of which reveal information to the Aggregators and Collector beyond what follows from the heavy hitters themselves. Or the result could leak information about outlier values. This leakage can be mitigated using differential privacy (Section 7.4).

7.1. Sybil Attacks

Several attacks on privacy or robustness involve malicious Clients uploading reports that are valid under the chosen VDAF but incorrect.

For example, a DAP deployment might be measuring the heights of a human population and configure a variant of Prio3 to prove that measurements are values in the range of 80-250 cm. A malicious Client would not be able to claim a height of 400 cm, but they could submit multiple bogus reports inside the acceptable range, which would yield incorrect averages. More generally, DAP deployments are susceptible to Sybil attacks [Dou02], especially when carried out by the Leader.

In this type of attack, the adversary adds to a batch a number of reports that skew the aggregate result in its favor. For example, sending known measurements to the Aggregators can allow a Collector to shrink the effective anonymity set by subtracting the known measurements from the aggregate result. The result may reveal additional information about the honest measurements, leading to a privacy violation; or the result may have some property that is desirable to the adversary ("stats poisoning").

Depending on the deployment and the specific threat being mitigated, there are different ways to address Sybil attacks, such as:

1. Implementing Client authentication, as described in Section 7.2, likely paired with rate-limiting uploads from individual Clients.

2. Removing Client-specific metadata on individual reports, such as through the use of anonymizing proxies in the upload flow, as described in Section 7.3.

3. Differential privacy (Section 7.4) can help mitigate Sybil attacks to some extent.

7.2. Client Authentication

In settings where it is practical for each Client to have an identity provisioned (e.g., a user logged into a backend service or a hardware device programmed with an identity), Client authentication can help Aggregators (or an authenticating proxy deployed between Clients and the Aggregators; see Section 7.3) ensure that all reports come from authentic Clients. Note that because the Helper never handles messages directly from the Clients, reports would need to include an extension (Section 4.4.3) to convey authentication information to the Helper. For example, a deployment might include a Privacy Pass token ([I-D.draft-ietf-privacypass-architecture-16]) in an extension to allow both Aggregators to independently verify the Client's identity.

However, in some deployments, it will not be practical to require Clients to authenticate, so Client authentication is not mandatory in DAP. For example, a widely distributed application that does not require its users to log in to any service has no obvious way to authenticate its report uploads.

7.3. Anonymizing Proxies

Client reports can contain auxiliary information such as source IP, HTTP user agent, or Client authentication information (in deployments which use it, see Section 7.2). This metadata can be used by Aggregators to identify participating Clients or permit some attacks on robustness. This auxiliary information can be removed by having Clients submit reports to an anonymizing proxy server which would then use Oblivious HTTP [RFC9458] to forward reports to the DAP Leader. In this scenario, Client authentication would be performed by the proxy rather than any of the participants in the DAP protocol.

7.4. Differential Privacy

DAP deployments can choose to ensure their aggregate results achieve differential privacy ([Vad16]). A simple approach would require the Aggregators to add two-sided noise (e.g. sampled from a two-sided geometric distribution) to aggregate shares. Since each Aggregator is adding noise independently, privacy can be guaranteed even if all but one of the Aggregators is malicious. Differential privacy is a strong privacy definition, and protects users in extreme circumstances: even if an adversary has prior knowledge of every measurement in a batch except for one, that one measurement is still formally protected.

7.5. Task Parameters

Distribution of DAP task parameters is out of band from DAP itself and thus not discussed in this document. This section examines the security tradeoffs involved in the selection of the DAP task parameters. Generally, attacks involving crafted DAP task parameters can be mitigated by having the Aggregators refuse shared parameters that are trivially insecure (e.g., a minimum batch size of 1 report).

7.5.1. VDAF Verification Key Requirements

Knowledge of the verification key would allow a Client to forge a report with invalid values that will nevertheless pass verification. Therefore, the verification key must be kept secret from Clients.

Furthermore, for a given report, it may be possible to craft a verification key which leaks information about that report's measurement during VDAF preparation. Therefore, the verification key for a task **SHOULD** be chosen before any reports are generated. Moreover, it **SHOULD** be fixed for the lifetime of the task and not be rotated. One way to ensure that the verification key is generated independently from any given report is to derive the key based on the task ID and some previously agreed upon secret (verify_key_seed) between Aggregators, as follows:

```
verify_key = HKDF-Expand(
    HKDF-Extract(
        "verify_key",    # salt
        verify_key_seed, # IKM
    ),
    task_id,             # info
    VERIFY_KEY_SIZE,     # L
)
```

Here, VERIFY_KEY_SIZE is the length of the verification key, and HKDF-Extract and HKDF-Expand are as defined in [RFC5869].

This requirement comes from current security analysis for existing VDAFs. In particular, the security proofs for Prio3 require that the verification key is chosen independently of the generated reports.

7.5.2. Batch Parameters

An important parameter of a DAP deployment is the minimum batch size. If a batch includes too few reports, then the aggregate result can reveal information about individual measurements. Aggregators enforce the agreed-upon minimum batch size during the collection protocol, but implementations **SHOULD** also opt out of participating in a DAP task if the minimum batch size is too small. This document

does not specify how to choose an appropriate minimum batch size, but an appropriate value may be determined from the differential privacy (Section 7.4) parameters in use, if any.

If batches will be collected more than once, it is possible that some invalid reports could be accepted and successfully aggregated into the first collection, but be rejected with a subsequent aggregation parameter. This could reduce the number of successfully aggregated reports below the minimum batch size, and make the batch uncollectable with that aggregation parameter, as described in Section 4.1.3. This constitutes an efficient denial of service vector for a Sybil attacker. In deployments that will collect batches more than once, the actual batch sizes, as determined by the Leader or Collector (depending on query type) **SHOULD** be greater than the minimum batch size to allow for later rejection of some number of invalid reports.

7.5.3. Task Configuration Agreement and Consistency

In order to execute a DAP task, it is necessary for all parties to ensure they agree on the configuration of the task. However, it is possible for a party to participate in the execution of DAP without knowing all of the task's parameters. For example, a Client can upload a report (Section 4.4) without knowing the minimum batch size that is enforced by the Aggregators during collection (Section 4.6).

Depending on the deployment model, agreement can require that task parameters are visible to all parties such that each party can choose whether to participate based on the value of any parameter. This includes the parameters enumerated in Section 4.2 and any additional parameters implied by upload extensions Section 4.4.3 used by the task. Since meaningful privacy requires that multiple Clients contribute to a task, they should also share a consistent view of the task configuration.

7.6. Infrastructure Diversity

DAP deployments should ensure that Aggregators do not have common dependencies that would enable a single vendor to reassemble measurements. For example, if all participating Aggregators stored unencrypted input shares on the same cloud object storage service, then that cloud vendor would be able to reassemble all the input shares and defeat privacy.

8. IANA Considerations

8.1. Protocol Message Media Types

This specification defines the following protocol messages, along with their corresponding media types types:

- *HpkeConfigList Section 4.4.1: "application/dap-hpke-config-list"
- *Report Section 4.4.2: "application/dap-report"
- *AggregationJobInitReq Section 4.5.1.1: "application/dap-aggregation-job-init-req"
- *AggregationJobResp Section 4.5.1.2: "application/dap-aggregation-job-resp"
- *AggregationJobContinueReq Section 4.5.2.1: "application/dap-aggregation-job-continue-req"
- *AggregateShareReq Section 4.6: "application/dap-aggregate-share-req"
- *AggregateShare Section 4.6: "application/dap-aggregate-share"
- *CollectionReq Section 4.6: "application/dap-collect-req"
- *Collection Section 4.6: "application/dap-collection"

The definition for each media type is in the following subsections.

Protocol message format evolution is supported through the definition of new formats that are identified by new media types.

IANA [shall update / has updated] the "Media Types" registry at https://www.iana.org/assignments/media-types with the registration information in this section for all media types listed above.

[OPEN ISSUE: Solicit review of these allocations from domain experts.]

8.1.1. "application/dap-hpke-config-list" media type

Type name: application

Subtype name: dap-hpke-config-list

Required parameters: N/A

Optional parameters: None

Encoding considerations:
only "8bit" or "binary" is permitted

Security considerations: see Section 4.2

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.2. "application/dap-report" media type

Type name: application

Subtype name: dap-report

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.4.2

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type:
N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.3. "application/dap-aggregation-job-init-req" media type

Type name: application

Subtype name: dap-aggregation-job-init-req

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type:
N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.4. "application/dap-aggregation-job-resp" media type

Type name: application

Subtype name: dap-aggregation-job-resp

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

- **Magic number(s):** N/A
- **Deprecated alias names for this type:** N/A
- **File extension(s):** N/A
- **Macintosh file type code(s):** N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.5. "application/dap-aggregation-job-continue-req" media type

Type name: application

Subtype name: dap-aggregation-job-continue-req

Required parameters: N/A

Optional parameters:
None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.6. "application/dap-aggregate-share-req" media type

Type name: application

Subtype name: dap-aggregate-share-req

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification:
this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.7. "application/dap-aggregate-share" media type

Type name: application

Subtype name: dap-aggregate-share

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s):
N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.8. "application/dap-collect-req" media type

Type name: application

Subtype name: dap-collect-req

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.9. "application/dap-collection" media type

Type name: application

Subtype name: dap-collection

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 4.6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author:
see Authors' Addresses section

Change controller: IESG

8.2. DAP Type Registries

This document also requests creation of a new "Distributed Aggregation Protocol Parameters" page. This page will contain several new registries, described in the following sections.

8.2.1. Query Types Registry

This document requests creation of a new registry for Query Types. This registry should contain the following columns:

[TODO: define how we want to structure this registry when the time comes]

8.2.2. Upload Extension Registry

This document requests creation of a new registry for extensions to the Upload protocol. This registry should contain the following columns:

[TODO: define how we want to structure this registry when the time comes]

8.2.3. Prepare Error Registry

This document requests creation of a new registry for PrepareError values. This registry should contain the following columns:

Name: The name of the PrepareError value

Value: The 1-byte value of the PrepareError value

Reference: A reference to where the PrepareError type is defined.

The initial contents of this registry are as defined in Section 4.5.1.2, with this document as the reference.

8.3. URN Sub-namespace for DAP (urn:ietf:params:ppm:dap)

The following value [will be/has been] registered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry, following the template in [RFC3553]:

Registry name: dap

Specification: [[THIS DOCUMENT]]

Repository: http://www.iana.org/assignments/dap

Index value: No transformation needed.

Initial contents: The types and descriptions in the table in Section 3.2 above, with the Reference field set to point to this specification.

Contributors

Josh Aas
ISRG
josh@abetterinternet.org

Junye Chen
Apple
junyec@apple.com

David Cook
ISRG
dcook@divviup.org

Charlie Harrison
Google
csharrison@chromium.org

Peter Saint-Andre
stpeter@gmail.com

Shivan Sahib
Brave
shivankaulsahib@gmail.com

Phillipp Schoppmann
Google
schoppmann@google.com

Martin Thomson
Mozilla
mt@mozilla.com

Shan Wang
Apple
shan_wang@apple.com

References

Normative References

[HPKE] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[OAuth2] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/rfc/rfc6749>.

[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>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <https://www.rfc-editor.org/rfc/rfc3553>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <https://www.rfc-editor.org/rfc/rfc5861>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/rfc/rfc7807>.

[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>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/

RFC9111, June 2022, <https://www.rfc-editor.org/rfc/rfc9111>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/rfc/rfc9457>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/rfc/rfc9458>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[VDAF] Barnes, R., Cook, D., Patton, C., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", Work in Progress, Internet-Draft, draft-irtf-cfrg-vdaf-08, 20 November 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-08>.

Informative References

[BBCGGI19] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs", 5 January 2021, <https://eprint.iacr.org/2019/188>.

[BBCGGI21] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Lightweight Techniques for Private Heavy Hitters", 5 January 2021, <https://eprint.iacr.org/2021/017>.

[CGB17] Corrigan-Gibbs, H. and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics", 14 March 2017, <https://crypto.stanford.edu/prio/paper.pdf>.

[Dou02] Douceur, J., "The Sybil Attack", 10 October 2022, <https://link.springer.com/chapter/10.1007/3-540-45748-8_24>.

[I-D.draft-dcook-ppm-dap-interop-test-design-04] Cook, D., "DAP Interoperation Test Design", Work in Progress, Internet-Draft, draft-dcook-ppm-dap-interop-test-design-04, 14 June 2023, <https://datatracker.ietf.org/doc/html/draft-dcook-ppm-dap-interop-test-design-04>.

[I-D.draft-ietf-privacypass-architecture-16] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", Work

in Progress, Internet-Draft, draft-ietf-privacypass-architecture-16, 25 September 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-16>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[Vad16] Vadhan, S., "The Complexity of Differential Privacy", 9 August 2016, <https://privacytools.seas.harvard.edu/files/privacytools/files/complexityprivacy_1.pdf>.

Authors' Addresses

Tim Geoghegan
ISRG

Email: timgeog+ietf@gmail.com

Christopher Patton
Cloudflare

Email: chrispatton+ietf@gmail.com

Brandon Pitman
ISRG

Email: bran@bran.land

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net